AUTOMATING PLACEMENT OF INSTRUMENTATION IN APPLICATIONS

Seema Kaushal
Motorola

Hanan Lutfiyya
Department of Computer Science
The University of Western Ontario
London, Canada
hanan@csd.uwo.ca

Abstract:

In this paper, we present an architecture that provides the functionality to place customized and automated instrumentation. We determine the components that are needed for this purpose, services offered by each of these components and the algorithms showing the steps taken. Using this architecture, the time and effort needed to develop instrumentation toolsets can be reduced. Consequently, the time and effort needed to place instrumentation in distributed applications, to make them manageable, can be greatly reduced. We also describe the current state of the prototype, our conclusions and future directions.

Keywords: Instrumentation, Management, Distributed Applications, TXL

1. Introduction

Effective management of distributed applications requires the ability to monitor application-specific attributes e.g., the amount of time a specific remote procedure call took. This requires application instrumentation; that is, code inserted into the application at strategic locations so that the application process can maintain monitored information, respond to management requests and generate event reports based on the evaluation of a condition on the state of the monitored information.

The advantage of manual instrumentation (i.e., adding instrumentation by hand) is that applications can be instrumented to meet their specific needs i.e., the instrumentation is customized. A disadvantage of manual instrumentation is the extra effort, resources and time is required by developers. It is this additional developer time that is often cited as a criticizm of instrumentation [1].

With automation, instrumentation can be placed automatically This would, not only save time in the development process, but also minimize the potential for errors.

This paper discusses the components of a toolset needed to provide the functionality to place customized and automated instrumentation. The paper is organized as follows: Section 2 describes an instrumentation architecture. In Section 3, the requirements and a description of the components and the services offered by each of the components is described. Section 4 describes the prototype. Sections 5 and 6 describe related work and conclusions.

2. Instrumentation Architecture

The purpose of this section is to briefly discuss the instrumentation architecture described in [6].

An application attribute is associated with a sensor. Management requirements often place contraints on the values that an application attribute may take e.g., the time it takes to complete a remote procedure call should be one second. Sensors have variables representing information that includes threshold values and comparison operators that are used to compare monitored attribute values with the threshold values. The sensor's methods (*probes*) are used to initialize sensors with threshold values and collect values of attributes. The coordinator is the interface between the sensors and the management system. An actuator is similar to a sensor except it encapsulates functions that can exert control over the intrumented process to changes it behaviour.

As an example, let us assume that we have a communication statistics sensor, which is responsible for computing the communication statistics of a remote procedure call (RPC) to a server that we will call echo_server(...). This server is to be passed a string. This sensor, denoted by rpcSensor, may have the following methods: Process_rpcRequestBegin(...), which will record the RPC's start time and Process_rpcRequestEnd(...), which will record the RPC's end time. By inserting these probes before and after a RPC, the communication statistics sensor is able to compute the time taken by a particular RPC to complete and if a threshold is exceeded it passes a notification to the Distributed Application Management System through the management coordinator.

3. Toolset Architecture

Placing instrumentation code is the process of inserting probes at strategic points (Probe Points) within the application's source code. A probe point can be described using a *pattern*. A *pattern* is a string that describes source code that should be searched for in order to insert instrumentation code. An example of a pattern is fopen($1,$2). $1, $2 are placeholders representing strings for the file name and the mode used to open the file. Another example of a pattern is echo_server($1) where $1 is a placeholder for a string.

The overall approach to automatically place instrumentation code is to parse the source code to create a parse tree, search for patterns in the parse tree, and inserting the instrumentation code at the points where the pattern was found. This results in modifying the original parse tree. The leaves of the modified parse tree are, then, written back to the source code form.

Several repositories are needed to support this approach. Information about patterns is stored in a Pattern Repository. Part of the information associated with a pattern record is the type of source code construct that the pattern is associated with e.g., IPC, OS, middleware (e.g., DCE), etc. This functionality allows for the categorization of patterns (e.g., IPC, OS). An example of the usefulness of this is the following: The user (through the GUI) can specify classes of patterns that are to be used in determining probe points. This way the user does not have to specify each individual pattern; rather they specify the class that the pattern belongs to. The user is also able to add patterns to the repository.

The Pattern Recognizer traverses the source code's parse tree to find a pattern. For each match found, the source code strings are passed to the Probe Writer component, which is is used for writing the corresponding probes i.e., the actual probe strings. This processing needs the instrumentation repository that relates patterns and sensors/actuators as well as a string with variable placeholders (*probe patterns*) that represents a probe for each probe of the sensor/actuator. The Probe Writer analyses the source code to get all the variables that are needed to replace the variable placeholders in the probe pattern. The result is an instantiated probe pattern which is the actual probe to be inserted in the source code. This is put into the Probes Location repository. The following example illustrates this need: A sensor is used to count the number of times each file is accessed. The name of the file needs to be known. This name is passed to the sensor using a probe that is placed just before the fopen(...) call or fread(...) call etc; The general form of the probe code may be (...).CountFileAccesses($1) where $1 is a string that represents any file name. Thus, the actual probe code will differ for different fread(...) calls. After the tree has been traversed, the Probe Inserter component retrieves the probes from the Probe Location component to insert the probes in the parse tree of the source code at predetermined locations. The source code is then written back to the appropriate file from its corresponding transformed parse tree.

An editor is provided that allows the user to selectively choose points to add instrumentation. It provides the user with basic editing facilities such as *Open*, *Close* or *Save* a file and *Cut* or *Paste* text. It displays the list of sensors and actuators so that any desired sensor/actuator can be selected and its probes can be added at the desired location. This makes use of the Instrumentation Repository component.

The user interface allows the user to add patterns, sensors, associations between patterns and sensors, and update the status if a pattern (i.e., is a pattern to be searched for automatically).

4. Prototype and Initial Evaluation

As a proof of concept, we developed a prototype tool based on the architecture described in the previous section. The developed prototype can automatically instrument DCE and socket *C/C++* applications and also provides the flexibility of adding customized instrumentation. The User Interface is currently implemented as one process in *Java 1.1*. The repositories are implemented as a set of flat files. Access to the repositories is through UNIX shell scripts. The processing components were implemented through the use of UNIX shell scripts. Parsing and pattern recognition services are implemented using *TXL* [3]. The Probe Writer and the Probe Inserter make use of the *Sed* and *Awk* facilities in Unix.

The developed toolset was used by several members of our research group to instrument socket and DCE applications including an MPEG player consisting of 5000 lines of code. The tool was found to be fairly easy to use and intuitive.

5. Related Work

Tools examined that automate the process of placing instrumentation to a varying extent included Pablo [2], AIMS [8], Paradyn [5]. Unlike our work, none of these tools provide the developer with the easy flexibility in adding their own set of source code constructs for automated instrumentation. The work closest to ours can be found

in [4]. There were several differences in our approaches including their inability to add patterns and thus change the probe points where instrumentation should be placed.

6. Conclusions

This paper presented a toolset that can automatically instrument programs as well as allow manual instrumentation. Possible directions for future work include the following: (i) Currently, only those patterns that are associated with function calls are recognized. This should be extended to more complex patterns. (ii) In some cases, not all files or communications need to be monitored. The toolset should provide the ability to specify a constraint on where the instrumentation should take place. (iii) The current toolset prototype was implemented to demonstrate that the toolset design concepts were sound. More work can be done on strengthening the prototype to make it more robust and expand it to other environments such as *CORBA*. (iv) The toolset prototype should be further evaluated by instrumenting larger distributed applications. This will help us see how the current implementation, of our prototype, scales up when instrumenting large applications.

Acknowledgements

We would like to thank Gary Molenkamp, Michael Katchabaw and other members of the Distributed Systems Research Lab at the University of Western Ontario. This work is supported by the National Sciences and Engineering Research Council (NSERC) of Canada and the IBM Centre of Advanced Studies in Toronto, Canada.

References

[1] U. Blumenthal, G. Kar, and A. Keller. Classification and computation of dependencies for distributed management. *IEEE Symposium on Computers and Communications (ISCC 2000)*, July 2000.

[2] Y. Chang. Pablo MPI Instrumentation User's Guide. *Technical report, Univ. of Ill*, 1999.

[3] James R. Cordy and Ian H. Carmichael. *The TXL Programming Language Syntax and Semantics*. Software Technology Laboratory, Department of Computing and Information Sciences, Queen's University at Kingston, Kingston, Canada, June 1993. Version 7.

[4] R. Hauck. Architectuer for an automated management instrumentation for component based applications. *Proceedings of the 12th International Workshop on Distributed Systems: Operations and Management DSOM'2001*, Nancy France, October 2001.

[5] J. Hollingsworth, B. Miller, M. Goncalves, O. Naim, Z. Xu, and L. Zheng. MDL: A Language and a Complier for Dynamic Program Instrumentation. *International Conference on Parallel Architectures and Compilation Techniques*, 1997.

[6] M. Katchabaw, S. Howard, H. Lutfiyya, A. Marshall, and M. Bauer. Making Distributed Applications Manageable through Instrumentation. *The Journal of Systems and Software*, 45:81–97, 1999.

[7] W. Rosenberry, D. Kenney, and G. Fisher. *Understanding DCE*. O'Reilly and Associates, Inc., 1993.

[8] J. Yan, S. Sarvkkai, and P. Mehra. Performance Measurement, Visualization and Modeling of Parallel and Distributed Programs using the AIMS Toolkit . *Software Practice and Experience*, 25(4):429–461, April 1995.