Computer Standards & Interfaces 52 (2017) 71–84

Contents lists available at ScienceDirect

Computer Standards & Interfaces

journal homepage: www.elsevier.com/locate/csi

Dynamic slicing of distributed Aspect-Oriented Programs: A context-sensitive approach

Jagannath Singh*, P.M. Khilar, D.P. Mohapatra

Department of Computer Science and Engineering National Institute of Technology, Rourkela, India

ARTICLE INFO

Keywords:
Program slicing
Context-sensitive
Aspect-Oriented Programming
Distributed Aspect-Oriented Programs
Dynamic slicing

ABSTRACT

This paper presents a dynamic parallel context-sensitive slicing algorithm for distributed AOPs. The context-sensitivity makes the computed slice more precise and accurate. We introduce parallelism in our algorithm to make slice computation faster. We have developed a tool called D-AspectJ slicer to compute dynamic slices for distributed AOPs. The proposed slicing technique is compared with two other related existing techniques using ten case studies. The experimentation shows that our proposed slicing algorithm generates smaller slices in less time as compared to other two existing algorithms.

1. Introduction

The processing time taken for a single system to complete a task is larger than that of the processing time taken by more than one system performing the same task [1]. The most common example of a distributed system is the *client-server* architecture. The programs are written to perform some task in distributed manner are called distributed programs [2]. In this paper, we are concentrating on distributed Aspect-Oriented Programs (AOPs). The most expensive and tedious task in distributed programming is to handle the communication between the network nodes (i.e. computers) [3]. In the traditional object-oriented distributed programs, the distributed modules of the program communicate with each other by sending and receiving messages. But, the message can be passed to another computer at any point of time. Hence the codes for communication are scattered throughout the program. This type of code or section scattered throughout the program is called *crosscutting concern* [4]. AOP is a well-known programming paradigm, where all the crosscutting concerns are identified and bundled together in the form of an Aspect [4]. This Aspect module is responsible for the message passing between different modules running on different computers of the distributed system. This type of Aspect-Oriented Programs is called distributed AOPs. These distributed AOPs are very complex and difficult to understand. The rest of the activities in Software Development Life Cycle (SDLC) apart from coding, rely on the complexity of the program. If a program is difficult to understand, then it will be very much difficult to test it and debug the faults present in the program. Hence, there is a necessity of a program analysis tool that can reduce the complexity of a given distributed AOP.

Program slicing is a type of program analysis technique that studies the impact of a statement in a program on other statements of the same or different programs [5]. Given a slicing criterion, program slicing finds the other statements that may affect the present value of the variables in the slicing criterion [6]. Program slicing generates a sub-program for a given program, which is the collection of all the statements that are related to the given slicing criterion. Program slicing is also used in model-based testing to first produce slices of UML diagrams and then to generate test cases [7–9]. According to Weiser [5], a computed slice should be an executable slice, which means that the computed slice should produce the same output as the original program would have produced, if given the same input.

The major objectives of this paper are as follows:

- building a suitable intermediate graph for representation of the features of distributed AOP,
- designing a dynamic context-sensitive parallel algorithm for finding the precise slices of distributed AOPs, in less time.
- designing a prototype implementation of the proposed algorithm, and
- comparing the performance of our proposed approach with the similar existing approaches.

The rest of the paper is structured as follows: Section 2 presents the basic concepts of distributed AOP, dynamic program slicing, and some important definitions. In Section 3, we present the existing related work. Section 4 contains the details of a new intermediate representation, called Distributed Aspect Dependence Graph (DADG) for distributed AOPs. Our proposed parallel context-sensitive dynamic slicing

* Corresponding author.
E-mail addresses: jagannath.singh@gmail.com (J. Singh), pmkhilar@nitrkl.ac.in (P.M. Khilar), durga@nitrkl.ac.in (D.P. Mohapatra).

http://dx.doi.org/10.1016/j.csi.2017.01.007
Received 11 July 2016; Received in revised form 18 January 2017; Accepted 18 January 2017
Available online 26 January 2017

algorithm is also described in Section 4. We have implemented our slicing approach and developed a distributed AspectJ slicer. In Section 5, we present the architecture of the slicer, ten case studies, and comparison with some related work. Section 6 concludes the paper, with some insights into our future work.

2. Basic concepts

In this section, we present some basic concepts, which are most relevant and important for understanding our proposed technique. First, we describe the concept of distributed AOP. Then, we discuss different types of program slicing.

2.1. Distributed Aspect-Oriented Programs

Distributed computing is the concept of computing with independent processors connected through communication links. After establishing the distributed architecture, the next step is to develop the programs that can run in a distributed environment [2]. These programs consist of several concurrently executable parts allocated to different processors for simultaneous execution. The distributed systems rely on message passing for communication between all the computers [10]. This message passing makes the distributed programs more complex. Here, the advantages of *aspects* can be availed to reduce the complexity of the distributed programs. The code that handles the message passing between two computers can be identified and kept in a separate module as an Aspect [11]. This type of programs are called distributed Aspect-Oriented Programs.

Among all the known architectures of distributed computing, the *client-server* architecture is the simplest and most commonly used architecture [1,3]. In Fig. 1, we have presented an example distributed program for calculation of factorial of a given number. In Fig. 1, *MyClient. java* is a client program, which first establishes the connection with a server program by calling *Socket*() method. Then it asks the user to enter a number through a keyboard. Using *getOutputStream*() method the received number is then sent to the server. At the end, the client program receives the result from the server program and prints the result.

In Fig. 2, *MyServer. java* is a server program, which is responsible for calculating the factorial of a given number. When it starts execution, it waits for a client program to send some data. Once it receives a number from the client program through *getInputStream*() method, it then calculates the factorial of the input number. Then, it sends back the computed factorial of the number to the client that had sent the input number.

Now, we can observe that the server program is always running once it starts execution. If the server is ready then only it can receive any message from clients. But, from client side, it is very difficult to know whether the server is ready or not. Similarly, when the work of the server is over, before closing the server program, all clients must be notified that the server is going to be closed so that no client should send new data. To incorporate these modifications, again we have to change the existing code. Another alternative way is to design a separate program that can handle all theses additional requirements without changing the existing programs. We have created an aspect program using AspectJ [12] programming language to achieve this. This AspectJ program is shown in Fig. 3. We have developed an aspect called *MyServerAspect. aj* as shown in Fig. 3, which has one *pointcut* and two *advices*. The pointcut *PC*() captures the execution of *run*() method in *MyServer. java* program shown in Fig. 2. The *before*() advice executes before the *run*() method is executed and the *after*() advice executes after the *run*() method is executed. As a result, we can

```
//MyClient.java
        import java.net.*;
        import java.io.*;
        public class MyClient
                {
                public static void main(String [] args) {
                BufferedReader input=new BufferedReader(new
                        InputStreamReader(System.in));
                try     {
                Socket client = new Socket("localhost", 9093);
                OutputStream outToServer = client.getOutputStream();
                DataOutputStream out = new DataOutputStream(outToServer);
                System.out.println("Enter␣a␣number");
                String N=input.readLine();
                out.writeUTF(N);
                InputStream inFromServer = client.getInputStream();
                DataInputStream in =
                        new DataInputStream(inFromServer);
                System.out.println("Factorial=␣" + in.readUTF());
                        client.close();
                      }catch(IOException e)
                        {
                    e.printStackTrace();}}}
```

Fig. 1. An example client program for calculation of factorial of a number.