Abstract: A dynamic program slice is the part of a program that affects the computation of a variable of interest during program execution on a specific program input. Dynamic slices are usually smaller than static slices and are more useful in interactive applications such as program debugging and testing. The understanding and debugging of multithreaded and distributed programs are much harder compared to those of sequential programs. The nondeterministic nature of multithreaded programs, the lack of global states, unsynchronized interactions among processes, multiple threads of control and a dynamically varying number of processes are some of the reasons for this difficulty. Different types of dynamic program slices, together with algorithms to compute them have been proposed in the literature. Most of the existing algorithms for finding slices of distributed programs use trace files and are not efficient in terms of time and space complexity. Some existing algorithms use a dependency graph and traverse the graph when the slices are asked for, resulting in high response time. This paper proposes an efficient algorithm for distributed programs. It uses control dependence graph as an intermediate representation and generates the dynamic slices with fast response time.

Keywords: Program slicing; Dynamic slice; Program dependence graph; Debugging; Distributed programming; message passing.

I. INTRODUCTION

Program slicing is a well known decomposition technique for extracting the statements of a program related to a particular computation. A slice of a program P can be constructed with respect to a slicing criterion. A slicing criterion is a tuple <s, V> where s is a program point of interest and V is a subset of the program’s variables used or defined at s. The slice can be obtained by deleting the statements from the program P which have no effect on any of the variables in V as execution reaches statement s. There are two types of slices depending on the input to the program: static slice and dynamic slice. A static slice of a program P with respect to a slicing criterion <s, V> is the set of all the statements of program P that might affect the slicing criterion for every possible inputs to the program. In contrast, a dynamic slice contains only those statements of program P that actually affect the slicing criterion for a particular set of inputs to the program.

Nowadays most of the application 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. Development of real life distributed programs presents formidable challenge to the programmer so as to the debugging and testing process.

For usability of any dynamic slicer in an interactive distributed environment, the construction of slices should be made in a distributed manner. Each statement in a distributed system should contribute to the slice by determining its local portion of the global slice in a fully distributed fashion.

Weiser (1982) [9] introduced the concept of a static program slice and presented the first intraprocedural static slicing algorithm. His method used a Control Flow Graph (CFG) as the intermediate representation of the program, and was based on iteratively solving data-flow equations representing inter-statement influences. This algorithm did not handle programs having multiple procedures. Korel and Laski extended Weiser’s CFG based static slicing algorithm to compute dynamic slices. Their method computes dynamic slices by solving the associated dataflow equations. The method of Korel and Laski [8] needs O(N) space to store the execution history, and O(N^2) space to store the dynamic flow data, where N is the number of statements executed (length of execution) during the run of the program. Larson and Harrold were the first to consider object orientation aspects in their work. They introduced the class dependence graph which can represent a class hierarchy, data members, inheritance and polymorphism. They have constructed the system dependence graph (SDG) using the class dependence graphs to satisfactorily represent object oriented programs. Larson and Harrold have reported only a static slicing technique for sequential object-oriented programs, and did not address the concurrency and dynamic slicing aspects. Zhao, Song and Huynh, Wang et al. and Xu and Chen have addressed the issues of dynamic slicing of object-oriented programs, but they have not addressed the concurrency issues in object-oriented programs. Mohapatra et al. [4] have proposed an algorithm which uses a modified program dependence graph i.e. distributed program dependence graph (DPDG) for intermediate representation of programs. They have extended the basic techniques of the edge marking dynamic slicing algorithm of Mund et al. (2003) [6] to find out the distributed dynamic slices of a multithreaded java program thereby increasing the response time. Mund et al. (2006) [3] present an efficient interprocedural dynamic slicing algorithm for structured programs. They propose an intraprocedural dynamic slicing algorithm, and subsequently extend it to handle interprocedural calls. The interprocedural dynamic slicing algorithm uses a collection of control dependence graphs (one for each procedure) as the intermediate program representation, and computes precise dynamic slices. The proposed interprocedural dynamic slicing algorithm is more efficient than the existing dynamic slicing algorithms with faster response time. We use the basic concepts of Mund et al. [3] algorithm and propose an efficient algorithm for dynamic slicing of...
distributed programs computed in a distributed manner with fast response time.

The rest of the paper is organized as follows. In next section, we describe some basic definitions that are used by our algorithm. The next section describes our proposed algorithm. The working of the proposed algorithm is described in the subsequent section. The next section concludes the paper.

II. BASIC CONCEPTS AND DEFINITIONS

a) Control Dependence Graph
The control dependence graph (CDG) G of a program P is a graph G = (N,E), where each node n ∈ N represents a statement of the program P. For any pair of nodes x and y, (x,y) ∈ E if node x is control dependent on node y.

b) ActiveControlSlice(s)
Let s be a test node (predicate statement) in the CDG GP of a program P and UseVarSet(s) = {var1,...,vark}. Before execution of the program P, ActiveControlSlice(s) = Φ. After each execution of the node s in an actual run of the program, ActiveControlSlice(s) = {s} U ActiveDataSlice(var1) U …U ActiveDataSlice(vark) U ActiveControlSlice(t), where t is the most recently executed successor node of s in GP. If s is a loop control node, and the present execution of the node s corresponds to exit from the loop, then ActiveControlSlice(s) = Φ.

c) ActiveConcurrentSlice
ActiveConcurrentSlice is updated with every type of interaction that takes place between multiple machines in a distributed system e.g. Send and Receive, Lock and Unlock permissions. In case of communication between statements the slicer computes the ActiveConcurrentSlice for the sender and sends it to the slicer at the receiver end. This ActiveConcurrentSlice received by the receiver slicer helps in updating the it’s own ActiveConcurrentSlice.

<table>
<thead>
<tr>
<th>Machine A</th>
<th>Machine B</th>
</tr>
</thead>
<tbody>
<tr>
<td>main()</td>
<td>main()</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>10. send(p);</td>
<td>5. receive(d);</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 2.1: A distributed system with two machines

Let Uactive be the statement for interaction in a machine then, For Machine A
ActiveConcurrentSlice = Uactive U ActiveConcurrentSlice U ActiveControlSlice(t)
where t is most recently executed successor node of Uactive

For Machine B
ActiveConcurrentSlice = ActiveConcurrentSlice U ActiveConcurrentSlice (Received from A)
ActiveDataSlice(d) = ActiveDataSlice(p) U ActiveConcurrentSlice(B) U ActiveControlSlice(t)

d) ActiveDataSlice(var)
Let var be a variable in a program P. Before execution of the program P, ActiveDataSlice(var) = Φ. Let u be a Def(var) node, and UseVarSet(u) = {var1,...,vark}. Consider an actual run of the program with a given set of input values. After each execution of the node u in the actual run of the program, ActiveDataSlice(var) = {u} U ActiveDataSlice(var1) U ... U ActiveDataSlice(vark) U ActiveControlSlice(t), where t is the most recently executed successor node of u in GP.

e) DyanSlice(machineno, s, var)
Let s be a node of the CDG GP of a program P having identified by machineno, and var be a variable in the set DefVarSet(s) U UseVarSet(s). Before execution of the program P, DyanSlice(machineno, s, var) = Φ. Consider an actual run of the program with a set of given input values. After each execution of the node s in the actual run of the program, the dynamic slice DyanSlice(machineno, s, var) with respect to the slicing criterion <s, var> identified by machineno corresponding to the execution of s is updated as DyanSlice(machineno, s, var) = ActiveDataSlice(var) U ActiveControlSlice(t), where t is the most recently executed successor node of s in GP.

f) ActiveCallSlice
Let GP be the CDG of a multi-procedure program P. Before execution of the program, ActiveCallSlice = Φ. Consider an actual run of the program with a given set of input values. At an instance of the actual execution of the program, let uactive represent the active call node. Then ActiveCallSlice = {active} U ActiveCallSlice U ActiveControlSlice(t), where t is the most recently executed successor node of {active} in GP.

Figure 2.1: A distributed system with two machines

Let GP be the CDG of a multi-procedure program P. A stack is used called CallSliceStack to store a relevant sequence of ActiveCallSlices during an actual run of the program. During execution of the program the top element of the stack always represents the ActiveCallSlice. Before execution of each call node, the ActiveCallSlice corresponding to the execution of the call node is computed and pushed onto the stack CallSliceStack.

h) ActiveReturnSlice
Let GP be the CDG of a structured multi-procedure program P. Before each execution of the program P, ActiveReturnSlice = Φ. Let x be a RETURN node in GP, and UseVarSet(x) = {var1,...,vark}. Then, before each execution of the RETURN node x, ActiveReturnSlice = {x} U ActiveCallSlice U ActiveDataSlice(var1) U...U ActiveDataSlice(vark) U ActiveControlSlice(t), where t is the most recently executed successor node of x in GP.

i) Formal(x, var), Actual(x, var)
Let P1 be a procedure of a program P having multiple procedures, and x be a calling node to the procedure P1. Let f be a formal parameter of the procedure P1 and its corresponding actual parameter at the calling node x be a. Formal(x,a) = f
and Actual(x, f) = a. Note that Formal(x,a) = f iff Actual(x, f) = a.

III. ALGORITHM FOR DYNAMIC DISTRIBUTED SLICING

We now introduce our algorithm for slicing distributed programs. The Control Dependency Graph (CDG) for each machine is first constructed using the Control Flow Graph (CFG).

**Algorithm (For each machine)**

1. Construct the CDG GP of the program P statically only once.
2. Do the following before each execution of the program.
   - For each statement u of program P do the following
     - If u is a test (predicate) statement, then ActiveControlSlice(u) = Φ.
     - For each variable var 2 DefVarSet(u) U UseVarSet(u) do
       - DyanSlice(machineno, u, var) = Φ.
       - For each variable var of the program P do
         - ActiveDataSlice(var) = Φ.
         - ActiveCallSlice = Φ.
         - ActiveReturnSlice = Φ.
     - 3. Run the program P with the given set of input values, and repeat
        steps 4, 5, 6 and 7 until the program terminates.
     - 4. Do the following before execution of each call statement u.
       - Let u be a call statement to a procedure Q.
       - (a) Update CallSliceStack and ActiveCallSlice.
       - (b) For each actual parameter var in the procedure call Q do
         - ActiveDataSlice(Actual(u, var)) = ActiveDataSlice(var) U ActiveCallSlice.
     - 5. Do the following before execution of each RETURN statement u.
       - Update ActiveReturnSlice.
     - 6. Do the following before execution of each concurrent statement u of the program P
       - (a) If u is a send (var) node where data is a variable of program P then
         - Update ActiveConcurrentSlice and send it along with ActiveDataSlice(var) to the recipient machine in the distributed system.
       - (b) If u is a receive(var) node then
         - Update ActiveConcurrentSlice and ActiveDataSlice(var) to other machines in the distributed system.
       - (c) If u is a Notify / Unblock node then
         - Update ActiveConcurrentSlice and send it to other machines in the distributed system.
     - 7. Do the following after each statement u of the program P is executed.
       - (a) If u is a Def(var) statement and not a call statement then
         - Update ActiveDataSlice(var).
       - (b) If u is a call statement to a procedure Q then
         - For every formal reference parameter var in the procedure Q do
           - ActiveDataSlice(Actual(u, var)) = ActiveDataSlice(var).

IV. WORKING OF PROPOSED ALGORITHM

8. Exit when execution of the program P terminates.

**Algorithm**

<table>
<thead>
<tr>
<th>Machine A</th>
<th>Machine B</th>
<th>Machine C</th>
<th>Machine D</th>
</tr>
</thead>
<tbody>
<tr>
<td>main()</td>
<td>main()</td>
<td>main()</td>
<td>main()</td>
</tr>
<tr>
<td>{ }</td>
<td>{ }</td>
<td>{ }</td>
<td>{ }</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>10. send(p);</td>
<td>} ...</td>
<td>16. wait();</td>
<td>} ...</td>
</tr>
<tr>
<td>11. notifyAll();</td>
<td>} ...</td>
<td>17. receive(z);</td>
<td>} ...</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Figure 4.1: A distributed system with four machines</th>
</tr>
</thead>
<tbody>
<tr>
<td>a) Analysis of Algorithm</td>
</tr>
<tr>
<td>In this example, we are having four machines each running different parts of a distributed program. Each machine is executing its part of the program P instrumented with its local slicer. The slicer updates ActiveConcurrentSlice with every communication that takes place between other machines in the distributed system. In the above example, Machine A is broadcasting the message and notifying to other machines. After the notification, all other machines will receive the message and update their own slicer.</td>
</tr>
<tr>
<td>For Machine A</td>
</tr>
<tr>
<td>ActiveConcurrentSlice = Uactive U ActiveConcurrentSlice U ActiveControlSlice(t)</td>
</tr>
</tbody>
</table>
The ActiveConcurrentSlice and ActiveDataSlice(p) are sent to the Machine B Slicer.

For Machine B
ActiveConcurrentSlice = ActiveControlSlice U ActiveConcurrentSlice(Received from A)
ActiveDataSlice(x) = ActiveDataSlice(p) U ActiveConcurrentSlice U ActiveControlSlice(t)

For Machine C
ActiveConcurrentSlice(C) = ActiveControlSlice(C) U ActiveConcurrentSlice(Received from A)
ActiveDataSlice(y) = ActiveDataSlice(p) U ActiveConcurrentSlice(C) U ActiveControlSlice(t)

For Machine D
ActiveConcurrentSlice(D) = ActiveControlSlice(D) U ActiveConcurrentSlice(Received from A)
ActiveDataSlice(z) = ActiveDataSlice(p) U ActiveConcurrentSlice(D) U ActiveControlSlice(t)

b) Complexity of the algorithm
The space complexity of our algorithm is mainly due to the space requirement for storing the CDG G of the local part of the program P. If program P has n number of statements then maximum $O(n^2)$ space is required to store the graph G. It can be easily shown that the other data structures used by our algorithm requires maximum $O(n^2)$ space with disposal of the runtime data structures when not required. The time complexity of our algorithm remains $O(n)$.

c) Comparisons with related Algorithms
The advantage of this algorithm is that it does not use a trace file to store the execution history. It does not traverse a dependency graph. It uses some data structures to capture the runtime dependencies that exist in a distributed system. These run time data structures are updated with execution of each statement in the program. Hence the response time to extract a desired dynamic slice is substantially reduced.

V. CONCLUSION
In this paper we present an algorithm for slicing distributed programs. We use the basic concepts of the inter-procedural dynamic slicing algorithm and remodel it to extract slices of distributed programs with introduction of some additional data structures. We believe that to extract precise slices of distributed programs the slicing algorithm must also work in a distributed manner. The future scope of this paper lies in designing a testing tool for the distributed slicing algorithm.

VI. REFERENCES