# Distributed Dynamic Slicing of Java Programs

Durga Prasad Mohapatra,
Department of Computer Science and Engineering
National Institute of Technology
Rourkela - 769008, India
e-mail: durga@nitrkl.ac.in

Rajib Mall, Rajeev Kumar[*], D.S. Kumar, Mayank Bhasin
Department of Computer Science and Engineering
Indian Institute of Technology Kharagpur
Kharagpur - 721302, India
e-mail: {rajib, rkumar}@cse.iitkgp.ernet.in

**Abstract**

We propose a novel dynamic slicing technique for distributed Java programs. We first construct the intermediate representation of a distributed Java program in the form of a set of *Distributed Program Dependence Graphs* (DPDG). We mark and unmark the edges of the DPDG appropriately as and when dependencies arise and cease during run-time. Our algorithm can run parallely on a network of computers, with each node in the network contributing to the dynamic slice in a fully distributed fashion. Our approach does not require any trace files to be maintained. Another advantage of our approach is that a slice is available even before a request for a slice is made. This appreciably reduces the response time of slicing commands. We have implemented the algorithm in a distributed environment. The results obtained from our experiments show promise.

**Key words:** program slicing, dynamic slicing, program dependence graph, debugging, object-oriented program, multithreading, Java, distributed programming.

## 1 Introduction

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. A program slice with respect to a specified variable $v$ at some program point $P$ consists of those parts of the program which potentially affect the value of $v$ at $p$. The pair $< v, p >$ is known as the *slicing criterion*. A static slice is valid for all possible executions of a program, while a dynamic slice is meaningful for only a particular execution of a program [1, 2]. Program slicing has been found to be useful in a variety of applications such as debugging, program understanding, testing and maintenance [3–9].

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

[*]Corresponding author, Email: rkumar@cse.iitkgp.ernet.in; Tel: +91 3222 283464; Fax: +91 3222 278985

for these distributed systems seem to gain utmost importance.

Development of real life distributed object-oriented programs presents formidable challenge to the programmer. It is usually accepted that understanding and debugging of distributed object-oriented programs are much harder compared to those of sequential programs. The non-deterministic nature of distributed programs, lack of global states, unsynchronized interactions among threads, multiple threads of control and a dynamically varying number of processes are some reasons for this difficulty. 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. Many research reports addressed efficient handling of data structures such as arrays, pointers etc. in the sequential framework. Attempts have also been made to deal with unstructured constructs like goto, break etc. Although researchers have reported extension of the traditional concept of program slicing to static slicing of distributed programs, dynamic slicing of distributed object-oriented programs has scarcely been reported in the literature.

Any effective dynamic slicing technique for distributed object-oriented programs needs to address the important concepts associated with object-oriented programs such as encapsulation, inheritance, message passing and polymorphism etc. This poses new challenges during slice computation which are not encountered in traditional program slicing and render representation and analysis techniques developed for imperative language programs inadequate. So, the object-oriented features need to be considered carefully in the process of slicing.

We have already mentioned that object-oriented programs are often large. Therefore, to be practically useful in interactive applications such as debugging, program traces should be avoided in the slicing process. Maintaining execution traces become unacceptable due to slow $I/O$ operations. Further, to be useful in a distributed environment, the construction of slices should preferably be constructed in a distributed manner. Each node in a distributed system should contribute to the slice by determining its local portion of the global slice in a fully distributed fashion.

Keeping the above identified objectives in mind, in this paper we propose an algorithm for computing dynamic slices of distributed Java programs. Though we have considered only Java programs, programs in any other language can be handled by making only small changes to our algorithm. We have concentrated only on the *communication* and *concurrency* issues in Java. Standard sequential and object-oriented features are not discussed in this paper, as these are easily found in the literature [10–14]. For example, Larson and Harrold [10] have discussed the techniques to represent the basic object-oriented features. Their technique can easily be incorporated into our algorithm to represent the basic object-orientation features.

We have named our proposed algorithm *distributed dynamic slicing* (DDS) algorithm for Java programs. To achieve fast response time, our algorithm can run in a fully distributed manner on several machines connected through a network, rather than running it on a centralized machine. We use 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.

Our algorithm uses a modified program dependence graph (PDG) [15] as the intermediate representation. We have named this representation *distributed program dependence graph* (DPDG). We first statically construct the DPDG before run-time. Our 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 our approach 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.

The rest of the paper is organized as follows. In section 2 we present some basic concepts and definitions that will be used in our algorithm. In section 3, we discuss the intermediate program representation: *distributed program dependence graph* (DPDG). In section 4, we present our *distributed dynamic slicing* (DDS) algorithm for distributed object-oriented programs. In section 5, we briefly present an implementation of our algorithm. In section 6, we compare our algorithm with related algorithms. Section 7 concludes the paper.

## 2 Basic Concepts

Before presenting our dynamic slicing algorithm, we first briefly discuss the relevant features of Java. Then, we introduce a few basic concepts and definitions that would be used in our algorithm. In the following discussions and throughout the rest of the paper, we use the terms a *program statement*, a *node* and a *vertex* interchangeably.

### 2.1 Concurrency and Communication in Java

Java supports concurrent programming using threads. A thread is a single sequential flow of control within a program. A thread is similar to a sequential program in the sense that each thread also has a beginning, an execution sequence and an end. Also, at any given time during the run of a thread, there is a single point of execution. However, a thread itself is not a program; it can not run on its own. To support thread programming, Java provides a *Thread* class library, which defines a set of standard operations on a thread such as *start*(), *stop*(), *join*(), *suspend*(), *resume*() and *sleep*() etc. [16].

Java supports communication among threads both through shared memory and message passing. Objects shared by two or more threads are called *condition variables*. Access to these variables must be synchronized for the proper working of the system. The Java language and run-time system support thread synchronization through the use of *monitors*. A *monitor* is associated with a specific data item and functions to lock that data. When a thread holds the monitor for some data item, other threads can not inspect or modify the data. The code segments within a program that access the same data from within separate threads are known as *critical sections*. In Java programs, critical sections need to be marked with the keyword *synchronized* for synchronized access to shared data. Java provides the methods *wait*(), *notify*(), and *notifyall*() to support synchronization among different threads [16].

When a thread needs to send a message to another thread, it calls the method *getOutputStream*(). To receive a message, the receiving thread calls the method *getInputStream*(). Java provides *sockets* to support distributed programming among component programs running on different machines. By using the key word *Socket*, a client program can specify the *ip* address and the *port number* of a sever program with which it wants to communicate [16].

A *distributed object-oriented program* $P = (P_1, P_2, \ldots, P_n)$ is a collection of concurrent individual programs $P_i$ such that each of the $P_i$'s may communicate with other programs through the reception and transmission of messages. We refer to the individual programs $P_i$ as the *component programs*. We assume asynchronous (non-blocking) send and synchronous (blocking) receive message passing among component programs, in our DDS algorithm. However, other models can easily be considered through minor alterations to our proposed algorithm. Each component program may contain multiple threads. We assume use of *sockets* for message passing among threads of different component programs, and assume use of *shared objects* for message passing among different threads within a single component program.

We assume that the number of nodes on which a distributed object-oriented program runs, is predefined. We make no assumptions on the order in which messages arrive at the destination once they are sent. The only assumption we make is that messages sent by one thread to another are received in the same order as dispatched by the sending thread. However, messages sent concurrently from different threads to one thread may arrive in any arbitrary order. All messages that arrive at a thread are collected in a *message queue*. A thread executing a *getInputStream*() call removes the oldest message that is available at the front of the queue. The receiving thread waits for the sending thread to put a new entry in the queue, if the queue is empty. Here, communication is non-deterministic in the sense that, the receiving thread continues with its execution by selecting whichever message arrives first.

We explain the *message passing* mechanism in distributed Java programs by taking a sample distributed Java program. We subsequently use the example program to explain the notations we have used in our algorithm. We construct the intermediate representation of this example program in the next section. Finally, we use this example program to explain the working of our proposed algorithm.

Let us consider the distributed Java program shown in Fig. 1 and Fig. 2. In this example, Fig. 1 represents a *client program* and Fig. 2 represents a *server program*. The *client program* specifies the IP address of the machine where the *server program* runs and the port number for connection. The *client program* reads the value of the integer variable $x$ from the key board and performs some arithmetic computations using it. Then, it sends the results of the computation to the *server program* through a *socket*. The client program in Fig. 1 contains one thread called *clthd*. The *server program* in Fig. 2 contains two threads *thread1* and *thread2*. We distinguish the threads by assigning unique thread-ids to each of the threads. The thread *thread1* receives the result sent by the *client program* and performs some arithmetic computations. Then, *thread1* sends the results to *thread2* through a shared object *obj*. The thread *thread2* performs some computations using the shared object *obj* and sends the results to the *client program* through a *socket*. Using this result, the *client program* performs the final computations and displays the computed value.

```
1. class clthd extends Thread {
2.   BufferedReader rcvmsg;
3.   PrintWriter sendmsg;
4.   BufferedReader in=new BufferedReader(new InputStreamReader(system.in));
5.   Socket socket;
6.   public void run() {
7.      socket=new Socket("10.5.18.49",1500);  // connecting to server at given ip and port no.s
8.      sendmsg=new PrintWriter(socket.getOutputStream());  // declaration of sendmsg
9.      rcvmsg=new BufferedReader(new InputStreamReader (socket.getInputStream()));   // declaration
10.     String s=in.readLine();
11.     int x=Integer.parseInt(s);
12.     int z,q,y=15;
13.     if(x>y)
14.         z=x−y;
        else
15.         z=x+y;
16.     sendmsg.println(z);        // sending value of z to server
17.     System.out.println("value of z is: "+z);
18.     String msgfrom_server=rcvmsg.readLine();        //  receiving value from server
19.     int p=Integer.parseInt(msgfrom_server);
20.     if(p>x)
21.         q=p−x;
        else
22.         q=p+x;
23.     System.out.println("total is: "+q);
     }
24.  public class client {
25.     public static void main(String args[]) {
26.         clthd t=new clthd();
27.         t.start();
         }
      }
   }
```

Figure 1: An Example Client Program

```
1. class shar {
2.    int s;
3.    boolean flag=true;
4.    synchronized public void put(int c) {
5.      s=c;
6.      notify();
7.      flag=false;
      }
8.    synchronized public int get() {
9.       if(flag==true)
10.          wait();
11.      return s;
      }
   }
12. class thread1 extends Thread {
13.    BufferedReader rcvmsg;
14.    PrintWriter sendmsg;
15.    Socket serv_socket;
16.    int b,c;
17.    shar obj;           // declaration for shared object
18.    BufferedReader o=new BufferedReader(
          new InputStreamReader(System.in));
19.    public thread1(Socket req,BufferedReader in,
             PrintWriter out,shar ob) {
20.    serv_socket=req;
21.    sendmsg=out;
22.    rcvmsg=in;
23.    obj=ob;
     }
24.    public void run() {
25.      String msg=rcvmsg.readLine();  // receiving message from client prog
26.      int a=Integer.parseInt(msg);
27.      System.out.println("received from client is: "+a);
28.      String mss=o.readLine();
29.      int b=Integer.parseInt(mss);
30.      if(a>b)
31.           c=a−b;
         else
32.           c=a+b;
33.      obj.put(c);   // sending the value of c to thread2
34.      System.out.println("thd1: "+c);
     }          }
35. class thread2 extends Thread  {
36.     BufferedReader rcvmsg;
37.     PrintWriter sendmsg;
38.     Socket serv_socket;
39.     shar obj;   // declaration for shared object
40.     BufferedReader o=new BufferedReader(
           new InputStreamReader(System.in));
41.     public thread2(Socket req,BufferedReader in,
             PrintWriter out,shar ob) {
42.         serv_socket=req;
43.         sendmsg=out;
44.         rcvmsg=in;
45.         obj=ob;       }
46.     public void run() {
47.        int e,g,f=10;
48.        e=obj.get();    // receiving the value from thread1
49.        if(e>f)
50.            g=e−f;
           else
51.            g=e+f;
52.        sendmsg.println(g);    }     }  // sending value of g to client
53. public class syn_server {
54.    public static void main(String args[]) {
55.       ServerSocket serv_socket;
56.       BufferedReader rcvmsg;
57.       PrintWriter sendmsg;
58.       shar obj=new shar();
59.       serv_socket=new ServerSocket(1500);  // creating a new port
60.       Socket socket=serv_socket.accept();   // accepts client
61.       sendmsg=new PrintWriter(socket.
                  getOutputStream(),true);  // declaration for sendmsg
62.       rcvmsg=new BufferedReader(new
             InputStreamReader(socket.getInputStream()));  // declaration for rcvmsg
63.      thread1 t1=new thread1(socket,input,output,obj);
64.      thread1 t2=new thread2(socket,input,output,obj);
65.      t1.start();
66.      t2.start();
       } }
```

Figure 2: An Example Server Program

## 2.2 Notations and Terminologies

We now introduce a few terms and notations which would be used through out the rest of the paper.

**D1.** *Precise Dynamic Slice.* A dynamic slice is said to be *precise* iff it includes only those statements that actually affect the value of a variable at a program point for the given execution. It is very difficult to determine whether a given slice is precise or not since determining a precise slice is an undecidable problem [17]. However, using the notion of a precise slice, we can determine whether a given slice is more precise than another for most cases excepting a few pathogenic cases.

**D2.** *Correct Dynamic Slice.* A dynamic slice is said to be *correct* iff it contains all the statements of the program that affect the slicing criterion. A dynamic slice is said to be *incorrect* iff it fails to include some statements of the program that affect the slicing criterion. Note that the whole program is always a correct slice for any slicing criterion. A correct slice is *imprecise* if and only if it contains at least one statement that does not affect the slicing criterion.

**D3.** *def(var).* Let *var* be an instance variable in a class in an object-oriented program. A node $x$ is said to be a *def(var)* node, if $x$ represents a definition for the variable *var.*
In Fig. 1 the node 14 is a *def(z)* node.

**D4.** *defSet(var).* The set defSet(var) denotes the set of all def(var) nodes.
In Fig. 1 $defSet(z) = \{14, 15\}$.

**D5.** *use(var) node.* Let *var* be a variable defined in a class in an object-oriented program. A node $x$ is said to be a *use(var)* node, iff it uses the variable *var.*
In Fig. 1, node 17 is a *use(z)* node.

**D6.** *recentDef(thread, var).* Let $s$ be a def(var) node of a component program $P_i$. Let $p_i$ and $p_j$ be threads in $P_i$ . Then, $recentDef(p_i, var)$ represents the most recent definition of the variable *var* available to the thread $p_i$. Further, $recentDef(p_i, var) = (p_j, s)$ indicates that the most recent definition of the variable *var* in thread $p_j$ is also the most recent definition of the variable *var* with respect to thread $p_i$. $p_i$ may or may not be same as $p_j$, and the variable *var* can either be a local or a shared variable.

**D7.** *Distributed Control Flow Graph (DCFG).* A *distributed control flow graph* (DCFG) $G$ of a component program $P_i$ of a distributed program P = $(P_1, \ldots, P_n)$ is a flow graph *(N, E, Start, Stop)*, where each node $n \in N$ represents a statement of $P_i$, and each edge $e \in E$ represents potential control transfer among the nodes. Nodes *Start* and *Stop* are two unique nodes representing entry and exit nodes of the component program $P_i$ respectively. There is a directed edge representing a control flow from node $a$ to node $b$ if control may flow from node $a$ to node $b$.

**D8.** *Post Dominance.* Let $x$ and $y$ be two nodes in a CCFG $G$. Node $y$ *post dominates* node $x$ *iff* every directed path from $x$ to *stop* passes through $y$.

**D9.** *Control Dependence.* Let $G$ be a *DCFG* and $x$ be a *test* (predicate) node. A node $y$ is said to be *control dependent* on a node $x$ *iff* there exists a directed path $D$ from $x$ to $y$ such that

1. $y$ *post dominates* every node $z \neq x$ in $D$.
2. $y$ does not *post dominate* $x$.

**D10.** *Data Dependence.* Let $x$ be a *def(var)* node and $y$ be a *use(var)* node in a DCFG $G$. A node $y$ is said to be *data dependent* on a node $x$, *iff* there exists a directed path $D$ from $x$

to $y$ such that there is no intervening *def(var)* node in $D$.

**D11.** *Thread Dependence.* For a DCFG $G$, let $x$ be the node representing the *run*() statement of thread $p_i$. A node $y$ is said to be *thread* dependent on $x$, *iff* there exists a directed path $D$ from $x$ to $y$ such that none of the nodes in $D$ is a *run* node.

**D12.** *Synchronization Dependence.* A statement $y$ in a thread is *synchronization dependent* on a statement $x$ in another thread, iff execution of $y$ is dependent on execution of $x$ due to a *synchronization* operation.

Let $y$ be a *wait*() node in thread $t_1$ and $x$ be the corresponding *notify*() node in thread $t_2$. Then the node $y$ is said to be *synchronization dependent* on node $x$. For example, in Fig. 2, node 10 represents a *wait*() call (which is invoked in thread2) and node 6 represents the corresponding *notify*() call (which is invoked in thread1). So, in Fig. 2, node 10 is *synchronization dependent* on node 6.

**D13.** *Communication Dependence.* In a Java program two types of *communication dependencies* may exist. We restrict communication dependency among threads belonging to the same component program to be only *S-Communication dependence* type. Whereas communication dependency among threads belonging to different component programs is termed as *M-Communication dependence* type. In *S-Communication dependence*, *shared memory* may be used to support communication among threads. In this type, two threads exchange data via *shared objects.* In *M-Communication dependence*, communication among threads occurs through *sockets.*

*S-Communication Dependence.* For two threads belonging to the same component program, a statement $y$ in one thread is *S-Communication dependent* on a statement $x$ in another thread, if the value of an object defined at $x$ is directly used at $y$ through inter thread communication.

Let $x$ be a *def(var)* node in a shared object present in a *component program* $P_1$ and let $y$ be the corresponding *use(var)* node in the same shared object. Then node $y$ is said to be *S-Communication dependent* on node $x$. For example in Fig. 2, node 9 represents a *use(flag)* node (which is used in thread2) and node 7 represents the corresponding *def(flag)* node (in thread1). So, in Fig. 2, node 9 is *S-Communication dependent* on node 7. Similarly, node 11 (in thread2) is *S-Communication dependent* on node 5 (in thread1). Note that both thread1 and thread2 belong to the same component program.

*M-Communication Dependence.* In a component program $P_1$, let $x$ be a node representing a statement which invokes a *getOutputStream*() method and $y$ be the corresponding node representing a statement which invokes a *getInputStream*() method in another component program $P_2$. Let both $P_1$ and $P_2$ use the same socket for communication. Then, the node $y$ is said to be *M-Communication dependent* on node $x$. For example in Fig. 1, node 18 represents a statements which invokes a *getInputStream*() method. Node 52 in Fig. 2 represents the statement which invokes the corresponding *getOutputStream*() method. So, node 18 of Fig. 1 is *M-Communication* dependent on node 52 of Fig. 2.

## 3 Intermediate Representation

In this section, we introduce an intermediate representation for distributed Java programs. We have named our intermediate representation *Distributed Program Dependence Graph* (DPDG). We use this representation to compute dynamic slices of distributed Java programs. We first discuss some issues that must be addressed to be able to accurately capture the dependencies existing in a distributed program, we then introduce our DPDG, and finally explain how it can be constructed.

The intermediate representation for a *concurrent* object-oriented program on a *single* machine can be constructed statically as in [18]. But, this intermediate representation can not be used to accurately model a *distributed* object-oriented program where *true* concurrency exists among the different threads running on different machines. For *distributed* object-oriented programs, we can have *communication dependency* among threads running on different machines. A *getInputStream*() call executed on one machine, might have a pairing *getOutputStream*() on some other remote machine. To represent this aspect, we introduce a logical(dummy) node in the DPDG. We call this logical node as a *C-node.* In the following, we define a *C-node.*

**D14.** *C-Node.* Let $G_{D_1}$ and $G_{D_2}$ be the DPDGs of two component programs $P_1$ and $P_2$ respectively. Let $x$ be a node in $G_{D_1}$ representing a statement invoking a *getOutputStream*() method. Let $y$ be a node in $G_{D_2}$ representing the statement invoking the corresponding *getInputStream*() 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 *getOutputStream*() with a *getInputStream*() call at node $y$. Node $y$ is *M-Communication dependent* on node $x$.

As an example consider node 18 in Fig. 3 and node 25 in Fig. 4 which represent statements invoking *getInputStream*() methods. At those nodes, the messages sent by the sending threads (e.g. from statement 52 in Fig. 2 and from statement 16 in Fig.1, respectively), are received. So, the algorithm associates *C-nodes* $C(18)$ and $C(25)$ at nodes 18 and 25 respectively. Node 18 is *M-communication dependent* on node $C(18)$ and node 25 is *M-communication dependent* on node $C(25)$ due to message passing.

The *C-nodes* maintain the logical connectivity among DPDGs representing different component programs. We therefore call them *logical nodes.* A *C-node* does not represent any specific statement in the source code of a component program. Rather, it encapsulates the triplet < *send_TID, send_node_number, dynamic_slice_at_send_node* > representing the pairing of the components in a distributed program. Here, *send_TID* represents the *id* of the thread 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 threads of different component programs. Since *C-nodes* are not mapped to any specific program statement, we call them *dummy nodes.*

In case of inter-thread communication through sockets, the triplet < *send_TID, send_node_number, dynamic_slice_at_send_node* > should be made available on the C-node *C(x)* corresponding to the *getInputStream*() node $x$ of the DPDG. For this, the thread executing a *getOutputStream*() call needs to perform the following. The thread passes the message to be sent to the slicer. The slicer piggybacks this triplet on the message. Whenever any thread executes a *getInputStream*() call, the slicer extracts the triplet from the message in the message queue and passes the actual message to the receiving thread. Thus the slicer updates the information on *C-nodes* and establishes the communication dependency.

It may be noted that the number of *C-nodes* in the DPDGs of a distributed Java program, equals the number of *getInputStream*() calls present in the program. In the DPDG, for a *getInputStream*() node $x$, the corresponding *C-node* is represented as *C(x).*

Using the discussed terminology and concepts, we can now define a *Distributed Program Dependence Graph* (DPDG). Let $P = (P_1, \ldots, P_n)$ be a distributed Java program, and $P_i$ be a component 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 *thread dependent* on $x$. Such an edge is called a *thread dependence edge.*

4. $y$ is *synchronization dependent* on $x$. Such an edge is called a *synchronization dependence edge.*

5. $y$ is *communication dependent* on $x$. Such an edge is called a *communication dependence edge.*

For all the nodes $x$, representing *getInputStream*() calls, in the component program $P_i$, a dummy node *C(x)* is created, and a corresponding dummy M-Communication edge (x, C(x)) is added.

A Distributed Program Dependence Graph (DPDG) captures the basic thread structure of a distributed Java program component along with it's run-time behavior. Thus a DPDG represents dynamic thread creation, synchronization of threads, and inter-thread communication using message passing. This graph contains the information available from other remote slicers by having additional logical nodes(*C-nodes*). A DPDG can contain nine different types of nodes. In the following, we list these types of nodes:

1. a *def (assignment)* node represents a statement defining a variable,

2. a *use* node represents a statement using a variable,

3. a *predicate* node represents a statement containing an *if* construct,

4. a *run* node represents a statement containing a *run*() statement,

5. a *notify* node represents a statement containing a *notify*() method call,

6. a *wait* node represents a statement containing a *wait*() method call,

7. a *getInputStream*() node represents a statement invoking a *getInputStream*() method,

8. *getOutputStream*() node represents a statement invoking a *getOutputStream*() method,

9. a *C-node* is a dummy node associated with the *getInputStream*() node, and represents its logical connection with the corresponding *getOutputStream*() node of a remote DPDG.

The DPDGs of the example programs given in Fig. 1 and 2 are shown in Fig. 3 and 4 respectively. In these figures circles represent program statements and ellipses represent the *C-nodes.* Edges represent the various dependencies existing among program statements. Since, the *S-communication* dependence and *M-communication* dependence are handled in a similar fashion in our DDS algorithm, so we have used the same notations (dashed-dot edge) to represent them in the DPDG.

It can be observed that control dependencies do not vary with the choice of input values and hence can be determined statically at compile time. We refer to control dependencies as *static dependencies.* The dependencies arising due to data definitions, statements appearing under the scope of selection and loop constructs, inter-thread synchronization and inter-thread communication are handled at run-time after execution of every statement. These dependencies are *dynamic dependencies* and have to be handled appropriately at run-time.

## 4 Distributed Dynamic Slicing (DDS) Algorithm

In this section, we first briefly explain our DDS algorithm. Subsequently, we illustrate the working of our algorithm through an example. Next, we investigate the space and time complexities of the DDS algorithm.

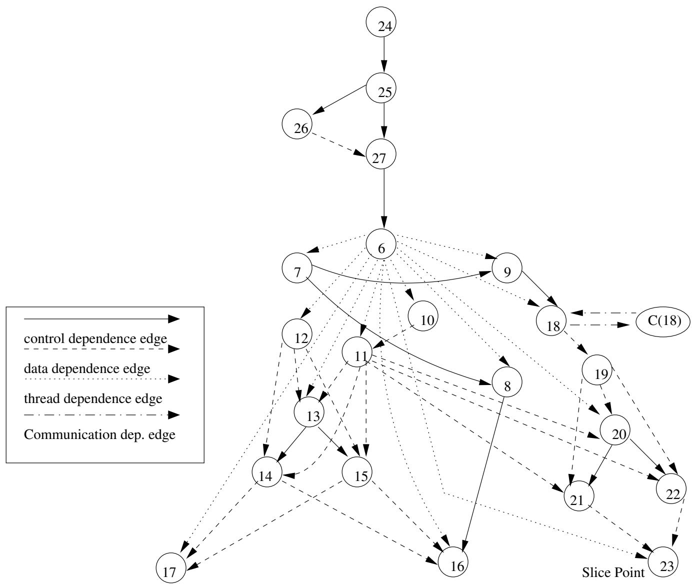


Figure 3: DPDG of the Example Client Program of Fig. 1

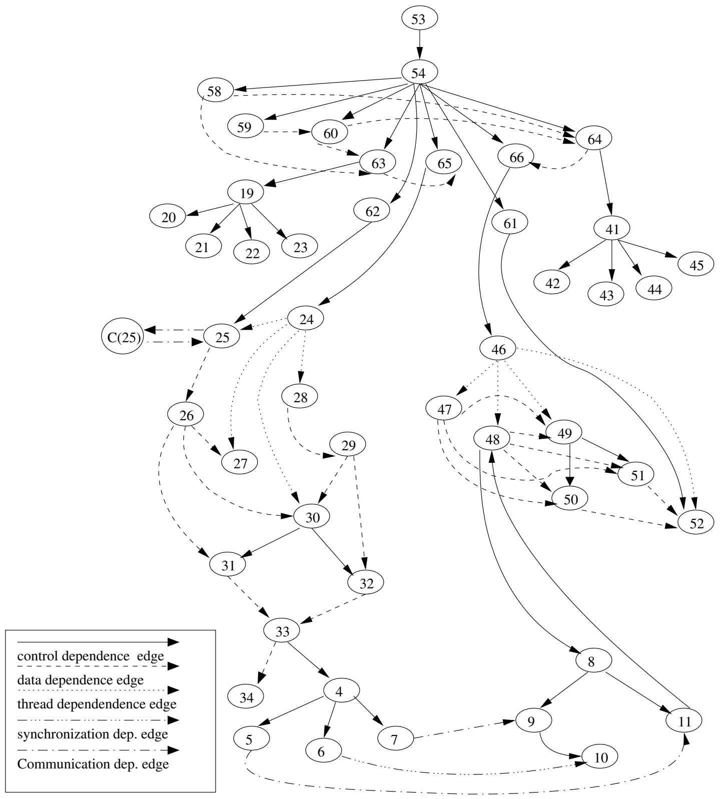


Figure 4: DPDG of the Example Server Program of Fig. 2

## 4.1 Overview of DDS Algorithm

We now provide a brief overview of our dynamic slicing algorithm. Before execution of a distributed Java program $P = (P_1, \ldots, P_n)$, the DCFG of each component program $P_i$ is constructed statically. Next, we statically construct the DPDG of each component program $P_i$ from the corresponding DCFG. During execution of a component program $P_i$, we mark an edge of the DPDG when its associated dependence exists, and unmark the edge when its associated dependence ceases to exist [19]. Since control dependencies do not change during run-time, we permanently mark the control dependence edges. We consider all the *data dependence* edges, *thread dependence* edges, *synchronization dependence edges* and *communication dependence* edges for marking and unmarking. We support communication to occur across different machines. The following activities are explicitly carried out in our DDS algorithm to capture this communication. Inter-machine communication is captured at run-time by adding *C-nodes* in the DPDG. The addition of *C-nodes* in the DPDG takes care of any communication dependency that might exist at run-time between communicating threads on different machines.

Whenever a statement invoking a *getInputStream*() method is executed during a run of the program, the slicer checks the message queue for availability of any message from any communicating thread. It then extracts the triplet $< send\_TID, send\_node\_number, dynamic\_slice\_at\_send\_node >$ that was piggybacked on the actual message. Then, the slicer updates the information on the *C-node* regarding the execution of the pairing *getOutputStream*() node in some thread on a remote or a local machine.

We compute the dynamic slice of a distributed Java program with respect to a *distributed slicing criterion.* We define a *distributed slicing criterion* for a component program $P_i$, as a triplet $< p, u, var >$, where $u$ is the statement of interest in thread $p$ and $var$ is a variable used or defined at statement $u$. During execution of the component program $P_i$, let *Dynamic_Slice (p, u, var)* with respect to the distributed slicing criterion $< p, u, var >$ denote the dynamic slice with respect to variable $var$ in the most recent execution of the statement $u$ in thread $p$. Let $(x_1, u), \ldots, (x_k, u)$ be all the *marked* incoming dependence edges of $u$ in the updated DPDG after an execution of the statement $u$. Then, we define the dynamic slice with respect to the present execution of the statement $u$, for the variable $var$ in thread $p$ as

Dynamic_Slice(p, u, var) $=\{(p, x_1), \ldots, (p, x_k)\} \cup Dynamic\_Slice(p, x_1, var) \cup \ldots \cup Dynamic\_Slice(p, x_k, var)$.

Let $\{$*var_1*, *var_2*, $\ldots$, *var_k*$\}$ be the set of all the variables used or defined at a statement $u$ in some thread $p$. Then, we define dynamic slice of the statement $u$ as

Dynamic_Slice(p, u) $= Dynamic\_Slice(p, u, var\_1) \cup Dynamic\_Slice(p, u, var\_2) \cup \ldots \cup Dynamic\_Slice(p, u, var\_k)$.

Our slicing algorithm operates in the following three main stages:

Stage 1: Construct the intermediate program representation graph statically
Stage 2: Manage the DPDG at run-time
Stage 3: Compute the required dynamic slice

In the first stage, the DCFG of each component program $P_i$ is constructed from a static analysis of the source code. Also, in this stage the static DPDG is constructed using the DCFG. The stage 2 of the algorithm handles run-time updations and is responsible for maintaining the DPDG as the program execution proceeds. The maintenance of the DPDG at run-time involves marking and unmarking the different dynamic dependencies as they arise and cease, and creating nodes for dynamic creation of threads, objects, etc. Stage 3 is responsible for computing the dynamic slices for a given slicing criterion using the DPDG. Once a slicing criterion is specified, our DDS algorithm immediately displays the dynamic slice with respect to the slicing criterion by looking up the corresponding *Dynamic_Slice* computed during run time.

To achieve fast response time, our DDS algorithm parallely runs 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. We now present our DDS algorithm for distributed Java programs in pseudo-code form.

**Algorithm:** Distributed Dynamic Slicing (DDS) algorithm.
**Input:** Slicing Criterion $< p, u, var >$
**Output** Dynamic_ Slice(p, u, var)

**Stage 1: Constructing Static Graphs**

```
1. DCFG Construction
   (a) Node Construction
       i. Create two special nodes start and stop
       ii. For each statement s of the sub-program $P_i$ do the following:
           A. create a node s
           B. Initialize the node with its type, list of variables used or defined, and its scope.
   (b) Add control flow edges
       for each node x do the following
           for each node y do the following
               Add control flow edge (y, x) if control may flow from node y to node x.
2. DPDG Construction
   (a) Add control dependence edges
       for each test(predicate) node u, do
           for each node x in the scope of u, do
               Add control dependence edge (u, x) and mark it.
   (b) Add data dependence edges
       for each node x, do
           for each variable var used at x, do
               for each reaching definition u of var, do
                   Add data dependence edge (u, x) and unmark it.
   (c) Add thread dependence edges
       for each run node u, do
```

Add *thread dependence edge (u, x)* for every node *x* that is thread dependent on *u* and unmark it.

(d) **A**dd synchronization dependence edges

```
for each wait node x in thread $t_1$, do
    for the corresponding notify node u in thread $t_2$, do
        Add synchronization dependence edge (u, x) and unmark it.
```

(e) **A**dd S-Communication dependence edges

```
for each use(var) node x in thread $t_1$, do
    for the corresponding def(var) node u in thread $t_2$, do
        Add S-Communication dependence edge (u, x) and unmark it.
```

(f) **A**dd M-Communication dependence edges

```
for each getInputStream() node u, do
    Add a C-node C(u)
    Add M-Communication dependence edge (u, C(u)) and unmark it.
```

**Stage 2: Managing the DPDG at run-time**

1. **I**nitialization: Do the following before execution of each of the component program $P_i$ at the corresponding local slicers:

   (a) Set *Dynamic_slice(NULL, u, var)* = $\phi$ for every variable *var* used or defined at every node *u* of the DPDG.

   (b) Set *recentDef(NULL, var)* = $\phi$ for every variable *var* in $P_i$.

   (c) Set message queue = $\phi$.

   (d) Set $< send\_TID, send\_node\_number, dynamic\_slice\_at\_send\_node >$ = NULL for every C-node *C(x)*.
   //end of initialization

2. **R**untime Updations: Run the component programs parallely. For a component program $P_i$, carry out the following at the corresponding local slicer after each statement *(p, u)* of $P_i$ is executed:

   (a) Unmark all incoming *marked* dependence edges to $(p, u)$ excluding the *control dependence edges*, if any, associated with the variable *var*, corresponding to the previous execution of the node $u$.

(b) **U**pdate data dependencies: For every variable *var* used at node *(p,u)*, mark the data dependence edge corresponding to the most recent definition *recentDef(p, var)* of the variable *var*.

(c) **U**pdate thread dependencies: For every node $u$, mark the thread dependence edge between the most recently executed *run node* and the node $(p, u)$.

(d) **U**pdate synchronization dependencies: If $u$ is a *wait* node, then mark the incoming synchronization dependence edge corresponding to the associated *notify* node.

(e) **U**pdate S-Communication dependencies: If u is a *use(var)* node in thread t1, then mark the incoming S-Communication dependence edge, if any, from the corresponding *def(var)* node in thread t2.

(f) **U**pdate M-Communication dependencies: If *(p,u)* is a *getInputStream*() node, then *mark* the incoming *M-Communication dependence edge*, if any, from the corresponding *C-* node *C(u)*.

(g) **U**pdate dynamic slice for different dependencies:

i. **H**andle data dependency:
Let $\{(d_1, u), \ldots, (d_j, u)\}$ be the set of *marked* incoming *data dependence* edges to *u* in thread $p$. Then,
Dynamic_Slice(p,u) $=\{(p, d_1), \ldots, (p, d_j)\} \cup Dynamic\_Slice(p, d_1) \cup \ldots \cup Dynamic\_Slice(p, d_j)$, where $d_1, d_2, \ldots, d_j$ are the initial vertices of the corresponding marked incoming edges of *u*.

ii. **H**andle control dependency:
Let *(c, u)* be the *marked control dependence edge*. Then,
Dynamic_Slice(p,u) $=Dynamic\_Slice(p, u) \cup \{(p, c)\} \cup Dynamic\_Slice(p, c)$.

iii. **H**andle thread dependency:
Let *t, u* be the *marked thread dependence edge.*Then,
Dynamic_Slice(p,u) $=Dynamic\_Slice(p, u) \cup \{(p, t)\} \cup Dynamic\_Slice(p, t)$.

iv. **H**andle synchronization dependency:
Let $u$ be a *notify* node in thread $p$ and $s$ be a *wait* node in thread $p1$. Let *s, u* be the *marked synchronization dependence edge.*Then,
Dynamic_Slice(p,u) $=Dynamic\_Slice(p, u) \cup \{p1, s\} \cup Dynamic\_Slice(p1, s)$.

v. **H**andle S-Communication dependency:
Let *u* be a *use(var)* node in thread $p$ and *(z, u)* be the *marked S-Communication dependence edge* from the corresponding *def(var)* node $z$ in thread $p_1$. Then,
Dynamic_Slice(p,u) $=Dynamic\_Slice(p, u) \cup \{(p1, z)\} \cup Dynamic\_Slice(p1, z)$.

vi. **H**andle M-Communication dependency:
Let *u* be a *getInputStream*() node and *(u, C(u))* be the *marked communication dependence edge* associated with the corresponding *C*-node *C(u)*. Then,
Dynamic_Slice(p,u) $=Dynamic\_Slice(p, u) \cup \{(p, C(u))\} \cup Dynamic\_Slice(p, C(u))$.

**Stage 3: Computing dynamic slice for a given slicing criterion**

1. **Dynamic Slice Computation:**

(a) For every variable *var* used at node *u* in thread *p* of the component program $P_i$, do
Let *(d,u)* be a *marked data dependence edge* corresponding to the most recent definition of the variable *var*,*(c,u)* be the *marked control dependence edge*, *(s,u)* be the marked synchronization dependence edge, *(t,u)* be the *marked thread dependence edge*,*(z,u)* be the marked S-Communication dependence edge, and *(C(u),u)* be the *marked M-Communication dependence edge*. Then,
Dynamic_Slice(p, u, var) = $\{(p,d),(p,c),(p1,s),(p,t),(p1,z),(p,C(u))\} \cup Dynamic\_Slice(p,d) \cup Dynamic\_Slice(p,c) \cup Dynamic\_Slice(p,s) \cup Dynamic\_Slice(p,t) \cup Dynamic\_Slice(p1,z) \cup Dynamic\_Slice(p,C(u))$
//p, p1 may be different threads.

(b) For a variable *var defined* at node *u*, do
Dynamic_Slice(p, u, var) = Dynamic_Slice(p, u).

2. **Slice Look Up:**

(a) If a slicing command $< p, u, var >$ is given for a component program $P_i$, carry out the following:

i. Look up *Dynamic_Slice(p, u, var)* for the content of the slice.

ii. Display the resulting slice.

(b) If the program has not terminated, go to step 2 of Stage 2.

**Working of the DDS Algorithm**

We illustrate the working of the algorithm with the help of an example. Consider the distributed Java program given in Fig. 1 and 2. The threads in the *client* and *server* programs are identified by unique thread-ids. Let the thread-id of the *clthd* in Fig. 1 be 1001, the thread-id of *thd1* in Fig. 2 be 2001 and the thread-id of *thd2* in Fig. 2 be 2002. The updated DPDGs are obtained after applying stage 2 of the DDS algorithm and are shown in Fig. 5 and Fig.6. Let us compute the dynamic slice with respect to variable $q$ at statement 23 of the thread *clthd* in the *client program* (Fig. 1). This gives us the slicing criterion $< 1001, 23, q >$ in the *client program.* With input data $s = 20$ in the client program in Fig. 1 and $b = 2$ in the server program in Fig. 2, we explain how our DDS algorithm computes the dynamic slice.

During the initialization step, our algorithm first unmarks all the edges of the DPDG and sets Dynamic_Slice($p$, $u$, $var$) = $\phi$ for every node $u$ of the DPDG. The algorithm has marked the *synchronization dependence edges* (6, 10) in Fig.6 as *synchronization dependency* exists between statements 6 and 10 due to *wait-notify* relationship. Statement 9 is *communication dependent* on statement 7 and statement 11 is *communication dependent* on statement 5 due to the shared objects *flag* and $s$ respectively. So, in Fig.6, the algorithm marks the *S-Communication dependence edges* (7, 9) and (5, 11). Node 9 is *communication dependent* on node $C(9)$ and node 63 is *communication dependent* on node $C(63)$ due to message passing. So, the algorithm marks the *M-Communication dependence edges* $(C(9), 9)$ as shown in Fig.5 and $(C(63), 63)$ in Fig.6. Similarly, the algorithm marks the control and data dependence edges when the respective dependencies arise. We have shown all the marked edges in Fig.5 and Fig. 6 in bold lines.

Now we explain how the DDS algorithm finds the backward dynamic slice with respect to the slicing criterion $< 1001, 23, q >$. According to our DDS algorithm, the dynamic slice at statement 23 of the *client* program, is given by the expression Dynamic_Slice(1001, 23, q) = {(1001, 21), (1001, 6)} ∪ Dynamic_Slice(1001, 21) ∪ Dynamic_Slice(1001, 21). By evaluating the expression in a recursive manner, we get the final dynamic slice at statement 23 of Fig.

1. The statements included in the dynamic slice are shown as shaded vertices in Fig. 5 and 6. The dynamic slice is also shown as the statements in rectangular boxes in Fig.7 and 8.

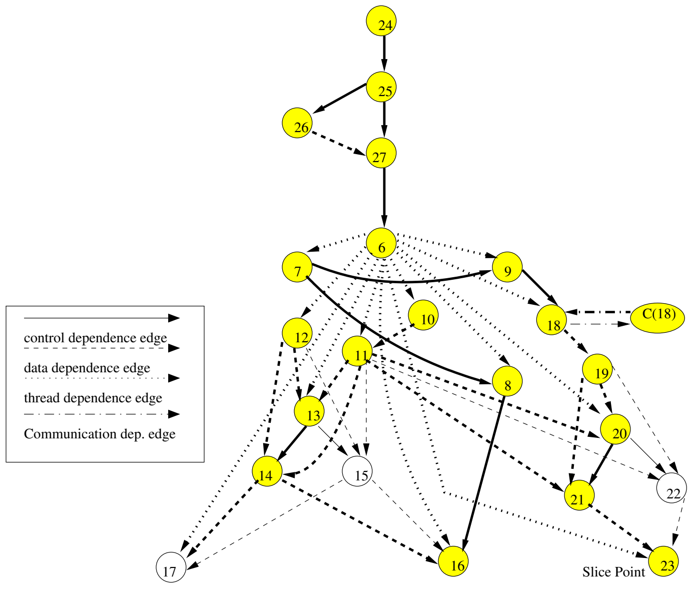


Figure 5: Updated DPDG of Client Program

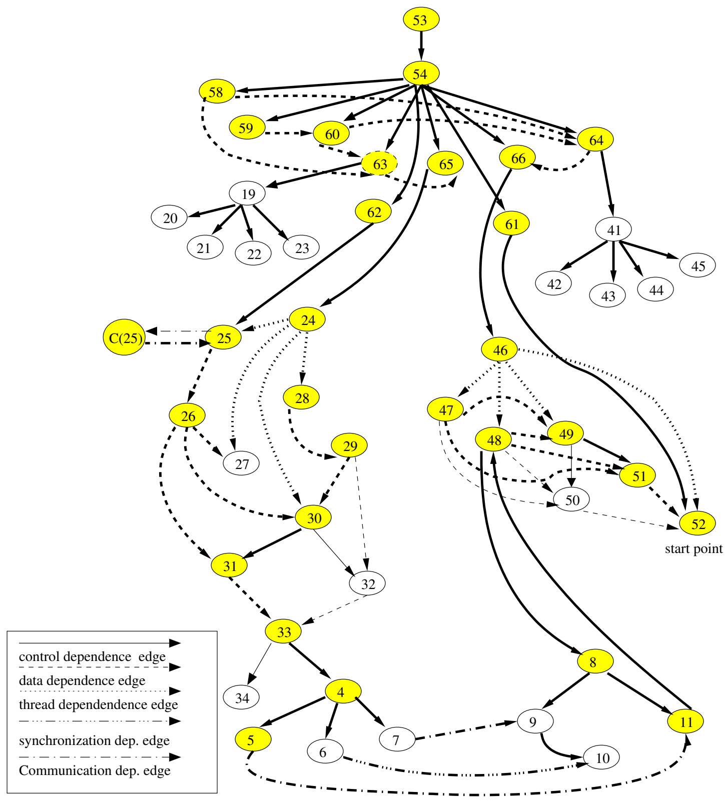
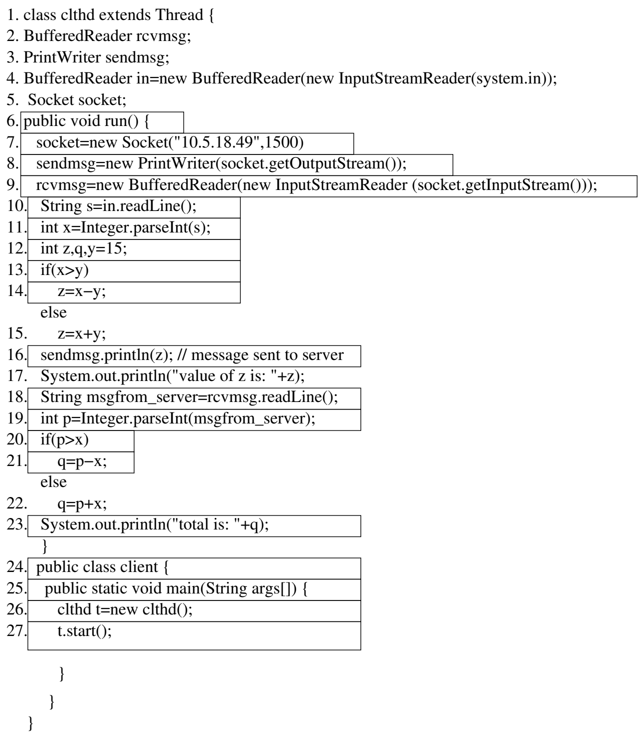


Figure 6: Updated DPDG of Server Program

## 4.2 Correctness of DDS Algorithm

In this section, we sketch the proof of *correctness* of our DDS algorithm.



```
1. class clthd extends Thread {
2. BufferedReader rcvmsg;
3. PrintWriter sendmsg;
4. BufferedReader in=new BufferedReader(new InputStreamReader(system.in));
5.  Socket socket;
6. public void run() {
7.   socket=new Socket("10.5.18.49",1500)
8.   sendmsg=new PrintWriter(socket.getOutputStream());
9.   rcvmsg=new BufferedReader(new InputStreamReader (socket.getInputStream()));
10.   String s=in.readLine();
11.   int x=Integer.parseInt(s);
12.   int z,q,y=15;
13.   if(x>y)
14.      z=x−y;
      else
15.      z=x+y;
16.   sendmsg.println(z); // message sent to server
17.   System.out.println("value of z is: "+z);
18.   String msgfrom_server=rcvmsg.readLine();
19.   int p=Integer.parseInt(msgfrom_server);
20.   if(p>x)
21.      q=p−x;
      else
22.      q=p+x;
23.   System.out.println("total is: "+q);
      }
24.  public class client {
25.    public static void main(String args[]) {
26.      clthd t=new clthd();
27.      t.start();

        }
      }
   }
```

Figure 7: Dynamic Slice for Slicing Criterion (1001, 23, q)

**Theorem 1** *DDS algorithm always finds a correct dynamic slice with respect to a given slicing criterion.*

**Proof.** The proof is given through mathematical induction. Let $P = (P_1, \ldots, P_n)$ be a distributed Java program for which a dynamic slice is to be computed using DDS algorithm. Let $P_i$ be a component program of $P$. For any given set of input values to $P_i$, the dynamic slice with respect to the first executed statement is certainly *correct*, according to the definition. From this, we can argue that, the dynamic slice with respect to the second executed statement is also *correct*. During execution of the component program $P_i$, assume that the algorithm has computed *correct* dynamic slices prior to the execution of a statement $u$. To complete the proof, we need only to show that the dynamic slice computed after execution of the statement $u$ is *correct*. Note that the statements that affect the execution of the statement $u$ must have been executed prior to this execution of the statement $u$. It is obvious that the dynamic slice *Dynamic_Slice(p, u, var)* contains all those statements which have affected the current value of the variable *var* used at $u$, since our DDS algorithm has marked all the incoming edges to $u$ only from those nodes on which node $u$ is dependent. The Steps 2(b), 2(c), 2(d), 2(e) and 2(f) of Stage 2 of the DDS algorithm ensure that the node $u$ is dependent (with respect to its present execution) on a node $v$ if and only if the edge *(u, v)* is marked in the DPDG of the component program $P_i$. If a node has no affect on the variable *var*, then it will not be included in the dynamic slice *Dynamic_Slice(p, u, var)*. So, *Dynamic_Slice(p, u, var)* is a *correct* dynamic slice. In other words, we can say that the dynamic slices computed prior to this execution of the statement $u$ are *correct*. Therefore, the Steps 2(g(i)), 2(g(ii)), 2(g(iii)), 2(g(iv)), 2(g(v)), 2(g(vi)) of Stage 2 and Steps 1(a) and 1(b) of Stage 3 of the DDS algorithm ensure that the dynamic slices computed after execution of the statement $u$ are *correct*. Further Step 2(b) of Stage 3 of the DDS algorithm guarantees that the algorithm stops when execution of the component program $P_i$ terminates. This establishes the correctness of the algorithm. □

## 4.3 Complexity Analysis

In this section, we analyze the space and time complexity of our DDS algorithm.

```
1.  class shar {
2.  int s;
3.  boolean flag=true;
4.  synchronized public void put(int c) {
5.  s=c;
6.  notify();
7.  flag=false;
     }
8.  synchronized public int get() {
9.  if(flag==true)
10.     wait();
11. return s;
     }
    }
12.  class thread1 extends Thread {
13.    BufferedReader rcvmsg;
14.    PrintWriter sendmsg;
15.    Socket serv_socket;
16.    int b,c;
17.    shar obj;
18.    BufferedReader o=new BufferedReader(
          new InputStreamReader(System.in));
19.    public thread1(Socket req,BufferedReader in,
              PrintWriter out,shar ob) {
20.    serv_socket=req;
21.    sendmsg=out;
22.    rcvmsg=in;
23.    obj=ob;
       }
24.    public void run() {
25.     String msg=rcvmsg.readLine();
26.     int a=Integer.parseInt(msg);
27.     System.out.println("received from client is: "+a);
28.     String mss=o.readLine();
29.     int b=Integer.parseInt(mss);
30.     if(a>b)
31.          c=a−b;
        else
32.          c=a+b;
33.     obj.put(c); //   message sent to thread2
34.     System.out.println("thd1: "+c);    }
      }
35.   class thread2 extends Thread  {
36.     BufferedReader rcvmsg;
37..    PrintWriter sendmsg;
38.    Socket serv_socket;
39.    shar obj;
40.    BufferedReader o=new BufferedReader(
          new InputStreamReader(System.in));
41.    public thread2(Socket req,BufferedReader in,
              PrintWriter out,shar ob) {
42.       serv_socket=req;
43.       sendmsg=out;
44.       rcvmsg=in;
45.       obj=ob;       }
46.    public void run() {
47.      int e,g,f=10;
48.      e=obj.get(); // message received from thread1
49.      if(e>f)
50.          g=e−f;
         else
51.          g=e+f;
52.     sendmsg.println(g); } }// message sent to client
53.   public class syn_server {
54.     public static void main(String args[]) {
55.       ServerSocket serv_socket;
56.       BufferedReader rcvmsg;
57.       PrintWriter sendmsg;
58.       shar obj=new shar();
59.       serv_socket=new ServerSocket(1500);
60.       Socket socket=serv_socket.accept();
61.       sendmsg=new PrintWriter(socket.
                 getOutputStream(),true);
62.       rcvmsg=new BufferedReader(new
             InputStreamReader(socket.getInputStream()));
63.      thread1 t1=new thread1(socket,input,output,obj);
64.      thread2 t2=new thread2(socket,input,output,obj);
65.      t1.start();
66.      t2.start();
       } }
```

Figure 8: Dynamic Slice for Slicing Criterion (1001, 23, q)

**Space complexity.** Let $P_i$ be a component program of the distributed Java program $P$. We assume that the number of threads in a component program $P_i$ is bounded by a small positive number. Let $P_i$ contains $n_i$ program statements. Let $k$ be the number of component programs in the distributed program $P$. The value of $k$ is usually a small finite number in a loosely coupled environment. Let $N$ be the total number of statements in all the component programs of $P$. So, $N = \sum_{i=1}^{k} n_i$. We used *C-nodes* for each *getInputStream*() node in the DPDG of every component program. The *C-nodes* are used to maintain logical connectivity among various component programs running on different machines. So, the slice at some arbitrary node of one DPDG may contain nodes of some other remote DPDGs. The number of *C-nodes* in the DPDGs of a distributed Java program, equals the number of *getInputStream*() calls present in the program. Since, the number of *getInputStream*() calls present in a component program is bounded, so number of *C-nodes* in all the DPDGs of the distributed program $P$ is bounded. It can be easily realized that the space requirement for the DPDG of a component program $P_i$ having $n_i$ statements is $O(n_i^2)$. We have assumed that the number of statements of a component program is bounded by the total number of statements in the whole distributed program . Also, if $N = \sum n_i$, then $(\sum n_i)^2 < N^2$. So, the space requirement for all the DPDGs of the distributed program $P$ having $N$ statements is $O(N^2)$. We need the following additional space at run-time for manipulating the DPDG:

1. To store *Dynamic_Slice(p, u, var)* for every statement $u$ of the component program $P_i$, at most $O(N)$ space is required, as the maximum size of the slice is equal to the size of the distributed program $P$. So, for $n_i$ statements in the component program $P_i$, at most$O(n_i N)$ space is required. Since $n_i$ is bounded by $N$, so in the worst case, the space requirement for *Dynamic_Slice(p, u, var)*, becomes $O(N^2)$, where $N$ is the total number of statements in $P$.

2. Let there are $v$ number of variables present in the component program $P_i$. To store *recentDef(thread, var)* for every variable *var* of $P_i$, at most $O(n_i)$ space is required. Assuming the number of variables present $(v)$ is less than the number of statements $(n_i)$, our DDS algorithm will require $O(n_i^2)$ space to store the *recentDef(thread, var)* of all the variables.

Since the space complexity of the DPDG and the run-time storage requirements is $O(N^2)$, the space complexity of our DDS algorithm is $O(N^2)$, $N$ being the total number of statements of the distributed program $P$.

**Time complexity.** To determine the time complexity of our DDS algorithm, we need to consider two factors making up the time required to compute a slice. The first one is the execution time requirement for the run-time manipulations of the DPDG. The second one is the time required to look up the data structure *Dynamic_Slice* for extracting the dynamic slice, once the slicing command is given. Let $S_i$ be the length of execution of the component program $P_i$. Let $S = \sum_{i=1}^{k} S_i$, where $k$ is the number of component programs in $P$. Let $N$ be the total number of statements in all the component programs in $P$, i.e . $N = \sum_{i=1}^{k} n_i$. Then, the time required for computing and updating information corresponding to an execution of a statement is $O(kN^2)$, since the updations occur simultaneously in the local nodes. The value of $k$ is usually a small finite number for a loosely coupled environment. So the time required for computing and updating information corresponding to an execution of a statement is $O(N^2)$. Hence, the run-time complexity of the DDS algorithm for computing the dynamic slice, *for the entire execution* of the distributed program $P$ is $O(N^2 S)$. We consider the complexity of computing the slice once a slicing criterion is defined, this excludes run-time computations required to maintain the DPDG.

The DPDG is annotated with the dynamic slices for the executed statements. So, the dynamic slices can be looked up in O(N) time, where $N$ is the total number of statements in the distributed program $P$.

## 5 Implementation

In this section, we present a brief description of a tool which we have developed to implement our dynamic slicing algorithm for distributed object-oriented programs. Our tool can compute the dynamic slice of a distributed object-oriented program with respect to a given slicing criterion. The current version handles only a subset of Java language constructs. Now, we are trying to extend our tool to handle the complete Java syntax. We have named our tool *Dynamic Slicer for Distributed Java programs* (DSDJ). To construct the intermediate graphs we have used the compiler witting tools JLEX and JYACC [20]. A distributed Java program is given as the input to the JLEX program. The JLEX program automatically generates the DPDGs for the component programs. The schematic design of our implementation is shown in Fig. 9.

A distributed Java program is read as input to our slicer. The *lexical analyzer* and *parser and semantic analyzer* components are combined and the joint component is termed as *program analysis component* [21]. The *lexical analyzer* part has been implemented using the standard lexical analysis tool *JLEX*. The *semantic analyzer* component has been implemented using *JYACC*, the standard tool for LALR(1) parser. During semantic analysis, the Java source code is analyzed token by token to gather the various program dependencies.

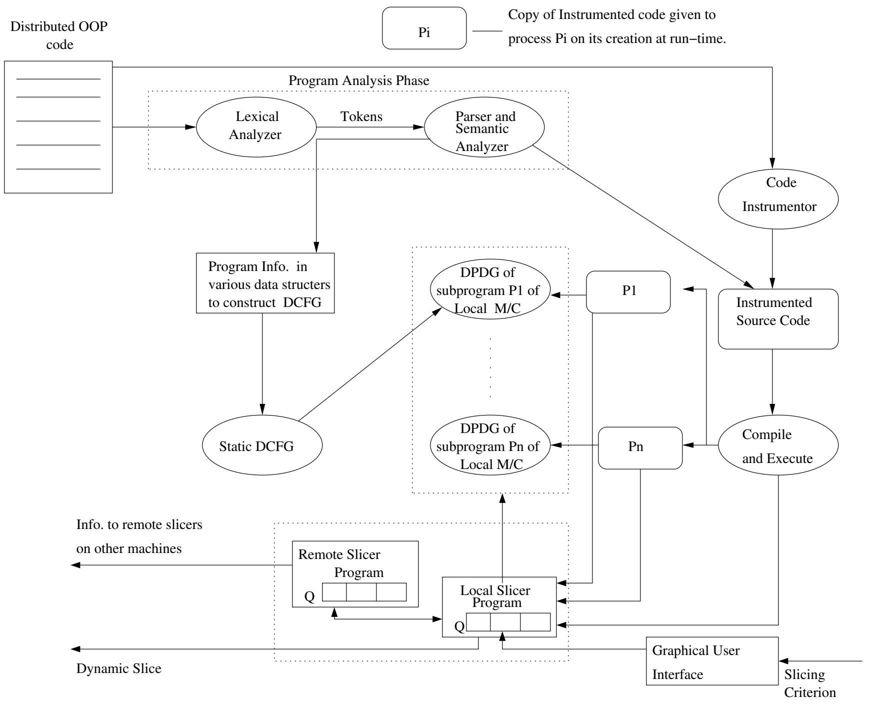

Distributed OOP code
Pi
Copy of Instrumented code given to process Pi on its creation at run−time.
Program Analysis Phase
Lexical Analyzer
Tokens
Parser and Semantic Analyzer
Code Instrumentor
Program Info. in various data structers to construct DCFG
DPDG of subprogram P1 of Local M/C
P1
Instrumented Source Code
Static DCFG
DPDG of subprogram Pn of Local M/C
Pn
Compile and Execute
Info. to remote slicers on other machines
Remote Slicer Program
Q
Local Slicer Program
Q
Dynamic Slice
Graphical User Interface
Slicing Criterion

Figure 9: Schematic Diagram of DSDJ

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 as mentioned in the stage 1 (DPDG Construction) of the DDS algorithm. The source program is then automatically instrumented, by adding calls to the slicer module after every statement in the source program.

We have used local slicers at every node of the distributed system for running the algorithm in a distributed manner. Each local slicer contributes to the dynamic slice by determining its local portion of the global slice in a fully distributed fashion. The local slicers update the DPDGs as and when the dependencies arise and cease at run-time, and compute the dynamic slice depending on the specified slicing criterion through the GUI. The local slicers also communicate among each other to help in the inter-component dependencies.

Table 1: Average run-time requirement and overhead cost of DDS algorithm

| Sl. No. | Total Prg. Size (# stmts) | # Component Programs | Normal Exec. Time (in mSec) | Avg. Run-Time Reqmt.(in mSec) | Over head cost(in mSec) |
|---|---|---|---|---|---|
| 1 | 250 | 2 | 94 | 142 | 48 |
| 2 | 355 | 2 | 117 | 185 | 68 |
| 3 | 462 | 2 | 141 | 237 | 96 |
| 4 | 558 | 3 | 165 | 294 | 129 |
| 5 | 670 | 3 | 190 | 355 | 165 |
| 6 | 782 | 3 | 215 | 416 | 201 |
| 7 | 894 | 3 | 247 | 482 | 235 |

Fig. 10 shows how the local slicers communicate with each other in a distributed environment.

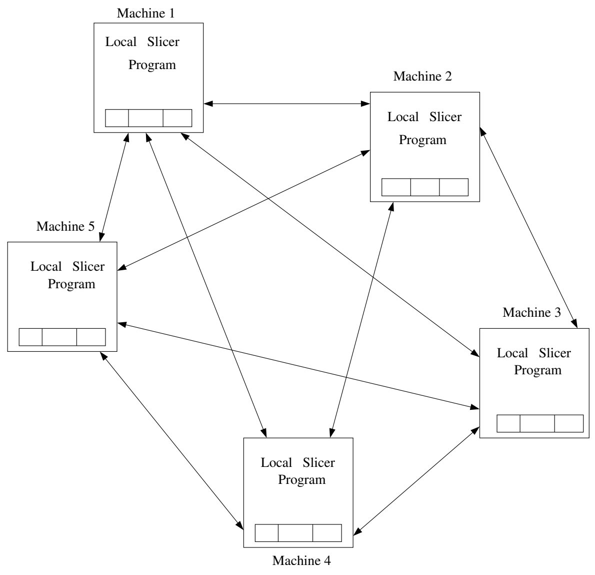


Figure 10: Communication among different local slicers

We have tested the working of our slicing tool, DSDJ, using a large number of distributed Java programs and for several slicing criteria. Our tool supports inter-thread synchronization and inter-thread communication using *sockets* and *shared memory*. We studied the run-time requirements of our DDS algorithm for several programs and for several runs. Table 1 summarizes the *average run-time requirements* and *over head costs* of the DDS algorithm. Since, we could not found any algorithm for dynamic slicing of distributed object-oriented programs, so we do not present any comparative results. We have only presented the results obtained from our experiments. The performance results of our implementation completely agree with the theoretical analysis. From the experimental results, it can be observed that the average run-time requirement increases slowly as the program size increases. So, the over head cost increases slowly as the program size increases. Table 2 summarizes the *memory requirements* of the DDS algorithm. It can be observed that the memory requirement increases slowly as the program size increases. This is due to the fact that the number of nodes and edges of the intermediate graph and the number of objects increase as the program size increases. It may be noted that, we have conducted the experiments for some typical example programs. So, the results such as average run-time requirements and memory requirements may vary from program to program. As the tool DSDJ does not need any trace files to store the execution history, it does not impose any restrictions on the size of the distributed programs. Also it saves the expensive file $I/O$ operations. Another advantage is that, the marking and

Table 2: Memory requirement of DDS algorithm

| Sl. No. | Total Prg. Size (# stmts) | # Component Programs | # Objects present | Memory requirement (in KB) |
|---|---|---|---|---|
| 1 | 250 | 2 | 55 | 502 |
| 2 | 355 | 2 | 76 | 714 |
| 3 | 462 | 2 | 88 | 930 |
| 4 | 558 | 3 | 105 | 1125 |
| 5 | 670 | 3 | 127 | 1350 |
| 6 | 782 | 3 | 145 | 1575 |
| 7 | 894 | 3 | 167 | 1798 |

unmarking technique used in our approach obviates the neccessity to create any new nodes in the different iterations of a loop. Thus, the run-time data structure remains bounded even in the presence of several loops.

## 6 Comparison With Related Work

Slicing of concurrent object-oriented programs has been investigated by many researchers [22–25]. Slicing of distributed procedural programs [26–30] has also drawn the attention of many researchers. To the best of our knowledge, no algorithm for *dynamic slicing* of distributed object-oriented programs has been reported in the literature. In the absence of any directly comparable work, we compare our algorithm with the existing *dynamic slicing* algorithms for distributed procedural programs.

Korel et al. [26] has proposed an extension of their dynamic slicing algorithm [2] to distributed programs with Ada type rendezvous communication. In their approach, each process generates a complete execution trace. The necessary dependence information to construct program slices is determined postmortem by analyzing the stored traces. Korel's slicing algorithm [26] operates on complete execution traces whose lengths may be unbounded. The computed slices are not independent programs and are executed using an explicit run-time scheduler which ensures the replay of the recorded communication events.

Duesterwald et al. [27] presented a *hybrid* parallel algorithm for computing dynamic slices of procedural distributed programs using a *distributed dependence graph.* Their algorithm combines both static and dynamic information to compute a slice. They used a Distributed Dependence Graph (DDG) to represent distributed program. A DDG contains a separate vertex for each statement and control predicate in the program. Control dependencies between statements are determined statically, prior to execution. Edges for data and communication dependencies are added to the graph at run-time. Slices are computed in the usual way by determining the set of DDG vertices from which the vertices specified in the criterion can be reached. Both the construction of the DDG and the computation of slices is performed in a distributed manner. Separate DDG construction procedure and slicing procedure are assigned to each process $p_i$ in the program. The processes in the program communicate when a *send* or a *receive* construct is encountered. Additionally, they proposed to transform non-deterministic communication constructs to deterministic ones to provide re-executable slices. Their approach requires the user to specify a slicing criterion in terms of a particular process and execution position. However, since a *single vertex* is used for all occurrences of a statement in the execution history, inaccurate slices may be computed in the presence of loops. They did not consider communication through *shared objects.* Also, their method can not be applied to programs where processes send messages *asynchronously* due to the assumption of synchronous message send.

Cheng [28] presented an alternate dependence graph-based algorithm for computing dynamic slices of procedural distributed and concurrent programs. He used *Program Depen-*

*dence Net* (PDN) as the intermediate representation. The PDN representation of a concurrent program is basically a generalization of the initial approach proposed by Agarwal and Horgan [31]. The PDN vertices corresponding to the executed statements are marked, and the static slicing algorithm is applied to the PDN sub-graph induced by the marked vertices. So, if a statement in a while loop is executed in some iteration, then the corresponding vertex is marked and included in the slice. But, if that statement is not executed in some other iteration, then that marked vertex is not removed from the slice. So, this approach yields inaccurate slices for programs having loops. Our algorithm *unmarks* the edges of the DPDG when the dependency does not exist. So, if a statement was executed in some iteration of a loop and for some other iteration it is not executed, then our algorithm successfully omits that statement from the slice.

Li et al. [30, 32] presented two novel *predicate-based dynamic slicing algorithms* for procedural distributed programs. Their algorithms are based on a *partially ordered multi-set* (POMSET) model. Unlike traditional slicing criteria that focus only on parts of the program that influence a variable of interest at a specific position in the program, a predicate focuses on those parts of the program that influence the predicate. The dynamic predicate slices capture some global requirements or suspected error properties of a distributed program and computes all statements that are relevant. Their algorithms handle distributed programs that communicate through message passing. They did not consider communication through *shared objects*. They have not considered the *object-orientation* aspects too.

Goel et al. [33] proposed compression schemes for representing execution profiles of shared memory parallel programs. Their representation captures control, data flow and synchronizations 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.

Gag et al. [34] 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 [35, 36] 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. They have not considered *object-orientation* aspects.

Existing techniques [26, 27] for debugging distributed programs include event-based debugging based on recorded event histories and execution replay. During instant replay, the original execution of a program (or an individual process) is reproduced based on the recorded order of received messages. All the existing methods [26, 27, 29, 33] use execution trace file whose size is proportional to the number of executed statements which itself can be unbounded in presence of loops, and upon this, they use graph reachability to compute dynamic slices which can take large amount of time. Our use of DPDG does not involve the use of trace files. As no trace files are used in our method, it also significantly improves the space as well as time complexities.

Our graph representation, is substantially different from all the existing methods [27–29] to take care of dynamically created threads and message passing using message queues. Our DPDG can handle thread creation, inter-thread synchronization and inter-thread communication. By using our method, messages can be sent *asynchronously* from one thread to another. In our approach, messages get stored in message queues and are later retrieved from the queue by the receiving thread. This is a more elaborate message passing mechanism compared to [26–30, 33]. Our dynamic slicing algorithm successfully handles the complications created by this message passing mechanism.

## 7 Conclusions

In this paper, we have proposed a novel technique for computing dynamic slices of distributed Java programs. We have introduced the notion of *distributed program dependence graph* (DPDG) as the intermediate program representation used by our slicing algorithm. We have named our algorithm *distributed dynamic slicing* (DDS) algorithm. It 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, our algorithm runs on several machines connected through a network in a distributed fashion. Our algorithm addresses the *concurrency* issues of Java programs while computing the dynamic slices. It also handles the *communication dependency* arising due to objects shared among threads on same machine and due to message passing among threads on different machines. Our algorithm does not require any trace file to store the execution history. Another important advantage of our algorithm is that when a slicing command is given, the dynamic slice is extracted immediately by looking up the data structure *Dynamic_Slice*, as it is already available during run-time. Although we have presented our dynamic slicing technique for Java programs, the technique can easily be adapted to other object-oriented languages such as C++.

## References

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

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

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

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

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

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

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

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

[9] 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.

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

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

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

[13] N. Wakinshaw, M. Roper, and M. Wood. The Java system dependence graph. In *Proceedings of IEEE International Workshop on Source Code Analysis and Manipulation*, pages 145–154, 2002.

[14] 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.

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

[16] P. Naughton and H. Schildt. *Java - The Complete Reference.* Mc GrawHill, 3rd Edition, 1998.

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

[18] 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*, pages 255–262, October 2004.

[19] 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.

[20] J. R. Levine, T. Mason, and D. Brown. *Lex and Yacc.* O'REILLY, 3rd Edition, 2002.

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

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

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

[24] 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.

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

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

[27] 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.

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

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

[30] 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.

[31] 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.

[32] J. Rilling, H. F. Li, and D. Goswami. Predicate based dynamic slicing of message passing programs. In *Proceedings of IEEE International Workshop on Source Code Analysis and Manipulation*, pages 133–144, 2002.

[33] 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.

[34] 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.

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

[36] 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.