Survey on Dynamic Slicing over Distributed Computing

Chandra Prakash Gupta
Department of Computer Science and Engineering, Centurion Institute of Technology, Jatni, Bhubaneswar, Orissa 752050, India

Irfanur Rahmana
Department of Computer Science and Engineering, Centurion Institute of Technology, Jatni, Bhubaneswar, Orissa 752050, India

Rakesh Kumar Raya
Department of Computer Science and Engineering, Centurion Institute of Technology, Jatni, Bhubaneswar, Orissa 752050, India

ABSTRACT
In this paper, discussed on dynamic program slicing algorithm which simplifies dependence and discussed the intermediate representation of a dynamic program slicing technique a Concurrent System Dependence Graph (CSDG) and intermediate representation of a distributed Java program in the form of a set of Distributed Program Dependence Graphs (DPDG).The algorithm can run parallel on a network of computers, with each node in the network contributing to the dynamic slice in a fully distributed fashion. The approaches discussed will not require any trace files to be maintained. Another advantage of this approach is that a slice is available even before a request for a slice is made. Analysis of the complexities of both the algorithm for dynamic program slicing technique and distributed dynamic slicing in Java

Keywords
Program slicing; Static slicing; Dynamic slicing; Debugging; Object-oriented programs; Threads; Multithreading; Java; Distributed programming; Synchronization

1. INTRODUCTION
Now a day's, size and complexity of object-oriented programming are increasing rapidly. This poses a formidable difficulty to the programmer to either understand the working of a program or debug an existing error. 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 are much harder compared to those of sequential programs. A typical nature of distributed programs, lack of global states, unsynchronized interaction among threads, multiple threads of control and a dynamically varying number of processes are some reason for this difficulty. An increasing amount of effort is being spent in debugging. In order to over-come with this situation, programmers need effective computer-supported techniques for decomposition and dependence analysis of programs. Program slicing is a technique for simplifying programs by focusing on selected aspects of semantics. The process of slicing deletes those parts of the program which can be determined to have no effect upon the semantics of interest. Program slicing has been found to be useful in a variety of applications such as debugging, program understanding, testing and maintenance. A program slice consists of the part of a program that affect the values computed at some point of interest referred to as a slicing criterion. Therefore a slicing criterion consists of a pair that is (line-number, variable). The part of a program which have a direct or indirect effect on the values computed at a slicing criterion C are called the program slice with respect to criterion C. the task of computing program slices is called program slicing.

Program slicing contains two types of slices based on the input to the program, first is static slice and second is dynamic slice. A static slice is valid for all possible execution of a program. A dynamic slice contains all statements that actually affect the value of a variable at a program point for a particular execution of the program rather than all statements that may have affected the value of a variable at a program point for any arbitrary execution of program. Dynamic slice is meaning full for only a particular execution. The advantage of dynamic slicing is the run-time handling of arrays and pointer variables. Therefore dynamic slices are usually smaller than static slices. In this paper we present a technique for dynamic slices for distributed programs. The slices are constructed for use in partial re-execution when debugging distributed programs.

2 COMPARISON
Developers have been dealing with several program slicing techniques to over-come with the problems like bugs, compatibility, portability and etc. Here will be comparison of two of the program slicing technique. First, computing dynamic slices of concurrent object-oriented programs *[1]* and secondly, distributed dynamic slicing of Java programs *[2]*.Dynamic slice contains all statements that actually affect the value of a variable at a program point for a particular execution of the program rather than all statements that may have affected the value of a variable at a program point for any arbitrary execution of the program. [8] Whereas distributed dynamic slice is a distributed subprogram that enables the re-execution of only the portion of the program that is of interest with respect to the computation of some selected values for specific input.[7]

We will be comparing the intermediate program representations: concurrent control flow graph(CCFG) and concurrent system dependence graph(CSDG), the marking-based dynamic slicing (MBDS) algorithm for concurrent object-oriented programs[1] with the intermediate program representations: distributed program dependence graph(DPDG), the distributed dynamic slicing(DDS) algorithm for distributed object-oriented programs.[2]

2.1 Basic concepts and definition
Before going through the graphs and algorithms we discuss briefly some of the features of Java. Then, we will discuss some of the basic concepts and definitions which will be used in the algorithm.

2.1.1. Concurrency in Java
Java is an Object Oriented Programming language. It supports various features, concurrency is one of them. Concurrent programming is supported by Java by the help of thread. A thread is identified similar to the sequential program in order that each and every thread is having beginning, running and end phase. Since a thread itself is not a program, so therefore it cannot be executed by its own. For this the Java support thread programming by providing Thread class library This Thread class library defines standard operations like *start(), stop(), suspend(),resume()and sleep(),* etc.[2]

2.1.2 Communication in Java

The communication among threads by both shared memory and message passing is provided by Java. When two or more threads share objects are known as condition variables. In Java programs, critical sections need to be marked with the keyword synchronized for synchronized access to shared data. To support synchronization among different threads, Java provides different methods like *wait(), notify()*, and *notifyall()*.Java Thread class provides few methods like *getOutputStream()* and *getInputStream()* for sending and receiving the messages between the threads.java provides sockets to support distributed programming. By the help of sockets, the client can identify the IP address and port number of the server to whom it wants to communicate.[2]

```
1  class Thread1 extends Thread{
2          BufferedReader receive_msg;
3          PrintWriter send_msg;
4          BufferedReader in=new BufferedReader(new
InputStreamReader(System.in));
5          Socket socket;
6          public void run(){
7          socket=new Socket("10.0.01.49",7514);
8        send_msg=new
PrintWriter(socket.getOutputStream());
9        receive_msg=new BufferedReader(new
InputStreamReader(socket.getInputStream()));
10         String str=in.readLine();
11         int a=Integer.parseInt(str);
12         int c,m,b=15;
13         if(a>y)
14                 c=a-b;
           else
15                 c=a+b;
16         send_msg.println(c);
17         System.out.println("value of c is:"+c);
18         String msg_from_server=
receive_msg.readLine();
19         int n=Integer.parseInt(msg_from_server);
20         if(n>a)
21                 m=n-a;
           else
22                 m=n+a;
23         System.out.println("total is:"+m);}
24    public class Client{
25         public static void main(String[] args){
26           Thread1 t1=new Thread1();
27            t1.start();}}}
```

Figure.1. Example Client Program

```
1 class Share{
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 receive_msg;
14     PrintWriter send_msg;
15    Socket server_socket;
16    int y,z;
17    Share object;
18    BufferedReader o=new BufferedReader(new
InputStreamReader(System.in));
19    public Thread1(Socket request,BufferedReader in,PrintWriter
out,Share ob){
20         server_socket=request;
21         send_msg=out;
22         receive_msg=in;
23         object=ob; }
24    public void run(){
25         String message=receive_msg.readLine();
26         int x=integer.parseInt(message);
27         System.out.println("received from client is:"+x);
28         String mss=o.readLine();
29         int y=Integer.parseInt(mss);
30         if(x>y)
31                z=a-b;
           else
32                z=x+y;
33         object.put(z);
34         System.out.println("thread1:"+z);}}
35  class Thread2 extends Thread{
36    BufferedReader receive_msg;
37    PrintWriter send_msg;
38    Socket server_socket;
39    Share object;
40    BufferedReader o=new BufferedReader(new
InputStreamReader(System.in));
41    public Thread2(Socket request,BufferedReader in,PrintWriter
out,Share ob){
```

```
42          server_socket=request;
43          send_msg=out;
44          receive_msg=in;
45          object=ob; }
46     public void run(){
47          int e,g,f=10;
48          e=obj.get();
49          if(e>f)
50                    g=e-f;
            else
51                    g=e+f;
52          send_msg.println(g);}}
53  public class SycServer{
54     public void main(String[] args){
55          ServerSocket server_socket;
56          BufferedReader receive_msg;
57          PrintWriter send_msg;
58          Share object=new Share();
59          server_socket=new ServerSocket(7514);
60          Socket socket=server_socket.accept();
61          send_msg=new PrintWriter(Socket.getOutputStream(),true);
62          receive_msg=new BufferedReader(new
InputStreamReader(socket.getInputStream()));
63          Thread1 t1=new Thread1(socket,input,output,object);
64          Thread1 t2=new Thread2(socket,input,output,object);
65          t1.start();
66          t2.start();}}
```

Figure.2. Example Server Program

2.1.3 Definitions

Definition 1- Precise Dynamic Slice

A dynamic slice is said to be precise if it includes only those sentences that essentially disturb the value of a variable at a point for the given execution.[1][2]

Definition 2- Correct Dynamic Slice

A dynamic slice is said to be correct if it contains all the statements of the program that affect the slicing criterion. A dynamic slice is said to be incorrect if it fails to include some statements of the program that affecttheslicingcriterion. [2]

Definition 3- def(var) and defSet(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 assignment for the variable var. The set *defSet(var)* denotes the set of all *def(var)* nodes. [1]

Definition 4- 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, if it uses the variable var. [1][2]

Definition 5- recentDef(thread, var)

Let *s* be a *def(var)* node of a component program P_i. Let pi 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 pi. [1][2]

Definition 6- 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 $n \varepsilon N$ represents a statement of P_i, and each edge e ε E represents potential control transfer among the nodes. [2]

Definition 7- Post Dominance*:*

Let *x* and *y* be two nodes in a (CCFG) G. Node *y* post dominates node *x* if every directed path from x to stop passes through *y*. [1][2]

Definition 8- 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* if there exists a directed path D from *x* to *y* such that:

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

```
class Thread1 extends Thread{
        private SyncObject s;
        private CompObject c;
        void Thread1(SyncObject s,CompObject
          x1,CompObjectx2,CompObject x3){
                this.s=s;
                this.x1=x1;
                this.x2=x2;
                this.x3=x3;}
1       public void run(){
2               x2.mul(x1,x2);
3               s.Snotify();
4               x1.mul(x1,x3);
5               s.Swait();
6               x3.mul(x2,x2);}}
class Thread2 extends Thread{
        private SyncObject s;
        private CompObject c;
        void Thread1(SyncObject s,CompObject
          x1,CompObject x2,CompObject x3){
                this.s=s;
                this.x1=x1;
                this.x2=x2;
                this.x3=x3;}
7       public void run(){
8               s.Swait();
9               x2.mul(x1,x1);
10              s.Snotify();
11              if(x1!=x2)
12                      x3.mul(x2,x1);
                else
13                      x2.mul(x1,x1);}}
14      class Sample{
```

```
15        public static void main(mString[] argm){
              CompObject x1,x2,x3;
              SyncObject s;
              s.reset();
16 x1=new CompObject(Integer.parseInt(argm[0]));
17    x2=new CompObject(Integer.parseInt(argm[1]));
18 x3=new CompObject(Integer.parseInt(argm[2]));
19        Thread1 t1=new Thread(s,x1,x2,x3);
20        Thread2 t2=new Thread(s,x1,x2,x3);
21        t1.start();
22        t2.start();}}
```

Figure.3. Example Program

Definition 9- 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*, if there exists a directed path D from *x* to *y* such that there is no intervening *def(var)* node in D. [1][2]

Definition 10- 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*, if there exists a directed path D from *x* to *y* such that none of the nodes in D is a run node. [2]

Definition 11- Synchronization Dependence

A statement *y* in a thread is synchronization dependent on a statement *x* in another thread, if execution of *y* is dependent on execution of *x* due to a synchronization operation. [1][2]

Definition 12- Communication Dependence

In a Java program two types of communication dependencies may exist.

i. S-Communication dependence
ii. M-Communication dependence

3.INTERMEDIATE PROGRAMREPRESENTATIONS

3.1. Distributed program dependence graph (DPDG)

The distributed program dependence graph (DPDG) $G_{Di}$of the component program P_i is a directed graph $(N_{Di} ; E_{di})$ where each node *n* (excepting the dummy nodes) represents a statement in P_i. For *x*, *y* element of N_{Di} , *(y, x)*element of E_{di} if any one of the following holds:

a. *y* is control dependent on *x*. Such an edge is called acontrol dependence edge.
b. *y* is data dependent on *x*. Such an edge is called a data dependence edge.
c. *y* is thread dependent on *x*. Such an edge is called a thread dependence edge.
d. *y* is synchronization dependent on *x*. Such an edge is called a synchronization dependence edge.
e. *y* is synchronization dependent on *x*. Such an edge is called a synchronization dependence edge.

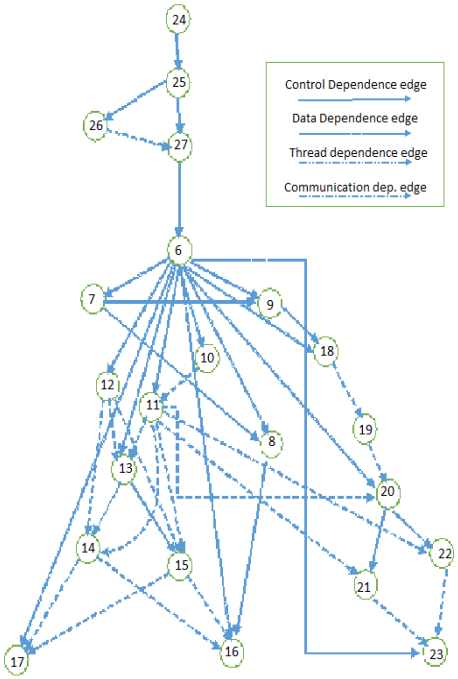

Figure. 4. The DPDG of the example Client program of Figure. 1.

A DPDG can contain nine different types of nodes. In the following, we list these types of nodes:

a. A *def(assignment)* node represents a statement defining a variable,
b. A *use* node represents a statement using a variable,
c. A *predicate* node represents a statement containing an *if()* construct,
d. A *run* node represents a statement containing a *run()* statement,
e. A notify node represents a statement containing a *notify()* method call,
f. A wait node represents a statement containing a *wait()* method call,
g. A *getInputStream()* node represents a statement invoking a *getInputStream()* method,
h. A *getOutputStream()* node represents a statement invoking a *getOutputStream()* method,
i. 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.

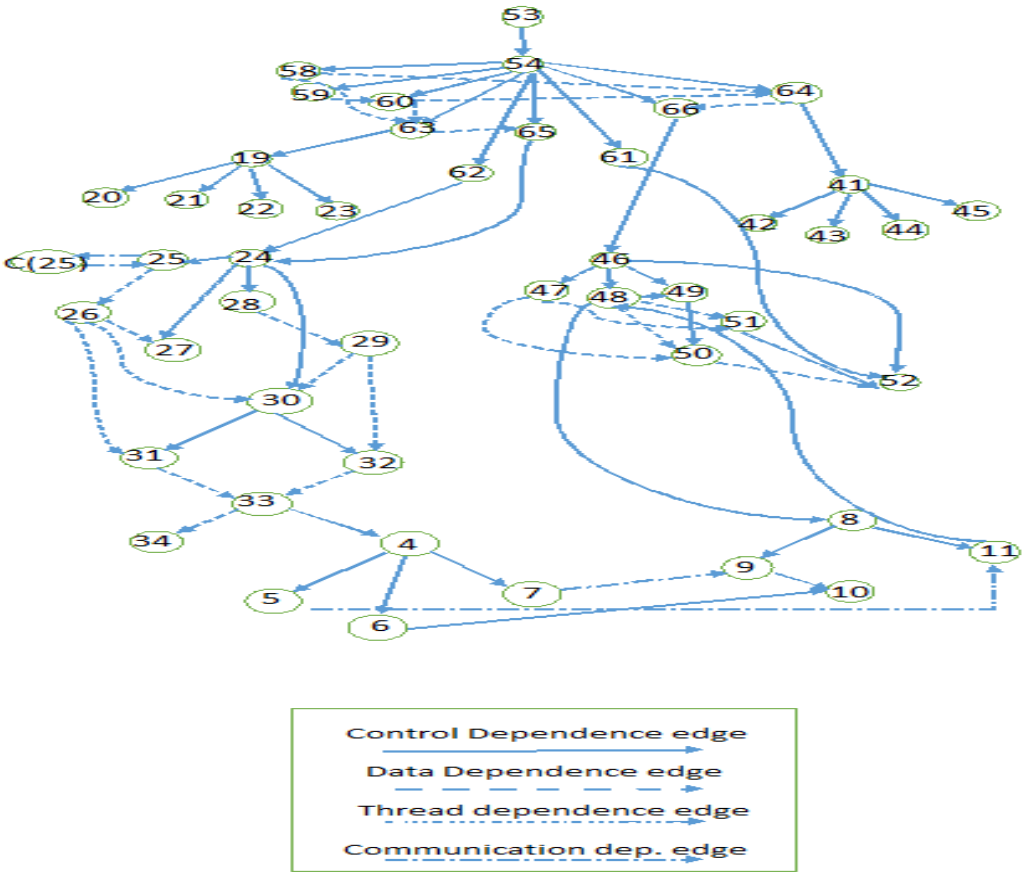

Figure. 5. The DPDG of the example Server program of Figure. 2.

3.2 Concurrent system dependence graph (CSDG)

When inter-thread synchronization and communication are present, some controls and data flows in the threads of a Java program become interdependent. To be able to capture this aspect, we use a dependence-based representation called the concurrent system dependence graph (CSDG) to represent the inter-thread synchronization and communication. CSDG is used to slice concurrent Java programs. First, we define a concurrent system dependence graph (CSDG) for a concurrent object-oriented program and then describe how to construct the CSDG.

A CSDG G_C of a concurrent object-oriented program P is a directed graph (N_C, E_C) where each node $n \varepsilon N_C$represents a statement in P. For $x, y \varepsilon N_C$, $(x, y) \varepsilon E_C$ if one of the following holds:

a. y is control-dependent on x. Such an edge is called a control dependence edge.
b. y is data-dependent on x. Such an edge is calledadata dependence edge.
c. y is synchronization-dependent on x. Such an edge is called a synchronization dependence edge.
d. y is communication-dependent on x. Such an edge is called a communication dependence edge.

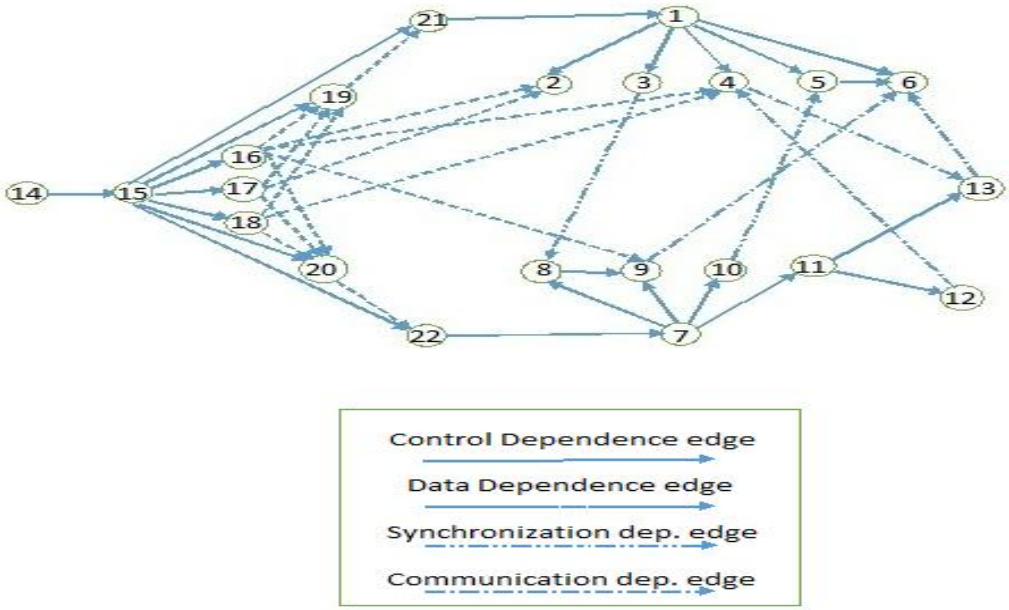

Figure. 7. The CSDG of the Sample program of Figure. 3.

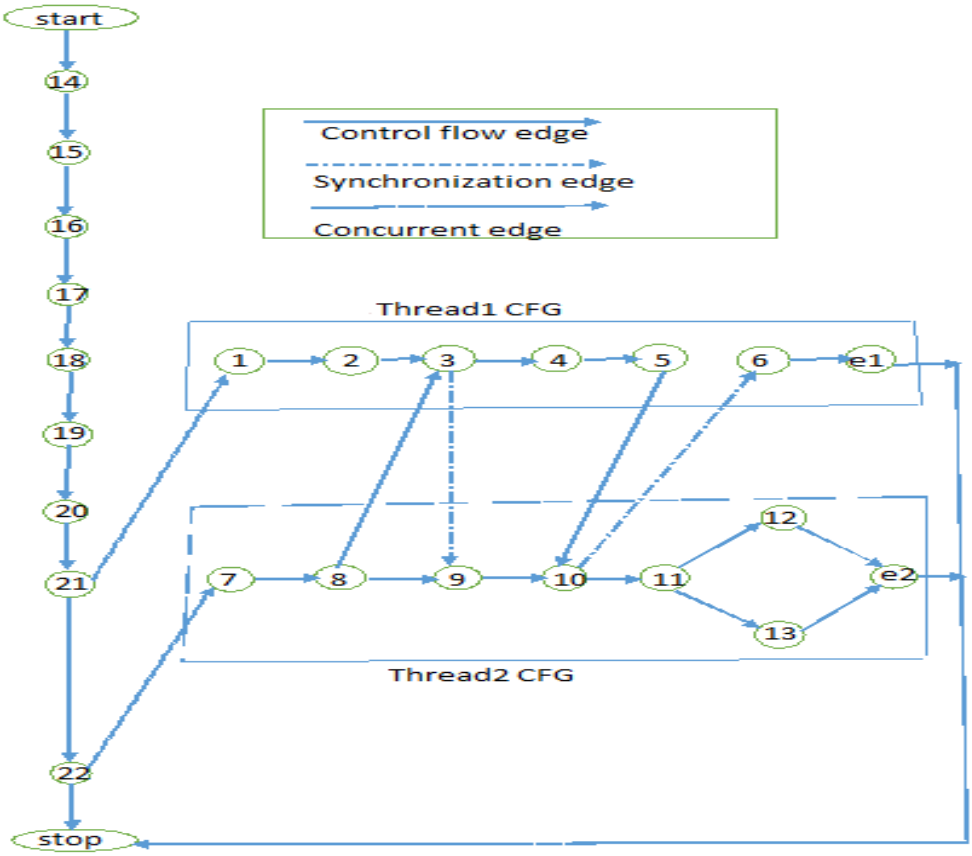

Figure. 6. The CFG of the Sample program of Figure. 3.

A CSDG can contain the following types of nodes:

a. Definition (assignment) node which represents a statement defining an object.
b. Use node which represents a statement using an object.
c. Predicate node which represents a statement containing an*if()* construct.
d. Notify node which represents a statement containing a *notify()* method call.
e. Wait node which represents a statement containing a *wait( )* method call.

4. ALGORITHM

4.1 Marking-based dynamic slicing (MBDS) algorithm

In computing dynamic slices of concurrent object-oriented program paper, a MBDS algorithm is proposed which we will be discussing in brief.

Before execution of a concurrent object-oriented program *P*, its CCFG and CSDG are constructed statically.

We permanently mark the control dependence edges, control dependence do not change during program execution. We consider all the data dependence edges, synchronization dependence edges and communication dependence edges for marking and unmarking during run-time.
The MBDS slicing algorithm operates in three main stages:

1. Statically constructing the intermediate program representation graph.

2. Managing the CSDG at run-time.

3. Computing the dynamic slice.

Algorithm: Marking Based Dynamic Slicing (MBDS) Algorithm.

Input: *Slicing Criterion<u,obj>*

Output: *Dynamic_Slice<u,obj>*

Stage 1: Constructing Static Graphs

1. CCFG Construction

(a) Node Construction

(i) Create two special nodes start and stop

(ii) For each statement *s* of the program *P* do the following:

(A) Create a node *s*

(B)Initializethenodewithits 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. CSDG Construction

(a) Add control dependence edges

for each *test(predicate)* node u do the following

for each node x in the scope of u do the following

Add control dependence *edge (u,x)* and mark it.

(b)Add data dependence edges
for each node *x*do the following
for each object *obj*used at *x* do the
following
for each reaching definition u of *obj*do thefollowing
Add data dependence *edge (u,x)* and unmark it.

(c) Add synchronization dependence edgesoreach wait node *x* in thread t_1 do the following

for the corresponding notify node *u* in thread

t_2 do the following

Add synchronization dependence *edge (u,x)* and unmark it.

(d) Add communication dependence edgesor each

use(obj) node *x* in thread *t1* do the following

for each *def(obj)* node *u* in thread t_2 do the following

Add communication dependence *edge (u,x)* and unmark it.

Stage 2: Managing the CSDG at run-time

1. Initialization. Do the following before execution of the program P

(a) Set *Dynamic_Slice(u,obj)* = $\phi$for every object *obj*used or defined at every node *u* of the CSDG.

(b) Set *recentdef(obj)* = *NULL* for every object *obj*of the program *P*.

2. Run-time update *s*. Run the program and carry out the following after each statement *u* of the program *P* is executed

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

(b) Update data dependencies. For every object *obj*used at node *u*, mark the incoming data dependence edge corresponding to the most recent definition *recentdef(obj)* of the object *obj*.

(c) Update synchronization dependencies. If *u* is a wait node, then mark the incoming synchronization dependence edge corresponding to the associated notify node.

(d) Update communication dependencies. If *u* is a *use(obj)* node, then mark the incoming communication dependence edge, if any, corresponding to the associated *def(obj)* node.

(e) Update dynamic slice for different dependencies:

(i) Handling data dependency

(ii) Handling control dependency

(iii) Handling synchronization dependence

(iv) Handling communication dependency

Stage 3: Computing dynamic slice for a given slicing criterion

1. For every object obj, used at node u, do the following Let *(d, u)* be a marked data dependence edge corresponding to the most recent definition of the object *obj*, *(z, u)* be the marked synchronization edge, *(t, u)* be the marked control dependence edge and *(c, u)* be the marked communication dependence edge.

Then,*Dynamic_Slice(u, obj) = {d, z, t, c} U dyn_slice(d) U dyn(z) U dyn_slice(t) dyn_slcie(c).*

2. For an object *obj*, defined at node *u*, do

Dynamic_Slice(u, obj) = dyn_slice(u).

4.2 Distributed dynamic slicing algorithm (DDS)

In distributed dynamic slicing of Java programs paper, a DDS algorithm is proposed which we will discuss briefly.

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.[1]

The 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.

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 edgesfor 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*, d

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) Add synchronization dependence edges

for each wait node *x* in thread *t1*, do

for the corresponding notify node u in thread *t2*,do

Add synchronization dependence *edge (u,x)* and unmark it.

(e) Add S-Communication dependence edges

for each *use(var)* node *x* in thread *t1*, do

for the corresponding *def(var)* node *u* in thread *t2*, do

Add S-Communication dependence *edge (u,x)* and unmark it.

(f) Add 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) Initialization: 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)* = ϕ for every variable varused or defined at every node u of the DPDG.

(b) Set *recentDef(NULL, var)* = ϕ for every variable varin *Pi*.

(c) Set message *queue* = ϕ.

(d)Set*(send_TID, send_node_number, dynamic_slice_at_send_node) = NULL* for every C-node *C(x).*

(2) Runtime Updates: 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) Update 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) Update thread dependencies: For every node *u*, mark the thread dependence edge between the most recently executed run node and the node *(p,u).*

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

(e) Update S-Communication dependencies: If *u* is a *use(var)* node in thread t_1, then mark the incoming S-Communication dependence edge, if any, from the corresponding *def(var)* node in thread t_2.

(f) Update 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) Update dynamic slice for different dependencies:

(i) Handle 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, $Dynamic\_Slice(p,u) = \{(p,d_1), \ldots, (p,d_j)\} \cup Dynamic\_Slice(p,d_1) \cup \_ \_ \_ \cup Dynamic\_Slice(p,d_j)$, *where* $d_1,d_2, \ldots ,d_j$are the initial vertices of the corresponding marked incoming edges of *u.*

(ii) Handle control dependency: Let *(c,u)* be the marked control dependence edge. Then, $Dynamic\_Slice(p,u) = Dynamic\_Slice(p,u) \cup \{(p, c)\} \cup Dynamic\_Slice(p, c)$.[1]

(iii) Handle 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) Handle synchronization dependency: Let *u* be a notify node in thread *p* and *s* be a wait node in thread p_1. Let *(s,u*) be the marked synchronization dependence edge. Then, $Dynamic\_Slice(p,u) = Dynamic\_Slice(p,u) \cup \{(p_1, s)\} \cup Dynamic\_Slice(p_1, s)$.

(v) Handle 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, $Dynamic\_Slice(p,u) = Dynamic\_Slice(p,u) \cup \{(p1, z)\} \cup Dynamic\_ Slice(p_1, z)$.

(vi) Handle 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, $Dynamic\_Slice(p,u) = Dynamic\_ Slice(p, u) \cup \{(p,C(u))\} \cup Dynamic\_Slice(p, C(u))$.

5. COMPLEXITY ANALYSIS

Here we will be discussing on the space and time complexities of the MBDS algorithm of computing dynamic slices of concurrent object-oriented programs paper and DDS algorithm of distributed dynamic slicing of Java programs

5.1. MBDS Algorithm

5.1.1. Space Complexity:

let assume, *P* be a concurrent object-oriented program with *n* statements. The CCFG and CSDG constructed in stage 1 are directed graphs on n nodes. [1]

Note, the graph of n number of nodes with optionally marked edges requires $O(n^2)$ *space.*

So, the space required for the CCFG and CSDG of *P* program with optionally marked edges is $O(n^2)$.

Some of the additional run-time spaces for computing the intermediate graph representation are required:

To store *Dynamic_Slice(u,obj)* for every node *u* of CSDG, at most $O(n^2)$ space is required.

To store *recentdef(obj)* for every object*obj* of *P*, at most *O(n)* space is required.

So the space complexity of the MBDS algorithm is calculated to be $O(n^2)$, where n is the number of executable statements in the program.

5.1.2. Time complexity

To calculate the time complexity of the MBDS algorithm, two factors has been considered the first one is the execution time required for the run-time manipulations of the CSDG and 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.

The complexity of set union is known to be *O(mn)*[10]. So, the worst case time complexity of MBDS algorithm for computing the dynamic slice, for the whole program is *O(mn).*

5.2. DDS Algorithm

5.2.1 Space Complexity

The space required for the DPDG of a component program Pi having ni statements is O(ni2). It is been assumed that the number of statements of a component program is bounded by the total number of statements in the whole distributed program. The space required for all the DPDG of the distributed program *P* having *N* statements is $O(N^2)$.

Some of the additional run-time spaces for manipulating the DPDG are required:

The space required for $Dynamic\_Slice(p, u, var)$ *is* $O(N^2)$, where *N* is the total number of statements in *P*.

It us been assumed that the number of variables present *(v)* is less than the number of statements (n_i). So, the DDS algorithm will require $O(n_i^2)$ space to store there*recentDef(thread, var)* of all the variables

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

5.2.2. Time Complexity:

To calculate the time complexity of the DDS algorithm, two factors has been considered marking up the time required to compute a slice.

1- The execution time required for the run-time manipulation of the DPDG.

2- The time required to look up the data structure *Dynamic_Slice* for extracting the dynamic slice, once the slicing command is given.

The run-time complexity of the DDS algorithm for computing the dynamic slice for entire execution of the distributed program *P* is $O(N^2S)$, where *S* is the length of execution of the component program *P*.

6. CONCLUSIONS

We discussed a technique for computing dynamic slices of distributed Java programs. We hadcompared the concept of Concurrent System Dependence Graph (CSDG) and distributed program dependence graph (DPDG) as the intermediate program representation used by theintroduced slicing algorithms. The algorithms are named as Marking-based

dynamic slicing (MBDS) and Distributed dynamic slicing (DDS) algorithm. These algorithms are based on marking and unmarking the edgesof the DPDG and CSDG as and when the dependencies arise and cease at run-time. To achieve fast response time, the DDS algorithm runs on several machines connected through a network in a distributed fashion. The DDS algorithm addresses the concurrencyissues of Java programs while computing the dynamic slices. It also handles the communication dependencyarising due to objects shared among threads on same machine and due to message passing among threads on different machines.The importantadvantage of the DDS 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 the dynamic slicing technique DDS is introduced for Java programs, the technique can be easily adapted to other object-oriented languages like C++.

7. REFERNCES

[1] Computing dynamic slices of concurrent object-oriented programs. Durga P. Mohapatra, RajibMalland Rajeev Kumar.

[2] Distributed dynamic slicing of Java programs. Durga P. Mohapatra, Rajjev Kumar, Rajib Mall, D. S. Kumar and MayankBhasin.

[3] E. Duesterwald, R. Gupta, M. Soffa, Volume 757, 1993, pp 497-511,Distributed slicing and partial re-execution for distributed programs.

[4] Mund, G., Mal , R., Sarkar, S., 2002. An efficient dynamic program slicing technique. Information and Software Technology 44, 123–132.

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

[6] Distributed slicing and partial re-execution for distributed programs. E. Duesterwald, R. Gupta,M. Soffa.

[7] Dynamic Program Slicing. Hiralal Agrawal, Joseph R. Horgan.

[8] D.P. Mohapatra et al. / The Journal of Systems and Software 79 (2006) 1661–1678 1669.

[9] G.B. Mund, R. Mall, S. Sarkar, Computation of intraprocedural dynamic program slices, Information and Software Technology 45 (2003) 499–512.