Coarse-Grained Dynamic Predicate Slicing for Message Passing Programs

Hai Hong Song

A Thesis
in
the Department
of
Computer Science

Presented in Partial Fulfillment of the Requirements
for the Degree of Master of Computer Science at
Concordia University
Montreal, Quebec, Canada

January 2006

Your file Votre référence
ISBN: 0-494-14336-3
Our file Notre référence
ISBN: 0-494-14336-3

Abstract

Coarse-Grained Dynamic Predicate Slicing for Message Passing Programs

Hai Hong Song

Comprehending distributed systems is a challenging task because of interdependency and non-determinability that exist in distributed systems. Program slicing, as a well-know decomposition and reduction technique, has been extended to assist during the comprehension of distributed application source code. Dynamic predicate slicing is a relatively new slicing technique that adopts the notion of global predicates as slicing criteria for distributed message passing programs. Dynamic predicate slicing focuses on identifying all these states during an execution of a message passing program, in which a particular predicate might be changed. In this research, a Coarse-Grained dynamic predicate slicing algorithm is implemented by using instrumentation techniques to insert extra instructions into applications for collecting and analyzing run-time information during executions. Dynamic predicated slicing is accomplished by multi-thread parallel computation utilizing both static information and dynamic information. An initial case study is presented to validate the applicability of the approach and to explore the overhead associated with collecting the dynamic information and the slice computation.

Acknowledgements

I would like to express my sincere gratitude to my supervisor, Dr. Juergen Rilling, for his support, guidance, encourage, and patience throughout my research work. Without his guidance, this thesis would not have been possible.

I am grateful to all the members of the CONCEPT research group, and particularly to Zhang, Yonggang, for his helps on the Java parser. I appreciatively acknowledge Cecile Dupin and Charles Lewis for having proofread earlier versions of this thesis and provided valuable comments and improvements.

Finally, I would like to thank my wife, my parents, and my family for their support, help, and encouragement.

Table of Contents

List of Figures

List of Tables

Chapter 1 INTRODUCTION

Software systems need to evolve and be changed while external requirements and hardware technology are updated. Before any change or evolution is made, software engineers have to well-understand these original systems. Moreover, the majority of software development effort is spent on maintaining existing systems rather than developing new ones. Software maintenance is estimated to consume 50% to 75% of the resources and time in the total software budget, and within software maintenance, comprehension requires 47% and 62% of the total time for enhancement and correction tasks respectively [Rug95]. Therefore, software comprehension/understanding plays a significant role in software development and evolution. However, software comprehension is a challenging task because it involves mapping different conceptual areas that makes it much more difficult to achieve.

Distributed systems are a popular and powerful computing paradigm. They are ubiquitous today throughout business, academia, government, and homes. The population of distributed systems makes comprehension of distributed systems critical for the development and maintenance of a system involving some distributed components. Because of inter-dependency between processes caused by communication or

synchronization and non-determinability of executions of distributed systems, comprehending distributed systems is more challenging than comprehending a traditional single-process sequential system.

Slicing as a well-known program decomposition technique was first introduced by Weiser [Wei81] into the software field in order to reduce the complexity of program debugging. It has been widely used in the software comprehension field. Program slicing technique is classified into two categories, static slicing and dynamic slicing, where static slices are computed based on static information about program, and dynamic slices are computed according to both static information and dynamic information about a particular execution of program. Program slicing has also been extended for comprehending distributed systems.

Rilling et al [Ril02a] introduced a novel dynamic predicate-based slicing technique for distributed message passing programs. They defined a dynamic predicate slice as those parts of a message passing program that are relevant to a global predicate during an execution of the program. Li et al [LiH04] extended the dynamic predicate slicing to two kinds of granularity-driven dynamic predicate slicing. Predicate slicing focuses on all states, in which the predicate might be changed during an execution of a message passing program, allows for more general slicing criteria, and supports more general

comprehension tasks for message passing programs.

In this research, we implement the dynamic predicate slicing algorithm presented by Rilling et al [Ril02a] [LiH04] as a part of work of the Comprehension Of Net-CEntered Programs and Techniques (CONCEPT) project. Source code and bytecode instrumentation techniques are used for collecting dynamic information about executions of a message-passing program. The dynamic predicate slicing algorithm is then implemented based on the static information of the program and the dynamic information collected by instrumented code during an execution.

The outline of the thesis is as follows: Chapter 2 introduces some general background related to software comprehension, message-passing programs, program slicing, and instrumentation and program tracing techniques. Chapter 3 details the concepts about dynamic predicate slicing, and the algorithm that will be implemented as part of this research. Chapter 4 discusses the motivation and hypotheses of this research, and proposes the research goal and a general overview of the implementation. Chapter 5 describes implementation details of the predicate slicing algorithm, and presents results of a preliminary experimental analysis. Chapter 6 finally provides conclusions and future works.

Chapter 2 BACKGROUND

This chapter presents some of the technical background of this thesis. First, section 2.1 provides definitions of reverse engineering and program comprehension, discusses difficulties associated with program comprehension. Distributed systems and message passing programs are introduced and challenges in distributed system comprehension are discussed in section 2.2. In section 2.3.both program slicing and distributed program slicing techniques are reviewed. The last section in this chapter presents in detail the dynamic analysis techniques used in the research.

2.1 Program Comprehension

"Programs, like people, get old."

By D. L. Parnas [Par94]

2.1.1 Reverse Engineering and Program Comprehension

- Definition

Chikofsky and Cross [Chi90] indicate that the term "reverse engineering" originally came from the hardware analysis field. During reverse engineering, engineers, other than the developer, examine an existing product in order to obtain its design information without the aid of any original drawings of the examined product. Applying the term to

software systems areas, reverse engineering is "the process of identifying software components, their interrelationships, and representing these entities at a higher level of abstraction" [Chi90] [Nel96]. Differing from traditional reverse engineering in the hardware analysis field, the goal of software reverse engineering is most often to gain an adequate understanding to help software maintenance, strengthen enhancement, or support replacement rather than to reproduce the product or system [Chi90]. From its definition, reverse engineering can be applied to any stage of the software development cycle, such as requirements recovery, re-documentation, and design rediscovery, even recompile from executable binary code. Virtually, all reverse engineering processes start from the source code of a system. The software source code is usually available as the input to the reverse engineering process [Som01].

Reverse engineering by itself involves only analysis, not change. The program itself is unchanged by the reverse engineering process. Sommerville [Som01] distinguishes between reverse engineering and re-engineering. The objective of reverse engineering is to derive the design or specification of a system from its source code; but the objective of re-engineering is to produce a new, more maintainable system. Reverse engineering is often part of the re-engineering process.

Program comprehension is an emerging interest area within the software engineering field. Program comprehension or program understanding is a cognitive process that uses

existing knowledge (i.e. the source code of a software system) to acquire new knowledge that meets the goals of a code cognition task. Rugarber gives the following definition: "Program Comprehension is the process of acquiring knowledge about a computer program" [Rug95]. As mentioned above, most software reverse engineering processes start from the source code, and in most cases, the source code of a program is the only reliable documentation of the behaviors of a software system. Therefore, to some degree, program comprehension and program understanding are terms that are often used interchangeably with reverse engineering [Nel96]. However, reverse engineering can widely be applied to all stages of the software development cycle. Program comprehension main concern is extracting information from the source code of a software system.

- Motivation

Software systems always need to be changed after they are delivered [Som01]. Once software is put into use, new requirements emerge, and existing requirements change as the business rules change. Parts of the software may have to be modified to correct errors that are found in operation, or to improve its performance or other non-functional characteristics. All of this means that, software systems evolve in response to demands for change in the lifetime of software systems. Parnas [Par94] says that "programs, like people, get old." Programs age due to continuous software changes and changes of external

environmental factors, like changes in business requirements, hardware platforms etc. Software change and software aging are becoming very important problems that organizations must face because organizations are now completely dependent on their software systems and have invested millions of dollars in these systems. Those old but important systems are called "Legacy Systems".

According to [Som01], legacy systems are the software systems that organizations still rely on. These systems may be more than 10 or 20 years old, but they are still business-critical. The business relies on the services provided by the software and any failure of these services would have a serious effect on the day-to-day running of the business. These legacy systems are not necessarily the systems that were originally delivered because of inevitable software changes. External and internal factors, such as the state of the national and international economies, changing markets, changing laws, management changes and structural reorganization, cause businesses to undergo continual changes. Consequently, software requirements and software itself change. For a legacy system, a complete specification of the legacy system is rare. Important business rules may be embedded in the software and may not be documented elsewhere. Moreover its system documentation is often inadequate and out of date. In some cases, the only reliable documentation is the system source code. These situations make maintenance and migration of these legacy systems particularly expensive and difficult. Sommerville

[Som01] says that a key problem for organizations is implementing and managing change to their legacy systems so that they can continue to support their business operations. Software maintenance, architectural transformation, and software re-engineering are main strategies to resolve the software change and aging problems for legacy systems.

Additionally, as currently practiced, the majority of the software development effort is spent on maintaining existing systems rather than developing new ones. Rugaber [Rug95] records that the estimates of the proportion of resources and time devoted to maintenance range from 50% to 75%. The greatest part of the software maintenance process, in turn, is devoted to understanding the system being maintained. Fjeldstad and Hamlen report that 47% and 62% of time spent on actual enhancement and correction tasks respectively, are devoted to comprehension activities. These involve reading the documentation, scanning the source code, and understanding the changes to be made [Rug95] [Nel96]. The implications are that if we want to improve software development, we should look at maintenance, and if we want to improve maintenance, we should facilitate the process of comprehending existing programs.

Software engineering itself is concerned with improving the productivity of the software development process and the quality of the systems it produces. In contrast, program comprehension involves acquiring knowledge about a computer program. Increased knowledge enables such activities as bug corrections, enhancements, reuse, and

documentation. Program comprehension and reverse engineering play prominent roles in the software development, and therefore are key factors in software maintenance and evolution, legacy system change and migration.

2.1.2 Difficulties in Program Comprehension

Program comprehension is not an easy task in the software engineering field. In this section, we discuss some general comprehension challenges. A more focused discussion on the specific challenges of program comprehension for distributed systems is present in section 2.2.4. Program comprehension involves mapping different conceptual areas that makes it much more difficult to achieve. Rugaber [Rug95] and Nelson [Nel96] summarized the different conceptual areas that correspond to the following five gaps.

- *The gap between a problem from some application domain and a solution to the problem in some programming language*

  A programming language is just a model environment to solve some real problem. While tools exist to assist in understanding what the code is doing from a code perspective, there is little to assist the reverse engineer in determining what is occurring with the code from a domain perspective.

- *The gap between the concrete world of physical machines and computer programs and the abstract world of high level design descriptions*

Simple, abstract concepts quickly become lost in the trivial detail of programming. Computer science education is largely about mapping from the abstract to the detailed implementation, but there is little to assist in the reverse mapping.

- *The gap between the desired coherent and highly structured description of a system as originally envisioned by its designers and the actual system whose structure may have disintegrated over time.*

  Even when good documentation is available for a system, maintenance over time causes the structure to drift from the original specification. The reverse engineer must be able to reconcile and synchronize the documented design and the current implemented design.

- *The gap between the hierarchical world of programs and the associational nature of human cognition*

  Computer programs are formal, hierarchical expressions. Humans think in associative chunks of data. A reverse engineer must be able to build up correct high-level chunks from the low-level details evident in the program.

- *The gap between the bottom-up analysis of the source code and the top-down synthesis of the description of the application*

  Code analysis is by its nature a bottom-up exercise. It requires, simultaneously,

higher level meaning to be extracted from code fragments, and higher-level concepts to be mapped to lower level implementations.

2.1.3 Program Comprehension Approaches

There are varieties of approaches for automated assistance that are available for program comprehension. Rugaber [Rug95] presented a full list of program comprehension approaches. Some of the more prominent approaches include:

- *Textual, lexical and syntactic analysis* – these approaches focus on the source code itself and its representations. These include the use of UNIX's lex, lexical metrics and even automated parsing of the code searching for clichés.

- *Graphing methods* – there is a variety of graphing approaches for program understanding. These include, in increasing order of complexity and richness: graphing the control flow of the program, the data flow of the program, and program dependence graphs.

- *Execution and testing* – there is a variety of methods for profiling, testing, and observing program behavior, including actual execution and inspection walkthroughs. Dynamic testing and debugging is well known and there are several tools available for this function.

2.2 Distributed System

Distributed systems are a popular and powerful computing paradigm. They are ubiquitous today throughout business, academia, government, and homes. Various definitions of distributed system have been given in literature.

2.2.1 Definition and Characteristics

Tanenbaum and Steen define a distributed system as "A distributed system is a collection of independent computers that appears to its users as a single coherent system" [Tan02]. In their definition, they focus on two aspects of the distributed system: the hardware - the independent machines, and the software - users think they are dealing with a single system.

Attiya and Welch provide another definition that is "A distributed system is a collection of individual computing devices that can communicate with each other" [Att04], which gives attention to the communication of the distributed system. In the distributed system, each semi-dependent machine communicates with others in order to cooperate to accomplish a task.

Coulouris et al's [Cou01] definition is the one that is adopted for this thesis. A distributed system is one in which hardware or software components located at networked computers communicate and coordinate their actions only by passing messages. The definition indicates that machines or components in a distributed system cooperate with

each other only through passing messages, although there is still the Distributed Shared Memory (DSM) architecture existing for distributed systems. In fact, the DSM is implemented based on a message passing system. A DSM is a simulation of an asynchronous shared memory model by the asynchronous message passing model, which runs on top of the message passing system providing the illusion of shared memory [Att04]. In the same way, Andrews describe the distributed system as distributed-memory multi-computers, in which each processor has its own private memory, and the interconnection is achieved by message passing [And00]. The processors communicate with each other by sending and receiving messages. In the following part of the thesis, the distributed system will be mentioned same as message passing system for our purpose. Figure 2.2.1 illustrates a distributed system.

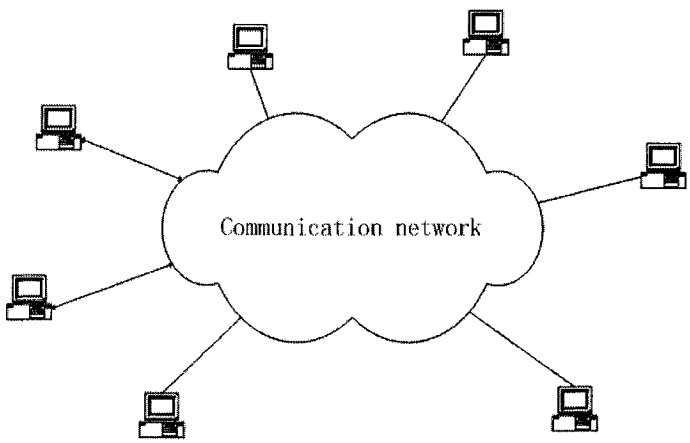

Figure 2.2.1 Distributed System

Mullender summarized in [Mul93] the following primary characteristics and important issues related to a distributed system:

- *Multiple Computer* – A distributed system contains more than one physical computer, each consisting of CPUs, some local memory, possibly some stable storage like disks, and I/O paths to connect it with the environment.
- *Interconnections* – Some of the I/O paths will interconnect the computers. If they cannot talk to each other, then it is not going to be a very interesting distributed system.
- *Shared State* – The computers cooperate to maintain some shared state. Put another way, if the correct operation of the system is described in terms of some global invariants, then maintaining those invariants requires the correct and coordinated operation of multiple computers.
- *Independent Failure* – Because there are several distinct computers involved, when one breaks, others may keep on going. Often users want the system to keep on working after one or more have failed.
- *Unreliable Communication* – Because, in most cases, the inter connections among computers are not confined to a carefully controlled environment, they will not work correctly all the time.
- *Insecure Communication* – The interconnections among computers may be exposed to unauthorized eavesdropping and message modification.
- *Costly Communication* – The interconnections among the computers usually provide lower bandwidth, higher latency, and higher cost communication than that available

between the independent processes within a single machine.

The canonical example of a general-purpose distributed system today is a networked system, a set of workstations/PCs and servers interconnected with a network. Networked systems allow the sharing of information and resources over a wide geographic and organizational spread. They allow the use of small, cost-effective computers and get the computing cycles close to the data. They can grow in small increments over a large range of sizes. They allow a great deal of autonomy through separate component purchasing decision, selection of multiple vendors, use of multiple software versions, and adoption of multiple management policies. Finally, they do not necessarily all crash at once. Thus, in the areas of sharing, cost, growth, and autonomy, networked systems are better than traditional centralized systems as exemplified, by timesharing. On the other hand, centralized systems do some things better than today's networked systems. All information and resources in a centralized system are equally accessible. Functions work the same way and objects are named the same way everywhere in a centralized system.

2.2.2 Concurrent, Parallel and Distributed Systems

In most cases, *concurrent*, *parallel* or *distributed* program or system are similar terms, and often people usually use these terms interchangeably, but there are still some subtle differences between them, and the terms are also confusing.

The word *concurrent* means that certain actions happen at the same time, thus a

concurrent program consists of a number of execution units (either processes or threads) that work together by communicating with each other to perform tasks at once [Spi99]. Communication is programmed using shared variables or message passing. Concurrent programs are typically written for one of two reasons: to improve performance or to satisfy an inherently concurrent specification [And00]. Concurrent programs can be executed on a single processor (by interleaving process execution) or on a multiple instruction stream, multiple data stream (MIMD) multiprocessor. Andrews also points out that the term concurrent program refers to any program that contains multiple processes.

Parallel programs are a different subset of concurrent programs. It refers to solving a task faster by employing multiple processors simultaneously [Leo01]. The distinguishing attribute of a parallel program is that it is written to solve a problem in less time than that would be taken by a sequential program [And00]. In other words, the main goal of parallel programs is to reduce execution time. One reason for writing parallel programs is to solve larger problems in the same amount of time. Another reason is to solve more instances of the same problem in the same amount of time. In the same way, a parallel program can be written using either shared variables or message passing.

As defined above, a *distributed system* is a collection of autonomous computers that are interconnected with each other and cooperate, thereby sharing resources such as printers and databases. The distributed system and parallel system have many common

characteristics: multiple processors are used in both systems, the processors are interconnected, and multiple processes are in progress at the same time and cooperate with each other. Leopold distinguish parallel and distributed computing: parallel computing splits an application into tasks that are executed at the same time, whereas distributed computing splits an application into tasks that are executed at different locations, using different resources [Leo01]. For parallel programs, an application is split into subtasks that are solved simultaneously, often in a tightly coupled manner. The programs are usually run on homogeneous architectures, which may have a shared memory. In contrast, a distributed system uses multiple resources that are situated in physically distant locations, and distributed systems are often heterogeneous, open and dynamic. In addition, distributed systems do not have a shared memory, at least not at the hardware level.

2.2.3 Message Passing Programs

As noted in the previous section, in distributed systems, the basic approach to inter-process communications is message passing, the most fundamental paradigm for distributed applications. In this paradigm, data representing messages are exchanged between two processes, a sender and a receiver. A process sends a message representing a request. The message is delivered to a receiver, which processes the request, and sends a message in response. In turn, the reply may trigger a further request, which leads to a subsequent reply, and so forth. The message-passing paradigm has been widely used in the

development of applications, including well known Internet services such as the Hypertext Transfer Protocol (HTTP, commonly known as "the web") and file transfer protocol (FTP). Figure 2.2.2 illustrates the message-passing paradigm.

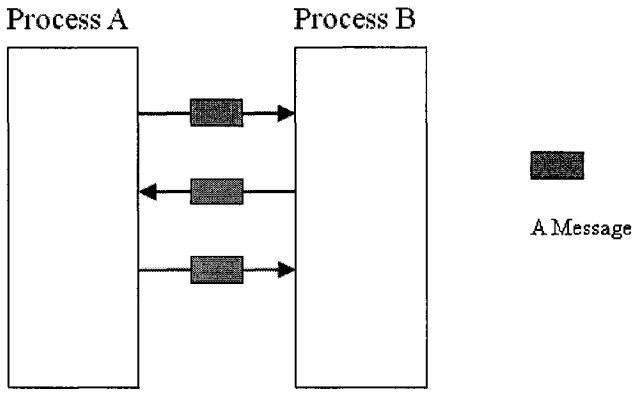

Figure 2.2.2 Message Passing Paradigm

In a message-passing model, several processes run in parallel and communicate with one another by sending and receiving messages. The processes do not have access to a shared memory. Hence, in message passing, the processes operate on disjoint address spaces, and all communication is accomplished through explicit message exchange. The central communication routines are *"send"* and *"receive"*. In order to exchange information for two processes, one of them must invoke a *send*, and the other must invoke a matching *receive*. It is always exactly two processes that are involved in such a communication. The functions *send* and *receive* exist in several variants, and practical message passing systems add further functions [Leo01].

Message exchange may serve different purposes. The most obvious purpose is the

exchange of data between a sender and a receiver that know each other and whose interaction was planned in detail by the programmer. A second purpose is the establishment of a connection between a sender and a receiver whose interaction was not planned in advance. A connection can only be established if the receiver is prepared for an eventual request: it must have posed an anonymous *receive*, and it must occasionally check whether a message has arrived or not. An eventual sender will use the initial message to transmit to the receiver details about the desired transaction. For instance, it can state the length of a message to be sent or the location of requested data. A third purpose of message exchange is synchronization. A process may send a message in order to indicate that it has reached a certain point of program execution. Note that synchronization is just a special case of communication. Separate synchronization routines as in shared-memory programming are not needed, since each communication implicitly induces synchronization between sender and receiver.

The message-passing model closely follows the architecture of distributed memory machines. Therefore it is easy to implement, but it operates on a low level. In consequence, message passing imposes a heavy burden on the programmer, who is responsible for managing all details of data distribution and task scheduling, as well as of the communication between tasks. Particular responsibilities include load balancing, data replication, and the maintenance of coherency. Message passing is therefore

time-consuming and error-prone and the programs are hard to maintain and debug. However, the primary argument for message passing is efficiency. Since everything is under the programmer's control, the programmer can achieve close to optimum performance if the programmer just spends enough time in performance tuning. Efficiency is the major reason for the predominance of message passing in computer-intensive application domains.

2.2.4 Distributed System Comprehension

The previous section described characteristics of distributed systems that make distributed applications inherently more complex and harder to understand. Mullender, for example, notes that "all [problems] exist in all computer systems, but they are much more apparent in distributed systems; there are just more pieces, hence more interference, more interconnections, more opportunities for propagation effect, and more kinds of partial failure." [Mul93]

- *Interdependencies Caused by Synchronization and Communication*

Dependability is one of the key factors that affects the design and implementation of distributed control systems. In general, a parallel and/or distributed program consists of a number of processes, and therefore, it has multiple threads of control flows and multiple threads of data flows. These control flows and data flows are not independent because of

the existence of inter-process synchronization among multiple control flows and inter-process communication among multiple data flows in the program. Synchronization is used for concurrent control, which might introduce synchronization dependence (a kind of control dependences) between concurrently executed statements. By accessing shared memory, inter-thread data dependences (communication dependence) occur.

The comprehension of large software systems consisting of many processes/threads is a challenging task due to the timing related interdependencies among processes that influence the comprehension process. Consequently, the task of developing a mental representation for comprehending parallel programs is more difficult, and it is critical that techniques are developed to automate and support the analysis and comprehension of distributed programs.

- *Non-Determinability and Non-Reproducible*

Another challenge is the potential of non-determinism in distributed programs. Distributed programs often make non-deterministic decisions; the order in which concurrently sent messages are received is time dependent and thus may vary from one execution to the next. It follows that repeated executions of a program on the same input may result in the execution of different program paths. However, when debugging, the programmer is interested in the exact execution that exhibits the error. Thus, unlike

re-executing sequential program slices, a slice for re-execution of a distributed program must be based on dynamically collected information. In order to precisely reproduce the original behavior, all nondeterministic decisions that are made during execution must be recorded, and program replay must be instrumented to reproduce the recorded decisions.

Moreover, a process in a parallel and/or distributed program may non-deterministically select a communication partner among a number of processes ready for communication with the process.

Communication delays among cooperating devices in a distributed computing system make it difficult to determine the system's state at any given time. The kinds of confederations being proposed seem to be inherently asynchronous, and therefore non-deterministic. There is no way for two executions to produce anything but different orderings of events, both of which may be valid at the time they occur. Therefore it is difficult to reproduce errors and to test possible, but not likely, situations.

2.3 Program Slicing

Program slicing first was introduced by Weiser into the software field in order to reduce the complexity of program debugging [Wei81]. A slice is the abstracted programs that keep the same specified subset of the original program's behavior. The process of getting a slice based on a specified request (slicing criterion) is called program slicing. Weiser [Wei82, Wei84] defines a slice as a reduced, executable program that preserves the original

behavior of the program with respect to a subset of variables of interest at a given program point. That is, a program slice consists only of these parts of a program that can potentially affect the values computed at a point of interest. Such a point of interest, that combines a program location and a subset of program variables of interest, is called slicing criterion.

Slicing is a method of program decomposition, and is applied to programs after they are written, therefore is useful in maintenance. Working on actual program text allows slicing to be specified precisely and performed automatically. The process of slicing deletes those parts of the program that can be determined to have no effect upon the semantics of interest. Program slicing enables programmers to view subset of program by filtering out code that is not relevant to the computation of interest. Slicing has applications in testing, debugging, reengineering, program comprehension, and software maintenance [Tip95] [DeL01] [Har01] [XuB05].

2.3.1 Static Slicing

The program slicing approach originally defined by Weiser is of a static nature. For static slicing, only statically available information is used for computing slices, hence this type of slice is referred to as a static slice. The result of static slicing is independent from the program input. The behavior of a static slice remains the same as the original programs for any given input.

According to the definition given by Weiser [Wei82], a static program slice S consists of

all statements in program ***P*** that may affect the value of variable ***v*** at some point ***p***. In a more formal way, the slice is defined for a slicing criterion $C = (x, V)$, where x is a statement in program ***P*** and V is a subset of variables in ***P***. A static program slice on the slicing criterion $<x, V>$ is a subset of program statements that preserves the behavior of original program at the program point p with respect to the program variables in V, i.e. the values of the variables in V at program point p are the same in both the original program and the slice.

In Weiser's approach [Wei84], static slices are computed by finding consecutive sets of indirectly relevant statements, according to data and control dependencies. Date dependence and control dependencies are defined in terms of the Control Flow Graph (CFG) of a program. A CFG contains a node for each statement and control predicate in the program; an edge from node ***i*** to node ***j*** indicates that the possible flow of control from the former to the latter. A CFG also contains special nodes labeled ***Start*** and ***Stop*** corresponding to the beginning and the end of the program respectively. Ottenstain presents a different algorithm to compute slices as backward traversals of the program dependence graph (PDG) [Ott84]. PDG is another program representation where nodes represent statements and predicates, while edges carry information about control and data dependences. Horwitz et al. [Hor90] extended the PDG based algorithm to compute inter-procedural slices on the System Dependence Graph (SDG). Many different

extensions of static slicing approaches have been proposed in the literature [XuB05].

2.3.2 Dynamic Slicing

Although static slicing can assist the program comprehension effort by simplifying the program under consideration, the static slices tend to be rather larger, especially for well-constructed programs, which are typically highly cohesive. In addition, a static slice may very often contain statements that have no influence on the values of the variables of interest for the particular execution in which the anomalous behavior of the program was discovered. In order to reduce the size of a slice and get more accurate slice concerning a particular execution, Korel and Laski [Kor88] originally introduce dynamic slicing. Dynamic program slicing refers to a collection of program slicing methods that are based on program executions. It may significantly reduce the size of a program slice. This slice reduction is possibly due to run-time information collected during program execution. This information can be applied to resolve some of the conservative assumptions that have to be made by static slicing regarding the control flow.

A dynamic program slice is this part of a program that affects the computation of a variable of interest during program execution on a specific program input. Formally, a dynamic slice is taken with respect to a set of variables ***V,*** input **I**, and point **P** in the execution history. The execution history of the dynamic slice is equivalent to the execution history of the original program after removing the occurrences of statement not in the

dynamic slice. The set of variables V and the point of interest within the program are just the same as these in static slicing. However, a dynamic slicing criterion specifies the input, and distinguishes between different occurrences of a statement in the execution history.

In [Kor88], Korel and Lasky proposed an iterative algorithm based on dynamic data flow dependences relations between statements. The dynamic slice computed by Korel's algorithm is executable. Agrawal and Horgan [Agr90] present another algorithm to produce dynamic slices that are not executable. Dynamic Dependence Graphs are used in their slicing algorithm. The algorithms presented by Korel and Agrawal's are both backward approaches. In order to compute a dynamic slicing, an execution trace has to be recorded first, and then this trace is traversed backwards to derive data and control dependencies to compute the dynamic slice. Korel and Yalamanchili [Kor94] present a forward algorithm that computes dynamic slices during program executing without major recording of the execution trace. Different extensions of dynamic slicing have also been proposed [XuB05].

2.3.3 Distributed Program Slicing

Program slicing, is an important and efficient software analysis technique that has also been widely applied in the distributed software field. The comprehension of a distributed program is a major challenge due to timing related interdependencies among processes and to the potential for non-determinism for a distributed program, that add complexity to the

comprehension process (detailed discussion was presented in section 2.2.4). By eliminating irrelevant portions and providing some level of abstraction of the program, program slicing can significantly reduce the required analysis effort for distributed program comprehension.

Most of the existing methods for slicing concurrent programs are based on the notion of graph reachability. One of the earliest approaches to static slicing of threaded programs was presented by Cheng [Che93]. He extended the notion of slicing for concurrent programs and presented a graph-theoretical approach to slicing concurrent programs based on a program representation, named Program Dependence Nets (PDN). A PDN includes new types of primary program dependences in concurrent programs, named the selection dependence, synchronization dependence, and communication dependence. Zhao et al. [Zha96] proposed a new program dependence representation named the System Dependence Net (SDN), which extends previous dependence representations to represent concurrent object-oriented programs. The system dependence net of a concurrent object-oriented program can be used to represent not only object-oriented features but also concurrency issues in the program, and allows us to compute slices of the program efficiently. Zhao [Zha99a] [Zha99b] addresses static slicing of Java multithread programs using thread dependence graphs (TDG). Those approaches can handle method calls and synchronized methods, but not synchronized statements. Krinke [Kri98] considers in his

work static slicing of multi-threaded programs with shared variables and focuses on issues associated with interference dependence. In his approach however, there is no explicit synchronization mechanism.

For dynamic slicing of distributed programs, Duesterwald et al. [Due92] introduced an execution trace representation, called the Distributed Dependence Graph (DDG) for distributed programs, and showed a parallel algorithm to compute dynamic slices of a distributed program based on its distributed dependence graph representation. The algorithm computes a slice for a particular slicing criterion using static control and data dependencies refined by dynamic communication dependencies. A parallel algorithm extracts program slices from the DDG in a fully distributed fashion, where each process identifies its local portion of the global slice. Korel and Ferguson [Kor92] proposed a method to compute dynamic slices of a distributed Ada program by analyzing influences in and between multiple executed paths of the program. For a distributed program, the execution history is recorded as a distributed program path. The slice is only guaranteed to preserve the behavior of the program if the rendezvous in the slice occurs in the same relative order as in the program. Kamkar and Krajina [Kam95] introduced a program presentation, called Distributed Dynamic Dependence Graph (DDDG) that represents control, data, and communication dependences in a distributed program. The graph is built at run-time and it is used to compute dynamic slices of the program.

Because of the existence of inter-process synchronization among multiple threads of control flow, inter-process communication among multiple threads of data flow, and the potential for non-determinism in the distributed system, the approaches above have different limitations. Although Cheng [Che93] and Duesterwald [Due92] use static dependence graphs for computing dynamic slice, the computed slice is not proved to be precise. In addition, the computed slices in Cheng [Che93] are not executable programs. For Zhao [Zha99a] and Krinke's [Kri98] approaches, there is no explicit synchronization mechanism. Korel and Ferguson [Kor92] and Duesterwald [Due92] compute slices that are executable programs, but deal with non-determinism in a different way. In addition, the algorithm presented by Kamkar [Kam95], Duesterwald [Due92] and Korel [Kor92] only supports a structured programming language subset.

2.4 Instrumentation and Program Tracing

Dynamic analysis evaluates a software system or components based on their behavior during execution. It takes advantage of the more detailed and precise (compared to static analysis) information available based on some program inputs [Bal99]. Dynamic analysis plays a significant role in software comprehension. Instrumentation and program tracing both are important approaches to collect information relevant to dynamic analysis.

Instrumentation, typically inserts additional instructions into the application under investigation to allow for collecting and analysis of certain run-time states and aspects of a

system. The collected information is then applied for dynamic analysis tools, such as profilers, instruction trace generators, monitors, test tools, etc. For Java applications, there are three types of instrumentation techniques; one approach is to instrument at the source code level. The second approach instruments the byte code generated for an existing system. The third approach is interfacing with the Java Virtual Machine through the Java Debugger Interface (JDI). The third approach is not covered in this research due to the limited applicability of this approach. Interfacing with the virtual machine can provide detail run-time information, however the performance penalty using the JDI is so significant that its applicability is limited only for very short program executions. Depending on the level of analysis (variable, statement, function or class level) the overhead associated for extracting run-time information using the JDI is between 10 – 1000 times of the original execution time. Furthermore, there would be an additional overhead for storing the information. Therefore, we do not cover this approach in detail in our literature review. Sections 0 and 2.4.2 discuss in more detail both, source code and byte code instrumentation. Recording of dynamic information (run-time information) requires some type of tracing support. Program tracing can be described as the process of recording program executions and has been applied in debugging, testing, monitoring, and comprehension etc. Instrumentation is usually used to generate trace information. Section 2.4.3 discusses in more detail challenges related to program tracing and their applicability

in dynamic program slicing.

2.4.1 Source Code Instrumentation

During source code instrumentation, extra instructions are inserted directly into the original source code. After the instrumentation, the modified source code has to be recompiled in order to be able to execute the instrumented application and obtain dynamic information. Therefore, the source code of an application must be available and moreover, there is an additional overhead for recompiling the instrumented source code. On the other hand, the first advantage of the source code instrumentation is that no specialized runtime environment is required. After the instrumented source code is recompiled, it can run within the same program environments as the original programs, i.e. the same JVM and the class loader. The second advantage of source code instrumentation is its support for statement level source code analysis, such as source code coverage tools, statements and branch coverage [Clo05]. The statement level information is available for the source code instrumentation because it operates directly on the original statements of source code. Several source code instrumentation tools and applications are discussed below.

"*Clover*" is a commercial code coverage analysis tool, developed by Cenqua Pty Ltd. [Clo05]. It is used to discover the sections of code that are not being adequately exercised by unit tests, and then used to measure testing completeness. *Cover* utilizes source code instrumentation. It copies and instruments a set of Java source files. The output

instrumented java source will then be compiled by a standard Java compiler. *Clover* measures three types of coverage analysis: statement, branch and method coverage. Moreover, *Clover* provides fully integrated plug-ins for many popular integrated development environments, such as *Eclipse*, *NetBeans*, *JBuilder* and *JDevelope,* and works seamlessly with *JUnit.*

"Daikon" is a dynamic invariant detector, developed by the Program Analysis Group in Massachusetts Institute of Technology [Dai05]. An invariant is a property that holds at a certain point or points in a program. Invariants are often seen in assert statements, documentations, and formal specifications. *Daikon* dynamic invariant detector runs a program, observes the values that the program computes, and then reports properties that were true over the observed executions. *Daikon* can detect properties in *Java*, *Perl*, *C*, *C++*, and *IOA* programs, in spreadsheet files, and in other data sources. *Daikon* provides a plug-in for *eclipse*, which instruments Java source code, obtains trace information, analyzes those traces, and creates appropriately annotated Java source code to represent the invariants found by *Daikon* while running the instrumented program.

"query and instr" are Java test coverage and instrumentation toolkits developed by Glen McCluskey & Associates LLC [McC05]. They are used for parsing, querying, and instrumenting Java source programs, and are ideally suited for applications, such as test coverage, metrics, instrumentation, extraction of information, documentation tools,

program tracing, and so on. The toolkits have two packages: *query* and *instr*. The *query* package is used to parse Java source programs into an internal tree form. It also contains classes and methods for querying the parse tree. The tree may be annotated; that is it may have information added to it which could be dumped out at a later time. This is the basis for program instrumentation. The *instr* is a test-coverage and instrumentation package, which is built on the *query* package [McC05]. It supports method and statement source instrumentation, and test coverage. These can be used as actual end-user programs to instrument code, or as the basis for customized applications.

2.4.2 Java Bytecode Instrumentation

The *Java Virtual Machine* (JVM) known as a runtime interpreter is the cornerstone of the Java platform. It is the component of Java that is responsible for its hardware and operating system independence. The JVM is an abstract computing machine. Like a real computing machine, it has an instruction set, known as JVM bytecode [Lin99]. The Java source code is normally compiled in a binary format to a bytecode instruction set (i.e. the class file) as an intermediate format. The Java bytecode is hardware independent and operating system independent. So the high level meaning of Java source code is first transformed by the Java compiler to this intermediate representation before execution.

Java bytecode instrumentation, also called bytecode injection, bytecode insertion or class file transformation, is the process of directly inserting or manipulating Java bytecode.

The instrumentation of Java bytecode generally inserts a special, short sequence of bytecode at the designated points in a Java class file. This transformation process must be strict and should adhere to the constraints imposed by the JVM Specification on the Java class file format, so any modification of JVM bytecode should be reflected on all other JVM bytecode within a Java class file.

The bytecode instrumentation can be performed either statically at the compile time or dynamically at the runtime. The static instrumentation of bytecode can occur during or after compilation of a Java source file. The instrumented bytecode is saved in a class file, like typical Java classes files, and then executed later by the JVM. The dynamic bytecode instrumentation takes place at runtime. A typical way to perform runtime instrumentation is to insert bytecode into a Java class when the bytecode of the class is being loaded into the JVM. The dynamic bytecode instrumentation can also be applied at runtime to redefine a loaded class or create a new class from scratch [Dah01].

The Java bytecode instrumentation allows researchers or software developers to perform dynamic analysis of those instrumented classes, mostly for debugging, testing, profiling, monitoring, or other introspection purpose. Several projects deal with instrumenting bytecode. Some existing works in this area are presented below.

The *Java Object Instrumentation Environment* (JOIE) [Coh98] is a framework for safe Java bytecode transformation, developed in Duke University. It provides both low-level

and high-level functionality to extend or adapt compiled Java classes. The low-level interface allows manipulation of the bytecode itself, whereas the high-level interface provides methods for inserting new interfaces, fields, methods or whole code slices. JOIE extends Java class loaders with load time transformation. The JOIE class loader allows the installation of bytecode transformers. A transformer can insert or remove bytecode instructions and alter the class file being loaded.

The *Bytecode Instrumentation Tool* (BIT) [Lee97], developed in the University of Washington, is a collection of Java classes that allows users to insert instructions to analysis methods anywhere in the bytecode, so that information can be extracted from the user program while it is being executed. BIT was successfully used to rearrange procedures and to reorganize data stored in Java class files. An application, called *ProfBuilder* [Coo98], was built on BIT, and it allowed for rapid construction of different types of Java profilers.

The Bytecode Engineering Library (BCEL) [Dah01] developed by the Apache Software Foundation is a toolkit for the static analysis and dynamic creation or transformation of Java class files. It enables developers to implement desired features on a high level of abstraction without handling all the internal details of the Java class file format. Unlike other bytecode manipulating tools, the BCEL is intended to be a general purpose tool for bytecode engineering. It gives full control to the developer on a high level of abstraction, and it is not restricted to any particular application area [Dah01]. BCEL is already being

used successfully in several projects such as compilers, optimizers, obfuscators, code generators and analysis tools.

The Java programming assistant (Javassist) [Chi03] is a reflection-based toolkit for developing Java bytecode translators. It is a powerful class library for transforming Java bytecode, and it enables Java programmers to modify a class file before the JVM loads it, and to define a new class at runtime. The main feature of this bytecode operating library is that it allows users to access Java bytecode in the high source code level, instead of in the low bytecode instruction level. Unlike other similar tools, programmers can modify a class file with source-level vocabulary. Users do not have to have detailed knowledge of the Java bytecode and the internal structure of class file. Javassist can compile a fragment of source text on line, for example, just a single statement, and then inserts it into the Java bytecode. This ease of use is a unique feature of Javassist compared to other similar tools.

2.4.3 Execution Tracing

Program tracing is an important program dynamic analysis technique, which is often used in program debugging, testing, monitoring, verification and comprehension. According to the IEEE standard 610.12 [IEE90], a trace is a record of the execution of a computer program, showing sequences of instructions executed, names and values of variables, or both. Its types include execution trace, retrospective trace, subroutine trace, symbolic trace, variable trace. An execution trace is a record of instructions executed

during the execution of a computer program [IEE90]. It often takes the form of a list of code labels encountered as the program is being executed.

According to the definition of dynamic slicing in section 2.3.2, a statement level execution trace is required to compute dynamic slices. In order to obtain a statement level execution trace, users need to produce a record to show the exact statements that were executed and the execution order of the statements during a particular run. The original programs need to be instrumented at the statement level. When a statement is executed, the instrumented instructions are triggered, and generate a record for the statement. It is common that a trace, once generated, is stored in a file. A trace file contains therefore a series of events where one is an execution of a statement.

The main challenge of tracing realistic sized programs is to resolve the size explosion problem. A long execution may generate a huge trace file, and creating these huge traces leads to additional overheads. Furthermore, operating on a huge trace file is inefficient and time-consuming. Hamou-Lhadj and Lethbridge [Ham01] classify the techniques used to reduce the amount of trace information into two categories, trace exploration and trace compression. The first one is concerned with the ability to browse the content of trace efficiently, and the second one directly focuses on reducing the size of the trace by removing or hiding some of its components. They summarize that the data collection techniques, pattern matching, sampling, hiding components and filtering are used in tools

to reduce the size of traces [Ham01].

When tracing a distributed system, users typically obtain separated trace records from various processes. Communication and synchronization information must be captured in order to match trace records from different processes. In message-passing based distributed systems, communication information is traced by recording each *send* and *receive* methods when they are invoked. A message-passing system is first instrumented at the method level to be able to trace *send* and *receive* methods. When the methods are called, the instrumented code will generate communication records. Logical clocks or timestamps are typically used to accomplish synchronization between processes. When a communication record is generated, a timestamp label is attached to the record. According to the timestamps of all communication records, these records can be ordered and connected between a *send* record and its corresponding *receive* record. Since extra instructions have to be instrumented to record the communication and synchronization information in a distributed system, tracing a distributed system is much more difficult than tracing a traditional sequential program. Moreover extra instructions mean that tracing a distributed system will result in an additional overhead compared to tracing a sequential program. It has to be noted, that instrumentation of the source and byte code may modify the program behavior and the non-deterministic behavior of these systems can cause situations where the recorded traces will not correspond to the non-instrumented program executions.

Chapter 3 DYNAMIC PREDICATE SLICING

Rilling, Li and Goswami [Ril02a] first presented a novel predicate-based dynamic slicing for message passing programs. According to their definition, a dynamic predicate slice contains all statements that are relevant to a global predicate during a run of message passing programs. They [LiH04] also extended the dynamic predicated slicing to two kinds of granularity-driven dynamic predicate slicing, coarse-grained and fine-grained dynamic predicate slicing. This chapter presents relevant notations, the predicate-based dynamic slicing, and the algorithm implemented in this thesis.

3.1 Predicate-Based Dynamic Slicing

3.1.1 Global Predicate

A global predicate is defined on variables that are typically located in different processes or channels in a distributed system. It may be viewed as an abstract state of a distributed system, and used to represent properties of a distributed system. Due to the state-explosion problem of executions of distributed systems, engineers meet challenges when maintaining or analyzing distributed systems. Global predicates can be used as filters to abstract behaviors of a distributed system and to capture some requirements of the distributed

system. In this context Rilling et al extended the notion of program slicing to be applicable in identifying these statements (executions) that have an influence on the predicate within the distributed environment.

Li et al [LiH04] give the following formal definition for a global predicate.

- ***Definition**: A global predicate (**F**) is a Boolean valued function defined on either local or global variables distributed among processes and channels.*

In a distributed system, x_i represents a local variable associated with process *i*, and c_{ij} represents a message in the channel from process *i* to process *j*. Some example global predicates given by Rilling et al [Ril02a] are shown below.

- $x_1 \wedge x_2 \wedge ... \wedge x_k$
- $f(x_1, x_2, ..., x_k) < t$ *where f is a linear function.*
- $x_1 \wedge x_2 \wedge (c_{12} = 0)$

3.1.2 Partially Ordered Multi-SET

Lamport [Lam78] introduced the concept of one event happening before another event in a distributed system, and defined a partial ordering of the events. This distributed event model is adopted by Li et al [LiH04] in the predicate-based dynamic slicing. They use the

model to represent an execution of a message passing system. The computation of a predicate slice is based on the distributed event model.

The following are some definitions and notations used in the analysis of execution of a message passing system.

- ***Action:*** an *action* is a statement in a message passing program ***P***. It is an atomic component in the program. According to this notation, the *action* set ***A*** is the set of statements in a program. $A = \{s_i\}$ s_i is a statement in ***P***, $i = 1 \ldots k$.
- ***Event:*** .an *event* is an instance of an action corresponding to the execution of a statement. An event is labeled with its execution order and its associated action.
- ***Output variable***: a variable modified by a statement is an *output variable* of the statement.
- ***Input variable***: a variable referenced by a statement is an *input variable* of the statement.
- ***Data dependency*** orders two statement events if an output variable of one of the statements is an input variable of the other statement.
- ***Control dependency*** orders a block predicate statement that must be evaluated before another statement within the scope of that block can be executed.

- ***Communication dependency*** orders a send event before a receive event in two different processes whenever the message received in the latter comes from the former.

Li et al [LiH04] use the Partially Ordered Multi-SET (POMSET) model to represent an execution of message passing programs. They present the following formal definition of a POMSET.

- ***Definition****: a Partially Ordered Multi-SET (**POMSET**) **<E, A, D, L>** represents a run of message-passing programs such that:*

  E *= set of events,*

  A *= set of actions (program statements),*

  D *= (control/data/communication) dependency ordering among the events,*

  L *= labeling function:* $E \rightarrow A$.

A POMSET is visualized by a labeled directed acyclic graph. After a message passing system is executed, each event in the execution forms a node in the corresponding POMSET. A node is labeled with its corresponding action in ***A***. The dependencies in ***D*** are represented by arrow-headed edges that connect relevant notes where the three different types of dependencies are drawn by using different line styles in the graph.

The dependency ordering relationships in D have the transitivity property. In a POMSET, if e_1 is ordered before e_2 and e_3 is ordered before e_2, e_1 is transitively ordered before e_3. D^* is used to refer to the ***transitive closure*** of the dependencies in D, which order any two events in the POMSET.

Li et al [LiH04] also give a definition of a ***prefix*** of another POMSET as follows.

- ***Definition****: a POMSET* $\mathbf{P_0} = \langle \mathbf{E_0}, \mathbf{A_0}, \mathbf{D_0}, \mathbf{L_0} \rangle$ *is a* ***prefix*** *of another POMSET* $\mathbf{P} = \langle \mathbf{E}, \mathbf{A}, \mathbf{D}, \mathbf{L} \rangle$ *if the following hold:*

  a. *if e is in* $\mathbf{E_0}$ *then e is in* $\mathbf{E}$ *and* $\mathbf{L_0}(e) = \mathbf{L}(e)$

  b. *if e' is in* $\mathbf{E_0}$ *and (e, e') is in* $\mathbf{D^*}$ *then e must also be in* $\mathbf{E_0}$*, and*

  c. *for e, e' in* $\mathbf{E_0}$*, (e, e') is in* $\mathbf{D_0^*}$ *iff (e, e') in* $\mathbf{D^*}$.

Where a POMSET $\boldsymbol{P}$ stands for an execution of a message passing system, a ***prefix*** P_0 of $\boldsymbol{P}$ presents a possible partial execution of the message passing system and consists of exactly those statement events in the partial execution. Therefore, each ***prefix*** represents a distinct global state of the execution. The global state is formed by the values of all program variables that can be determined precisely after executing those events in the ***prefix***.

When one dynamically slices a distributed system, the resulting slice includes a set of events selected in the execution because of their relevant to the slicing criterion. The set of events can be represented by a ***prefix*** of the POSET of the original execution [LiH04].

3.1.3 Dynamic Predicate Slicing

Rilling et al [Ril02a] first introduced a dynamic predicate slice. Given a global predicate as the slicing criterion, a dynamic predicate slice is an executable part of a message-passing program that can reproduce the same behavior with respect to the predicate as the original program. As discussed in section 2.3, traditional slicing criteria focus on those parts of a program that influence variables at a chosen position in the program. A predicate criterion focuses on all states in a run where the predicate might be changed.

The dynamic predicate slicing aims to identify the parts of a message passing program that potentially change the global predicate in the run, consequently modified a program property, rather than a set of variables in traditional slicing. Therefore, Rilling et al [Ril02a] state a dynamic predicate slice as a superset of traditional slices allows for a more general slicing criterion and supports more general comprehension tasks than traditional slices.

Li et al [LiH04] extend the dynamic predicate slice to two granularity-driven dynamic

predicate slices. The original slice is renamed the coarse-grained dynamic predicate slice, which identifies the relevance of a statement with respect to a predicate in a run. They also present a fine grained dynamic predicate slice, which is computed based on the detailed occurrences of a statement that are relevant to a predicate.

Coarse-Grained Dynamic Predicate Slicing

Coarse-Grained dynamic predicate slicing, which corresponds to the basic dynamic predicate slicing presented by Rilling et al [Ril02a], identifies those statements whose executions may affect the state of a global predicate. For a message passing program, a POMSET $P = \langle E, A, D, L \rangle$ presents an execution of the program. The global predicate used as a criterion is denoted by $F(X)$, which is defined on the set X of variables distributed among processes. An event in E is a ***predicate event*** if one of its output variables is in X, i.e. the event can change the state of the predicate. Li et al [LiH04] give the following definition of a coarse-Grained dynamic predicate slice.

- ***Definition**: Given a run **P** of a message-passing program, a **coarse-Grained dynamic predicate slice** is a subset of the statements (actions) of the program that satisfies the following requirements:*

    a. the subset is executable, and

*b. there exists at least one run **P'** of the subset such that for every global state of **P** satisfying the predicate, there exists a distinct global state of P' that also satisfies the predicate, and vice versa.*

The coarse-grained dynamic predicate slicing allows identifying the relevance of a statement with respect to a predicate. Its usefulness is associated to the knowledge of action relevance instead of event relevance. The coarse-grained slicing is adequate for knowing the portion of code relevant to a global predicate. It is limited in knowing more precisely the relevance of events.

Fine-Grained Dynamic Predicate Slicing

In order to achieve more precise relevance of a particular event with respect to a global predicate, Li et al [LiH04] present a fine-grained dynamic predicate slicing for message passing programs. The following notations introduced by Li et al [LiH04] are used to define the fine-grained predicate slicing.

- ***1/0-state*:** A global state is a ***1-state** of a* predicate if the value of the predicate is true in the state; a global state is a ***0-state*** if the value of the predicate is false in that state.
- ***Precede:*** consider two global states ***S*** and ***S'*** associated with two *prefixes* ***P*** and ***P'***

respectively. S ***precedes*** S' *iff* ***P*** is a proper *prefix* of ***P'***.

- ***Next:*** if a proper *prefix* ***P*** of ***P'*** differs from ***P'*** in only one event, then ***S'*** ∈ ***next(S).***
- ***Dominate: S dominates S'*** *iff* both are *1(/0)-state* and there exists a sequence of *1(/0)-state* $S_1 \ldots S_i$ such that $S_1 \in$ next(S), $S_2 \in$ next(S_1)...and $S' \in$ next(S_i).
- ***Dominant state:*** a state ***S*** is a ***dominant state*** if it is not dominated by another state.
- ***Dominant 1/0-state***: a dominant state is a ***dominant 1-state*** if the predicate holds in that state; otherwise, it is a ***dominant 0-state***.

The *prefix* associated with a *dominant state* is a minimal partial execution that just turns the predicate. Removing a single event from the partial execution will change the state of predicate. Therefore, *dominant states* can be used to identify those critical events that switch the global predicate. Based on the preceding notations, a fine-Grained dynamic predicate slice is defined by Li et al [LiH04] as follow.

- ***Definition****: a* ***fine-grained predicate slice*** *is a prefix of the run* ***P*** *that contains all the dominant states of the run. Hence it contains all critical changes of the predicate in the given run.*

Li et al [LiH04] also give a proof that the minimal prefix P* of P that contains all predicate events is a fine-grained predicate slice. The computation of a fine-grained

predicate slice is accomplished through including all predicate events and their dependent events in the dynamic predicate slice. In that way, the resulting slice contains all events that can change and consequently affect the predicate. According to the preceding discussion of the fine-grained predicate slicing, it is apparent that predicate slices are useful to reveal all critical events that switch the predicate.

3.2 Coarse-Grained Dynamic Predicate Slicing Algorithm

The dynamic predicate slicing algorithm implemented in the current research was first presented by Rilling et al [Ril02a]. The algorithm was refined and renamed coarse-grained dynamic predicate slicing contrasting with fine-grained dynamic predicate slicing by Li et al [LiH04]. This algorithm is a kind of forward dynamic slicing algorithms. The notion of dynamic forward computation was originally introduced by Korel and Yalamanchili [Kor94]. The forward slicing algorithm computes dynamic slices for all variables at run-time based on a removable block notion. The dynamic predicate slicing algorithm presented by Rilling et al extends the forward dynamic slicing algorithm to distributed systems. Not only data dependency and control dependency, but also communication dependency is considered in the computation. The algorithm compresses the dynamic composite dependency graph into dependency sets, one per statement, as the execution

unfolds forward. The following section 3.2.1 first presents the notations and assumption of the algorithm and the detailed algorithm is described in section 3.2.2.

3.2.1 Notations and Assumption

- ***Predicate variable:*** A global predicate P = F ($x_1 \wedge x_2 \wedge ... \wedge x_k$) where x_i is a local variable that belongs to process *i*. The variable x_i is called a predicate variable.
- ***Depend(S)*** = Dependency Set of the statement S.
- ***In(S)*** = Set of input variables that are needed in the execution of the statement *S,* i.e. *S* references these variables.
- ***Out(S)*** = Set of output variables modified by the statement *S*.
- ***Recent (v)*** = The statement on that the most recent value of variable *v* is assigned during the run.
- ***Control(S)*** = The control statements, if any, on which statement ***S*** has control dependency.
- ***P(X)*** = Global predicate defined on variable set ***X***.
- ***S(P)*** = Set of statements each of which influences a variable in the global predicate P.
- ***PS(P)*** = Predicate slice corresponding to the global predicate P(*X*).

The assumption of the algorithm is that In(S) = In(S') if S is a receive statement that

receives a message sent by the send statement S' during a run. This assumption implies that communication dependencies are considered during the computation. That also gives a prerequisite for the algorithm, matching a receive event with its corresponding send event.

3.2.2 Algorithm Description

Initialization part:

1 Depend(S) = {S} for all S in the processes;

2 Recent(v) = ∅ for all variables v in the processes;

The main routine:

3 **repeat**

4 Upon executing statement S **do**:

5 Depend(S) = Depend(S) ∪ ($\cup_{\forall v \in In(S)}$ Depend (Recent (v)) ∪ Depend (Control(S));

6 **for** each v ∈ Out(S) **do**: Recent(v) = S;

7 **until** program terminates;

8 S(P) = { S | ∃ v ∈ Out(S) and v ∈ X};

9 PS(P) = $\cup_{S \in S(P)}$ Depend(S);

Figure 3.2.1 Coarse-Grained Dynamic Predicate Slicing Algorithm

The Coarse-Grained dynamic predicate slicing algorithm shown in Figure 3.2.1 is executed for each process being analyzed. The algorithm computes the dependency set for each statement as the execution proceeds forward. The final dynamic predicate slice is computed by taking the union of the dependencies of all statements that appear as predicate

events.

In the initialization part, the first step (step 1) is to set up Depend(S) for all statements in the processes of message passing programs. The initial value of Depend(S) includes only S itself. Recent (v) for all variables in the processes is initialized with Ø (step 2).

The main routine includes two parts, one executes repeatedly for each executing statement, a.k.a. event, until program terminates (step 3 to step 7). The other part as final computation gets the predicate slice for a given predicate (step 8 and step 9). When a statement is being executed, Depend(S) is computed by considering two kinds of dependencies, data dependencies and control dependencies. In step 5, Depend (Recent (v)) for each v belongs to In(S) stands for all statements on which S data depends, and Depend (Control(S)) gets all statements have control dependency relationship with S. Moreover, if statement S modifies a variable v, i.e. v belongs out(S), the Recent(S) needs to set up to statement S (step 6). Step 8 finds out all statement influences the predicate P, i.e. if statement S modifies a predicate variable of P, put S into S(P). The final step (step 9) computes the resulted slice for predicate P, which is the union of all Depend(S) for each statement S belongs to S(P).

Chapter 4 Contributions

In this chapter, the contributions of this research are presented. Section 4.1 introduces both the motivations of this research and dynamic predicate slicing. The hypotheses of the research are presented in section 4.2. The research goals are listed in section 4.3. The system overview is presented in the last section of this chapter.

4.1 Motivations

▪ *Slicing for Distributed Systems*

Distributed systems have been applied so widely that almost all applications and systems, nowadays are utilizing some distributed elements. Therefore, the comprehension of distributed system has become critical in the software comprehension field. Program slicing has been established in the last decade as a program analysis technique, and has been used broadly in the program comprehension of sequential systems. There have been many different program slicing algorithms presented in literature. Most of the algorithms, focusing on traditional sequential programs, aimed at their specified application, and have succeeded in different program analysis tasks.

Even though there are some program slicing algorithms already presented for concurrent

or distributed systems, they have limitations or deficiencies with respect to their slicing precision, the detail of the analysis level, and their language paradigm support. Moreover, there is a need for providing a more powerful and efficient slicing algorithm for distributed systems in order to enhance the distributed program comprehension. Distributed programs, unlike sequential programs, run on different processes or even different machines and one has to deal with the required communication among these processes. For that reason, slicing distributed programs is much more difficult than slicing sequential programs. The information from several processes and the communication between processes must be taken into account in a distributed program slicing algorithm. In addition, computation of distributed program slices is usually implemented either concurrently in different processes, or simulative in different threads in order to achieve efficiency.

▪ *Predicate Slicing Criteria*

The slicing criterion as originally defined by Weiser [Wei82] is a 2-tuple <x, V>, where x is a statement in a program, P, and V is a subset of variables in P. A static program slice based on the slicing criterion <x, V> is a subset of program statements that preserves the same behavior of the original program at the program point p with respect to the program variables in V. The criterion defined for dynamic slicing by Korel and Laski [Kor88]

includes a set of variables V, input I, and point P in the execution history. These criteria of the static and dynamic slicing were first introduced for traditional sequential programs. They worked efficiently for helping to understand sequential programs.

Kamka and Kranjina [Kam95] applied the dynamic criterion to distributed programs. They defined a distributed slicing criterion as a non-empty set of selected statements in processes with a test case. Cheng [Che93] extended also the notion of slicing to concurrent programs. He redefines the static and dynamic slicing criteria for a concurrent program. The criterion he defines is almost the same as criteria for sequential programs. The only difference is that the dynamic criterion for concurrent programs includes not only variables V and input I, but also an execution history of concurrent programs, H. There are also many other extensions of either static or dynamic slicing criteria for distributed programs in the literature [XuB05]. Most of them are very similar with the criteria defined for sequential programs. The major characteristic of distributed systems, a global state machine, is not taken into account.

Global predicates are used to describe the abstract state of a distributed system in order to help developers to obtain design requirements or analysis, and understand the behaviors of a distributed system. The use of global predicates in program slicing comes rather

naturally. Predicates can be useful filters to abstract the behaviors of distributed programs. A global predicate can be defined on program variables distributed among processes and channels. It is often related to some requirement or suspected error properties of the program. Unlike traditional slicing criteria that focus only on those parts of the program that influence a variable at a chosen position in the program, a predicate slicing focuses on all states of an execution in which the predicate might be changed. The motivation of the dynamic predicate slice is to identify these program parts that might potentially modify a program property, rather than modify a set of variables.

▪ *Capturing Interdependency*

The majority of existing slicing algorithms in literature are based on some type of dependencies between statements to compute either static or dynamic [Tip95] [XuB05]. There are two kinds of dependencies that exist in sequential programs, data dependence and control dependence. They are defined in terms of a Program Dependency Graph (PDG) [Tip95]. Li et al [LiH04] formally redefine the data dependency by using the notations of output variables and input variables of a statement. A variable that is modified by a statement is an output variable of the statement, and a variable that is referenced (used) by a statement is an input variable of the statement. If an output variable of one statement is an

input variable of the other statement, there is a data dependency between them. The control dependency is between a predicate statement and the statements within the scope of that block. Data dependencies and control dependencies have been widely used to draw different graphs, such as the Control Flow Graph, the Program Dependence Graph, and the System Dependence Graph. These graphs represent a program, and are used to compute various slices.

However, for distributed programs, because of the synchronization and communication between processes, there are not only data dependency and control dependency in each process, but also interdependency among these processes themselves. For computing a slice for a distributed system, data dependency and control dependency alone are not enough. In order to achieve a precise algorithm for slicing distributed programs, the interdependency between processes must be obtained as well. Li et al defined [LiH04] the communication dependency for message passing programs. A communication dependency orders a "*send*" event before a "*receive*" event in two different processes where the message received in the latter comes from the former. Communication dependencies are detected at runtime.

▪ *Tracing Program Executions*

A dynamic slicing criterion specifies an input, for which the program will have a unique execution. Different from static slices, a dynamic slice is constructed with respect to the particular program execution based on the input specified by the dynamic slicing criterion. Dynamic slice resolves some imprecision of static approaches with respect to control flow. Therefore, dynamic slicing can compute smaller program slice, than static slicing. Moreover, statements included in a dynamic slice have direct bearing on the program behavior occurring during that particular program execution and captures that particular behavior.

However, distributed programs often make non-deterministic decisions because of the communication delay between distributed processes running on different machines. The communication delay is time dependent. Consequently, the order in which messages sent and received are time dependent, and may vary from one execution to the next. It follows that repeated executions of a program on the same input may result in different program paths. However, in the case of dynamic slicing of distributed programs for debugging, testing, or comprehending purposes, programmers are interested in the exact execution that exhibits an anomalous behavior. Dynamic (predicate) slicing is based on the assumption

that execution traces that capture the particular program behavior are available and therefore allow for a replay and re-analysis of that particular execution scenario.

4.2 Hypotheses

4.2.1 Dynamic Predicate Slicing for Distributed Systems

Global predicates have been widely used for the comprehension of distributed systems, representing and capturing program properties and abstract behaviors of systems. While distributed systems become more and more popular and wide-spread, a rather straight forward extension to program slicing has been to apply it also to distributed programs and their comprehension. In order to reflect the essential characteristic of a distributed system, one can apply global predicates in conjunction with program slicing to enhance the comprehension of distributed program [Ril02a]. The result is a new type of slicing for distributed programs, dynamic predicate slicing. A predicated slice is computed based on a predicate criterion that is defined on variables distributed among processes and even in channels. The dynamic predicated slicing computes all relevant statements that will affect the value of the global predicate (see for more details in the section 3.1.3).

Hypothesis 1: Coarse-grained dynamic predicate slicing can captures global

requirements or suspected error properties of a distributed program to support the comprehension of distributed systems

NULL Hypothesis 1: Dynamic predicate slicing can not capture properties of a distributed system and therefore does not support the comprehension of these systems.

Unlike traditional sequential programs, the states of a distributed system are described by the variables distributed among processes and even in channels, and the behaviors of a distributed system are depended on the global states instead of the local variables values. Capturing the change of the global states of a distributed system plays a key role in understanding distributed programs or debugging an error happening in an execution of the distributed programs. A global predicate is a Boolean valued function defined on either local or global variables distributed among process and channels. The global predicate reflects the global states of a distributed system, and captures the principal properties and behaviors of the distributed system. Unlike traditional slicing criteria that focus only on those parts of the program that influence a variable of interest at a specific position in the program, a predicate slice contains all states in the program run where the global predicate changes its value, thus the predicate slicing can provide a better support to understand distributed programs than traditional criteria slicing. Therefore, the hypothesis 1 will hold

and the null hypothesis 1 can be rejected.

4.2.2 Capturing Program Executions and Communications

Dynamic slicing is defined on user-specified inputs, and a dynamic slice is based on a particular execution history of programs. Thus tracing of program executions is required to computing a dynamic slice. Because of the inter-dependency and non-determinability of distributed systems, tracing each execution of processes alone is not enough to obtain an accurate execution history of distributed programs. The communication between processes has to be captured properly also. The instrumentation technology either in source codes or in bytecodes has been applied widely in the program analysis field in order to obtain runtime information.

Hypothesis 2: Instrumentation either at the source code or the bytecode level can provide the ability to trace the execution of programs and capture the communication between processes; therefore, the dynamic predicated slicing can be implemented by using the instrumentation to achieve the dynamic analysis.

Null-Hypothesis 2: Executions and the communication between different processes can not be captured and therefore the dynamic slicing algorithm cannot be implemented.

With the support of instrumentation probes can be inserted into programs at either the source codes or the bytecode level. These probes will collect the runtime information, and produce the necessary record for the later analysis purpose. As we mention in section 2.5, the source code instrumentation benefits the statement level analysis, and the execution history for dynamic slicing is a record of the statement that are run exactly in the execution. So, the source code instrumentation can fulfill the tracing requirement. To capture the communication information, we need to focus on method level analysis. The bytecode instrumentation is more efficient and has less impact on original programs than the source code instrumentation. It works sufficiently for the method level analysis. The tracer inserted through the bytecode instrumentation can monitor the occurrence of the "*send*" and "*receive*" methods in message-passing programs. The tracer collects the related dynamic information, and then produces a record for the communication. Therefore hypothesis 2 will hold and the null-hypothesis 2 can be rejected.

4.3 Research Goals

The presented research and implementation of the Dynamic Predicate Slicing is a part of work of the **C**omprehension **O**f **N**et-**CE**ntered **P**rograms and **T**echniques (CONCEPT) project, which is a continuation of a previous project, Montreal Object Oriented Slicing

Environment (MOOSE) [Ril01]. The CONCEPT project addresses the current and future challenges in the comprehension of large and distributed systems by providing programmers with novel comprehension techniques [Ril02b]. These techniques are based on a variety of source code analysis, visualization, and application approaches. The project is also currently in the process of expanding current approaches for distributed and client server based systems [Ril02b].

The more specific goals of the present research are the following:

(1) To study the application of dynamic program slicing for the comprehension of distributed systems,

(2) Review and select an appropriate instrumentation technique(s) for dynamic program analysis, and

(3) To implement the presented coarse-grained dynamic predicate slicing algorithm.

(4) Provide an initial experimental analysis of the algorithm with respect to its time and space complexity.

The programs analyzed by the predicate slicing algorithm are coded in Java. The Java language is very popular in the network programming environment. The distributed system paradigm we focused on is the message-passing pattern, which is the most fundamental

paradigm for distributed applications, and which also can be said to be synonymous to a distributed program, as we indicated in the section 2.2, the definition of a distributed system given by Coulouris et al [Cou01] is adopted in this research. We assumed that the "*send*" function is non-blocked and the "*receive*" function is blocked in the message-passing paradigm in order to reduce the complexity of message-passing programs. We adopted the Java Agent Development Framework (JADE) as the message-passing platform. Agents built on JADE communicate through "*send*" and "*receive*" message methods.

4.4 System Overview

4.4.1 CONCEPT Environment

The CONCEPT environment provides a framework to facilitate and enhance both visualization and source code analysis techniques [Ril02b]. It allows programmers to apply various cognitive models during the comprehension of large distributed systems to generate different levels of visual abstraction. The program slicing technique is used to reduce the amount of information displayed to the users and further enhance these visualization techniques. The CONCEPT environment is illustrated in Figure 4.4.1.

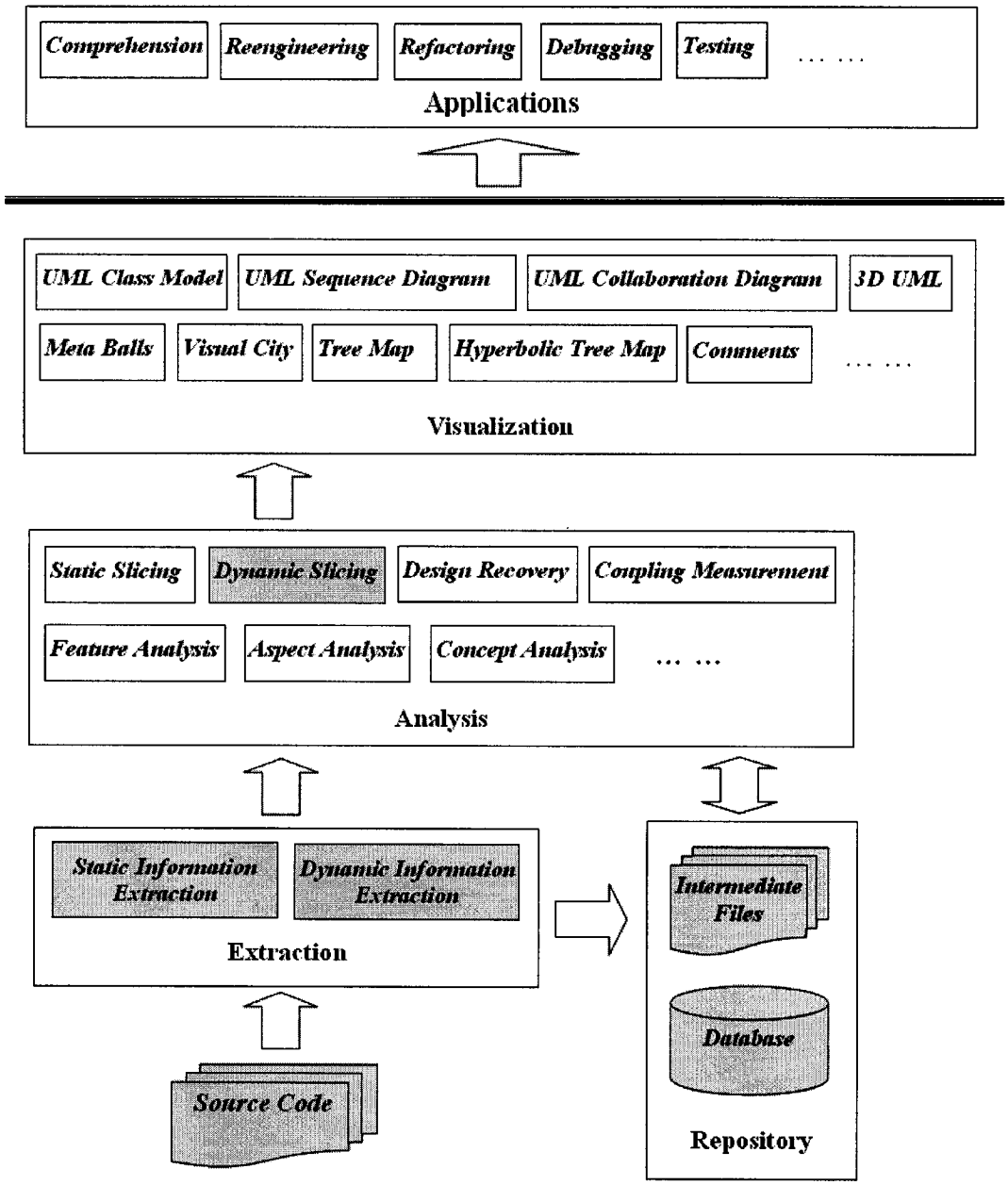

Figure 4.4.1 CONCEPT Environment

The CONCEPT environment assists users in achieving various tasks such as program comprehension, reengineering, refactoring, debugging, and testing. The knowledge about programs is first extracted from source code, which always reflects the current application

correctly, and is stored into the repository, either a database or intermediate files. The original rough information is processed by various calculation or analysis algorithms. The result data may be stored into the repository too, or provided to be visualized directly. The extracted and processed knowledge and data are visualized by using a visualization model suitable to various purposes.

4.4.2 Dynamic Predicate Slicing Overview

The presented research on predicate slicing, as a part of work of the CONCEPT project, focuses on dynamically slicing distributed systems (i.e. message-passing programs in this research) based on novel criteria, predicate criteria.

The gray components in Figure 4.4.1 are involved in this research, with a focus on dynamic predicate slicing, which falls in the category of dynamic slicing extended to message passing programs. The source code component are message passing programs, from which we extract static information through a parser provided by Zhang [Zha03], a member of the CONCEPT project, and collect dynamic information through the instrumentation mechanism described before. Both extracted information (static and dynamic) as well as resulting predicate slices are stored into the central repository.

In Figure 4.4.2 the details of the dynamic predicate slicing architectures are illustrated.

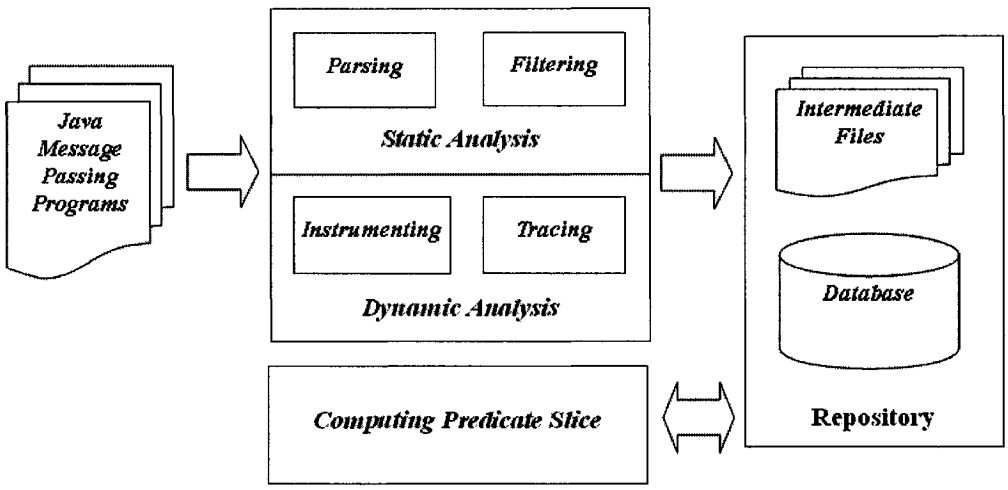

Figure 4.4.2 Dynamic Predicate Slicing Overview

As described earlier in section 2.4, an execution of message passing programs is represented by the POMSET model, and the computing of predicate slices is based on the POMSET model. The POMSET model represents data dependency, control dependency and communication dependency between events, which are occurrences of statements in an execution. The data dependency and control dependency are calculated from the static information obtained in the static analysis step. The static analysis component is similar to the corresponding component of the other slicing algorithms provided by the CONCEPT project. The component reuses a Java parser, which is developed by another member of CONCEPT group, with a specified filtering component to obtain necessary static information for the dynamic predicate slicing. The static information extracted by the static

analysis is stored into the repository, which is an intermediate file in the current research.

As defined in section 3.1.2, the dynamic part of a POMSET model includes the set of events in an execution of a message passing program (i.e. occurrences of statements in the execution), and communication dependency between processes. The dynamic analysis component achieves the runtime information collecting task. In the dynamic analysis, the instrumentation technique is used for tracing the execution of a message passing program and collecting communication information. In order to obtain execution trace and communication information, a message passing program is first instrumented, after then the instrumented program is recompiled, and is executed. The instrumented code will collects dynamic information about the execution of original program, and output the information into repository.

The dynamic predicate slice is finally computed by the computing component. The component retrieves the static information and dynamic information from the repository, and forms an in-memory model for the message passing programs. The computation then is performed on the in-memory model. The result slice based on a specified predicate is shown to the user, and stored into repository.

Chapter 5

IMPLEMENTATION AND EXPERIMENTAL ANALYSIS

5.1 Message Passing Sample Program

5.1.1 Java Agent Development Platform (JADE)

The Java agent development framework (JADE) developed by TILAB provides a software development environment for the distributed multi-agent applications based on the peer-to-peer communication architecture [Bel00] [Bel03]. JADE is compliant with FIPA (the Foundation for Intelligent Physical Agents) specifications, which are a collection of standards, intended to promote the interoperation of heterogeneous agents and the services they can represent. JADE is fully implemented in Java language.

The JADE agent platform provides flexible and efficient message-passing communication architecture. The communication between agents is performed through message passing, where FIPA ACL (Agent Communication Language) is the language used to represent messages. JADE creates and manages a queue of incoming ACL messages for each agent. A queue is private to each agent. Agents can access their queues via a combination of several modes: blocking, polling, timeout [Bel00]. In the current

research, we use the JADE communication architecture to simulate a message passing platform. Message passing sample programs used in the current research, are implemented based on the JADE environment.

5.1.2 Message Passing Sample Program for JADE

For this research, the sample program, presented by Li et al [LiH04] is adopted as a basic message passing example, and is re-implemented using JADE. The original sample program is shown in Figure 5.1.1.

```
Process0:
        int threshold0, r = 0;
(1)     input(threshold0);
(2)     send(threshold0, 1);            //send value of threshold0 to process whose pid = 1
(3)     send(threshold0, 2);
(4)     receive(r);                     //receive new value of r from any process
(5)     output('The largest element below threshold is ', r);

Process1:                                       Process2:
        int threshold1, n, x, a[10], p= 0, sum1= 0;     int threshold2, m, y, b[10], q= 0, sum2= 0;
(6)     input(n, a);                            (16)    input(m, b);
(7)     receive(threshold1);                    (17)    receive(threshold2);
(8)     while a[n] > threshold1     {           (18)    while b[n] > threshold2     {
(9)       { sum1 := sum1 +a[n];                 (19)      { sum2 := sum2 + b[m];
(10)        n : = n – 1;      }       }         (20)        m := m – 1;     }       }
(11)    p := a[n];                              (21)    q := b[m];
(12)    send(sum1, 2);    //send to process2    (22)    send(sum2, 1);
(13)    send(p, 0);                             (23)    send(q, 0);
(14)    receive(x, 2);    //receive from process2 (24)  receive(y, 1);
(15)    sum1 := sum1 + x;                       (25)    sum2 := sum2 + y;
```

Figure 5.1.1 Message Passing Sample Program

This message passing example is implemented by defining each process as a JADE

agent that runs on the agent platform across different machines. A JADE agent is a class extending from the *jade.core.Agent* class. The *Agent* class represents a common base class for user defined classes, and accomplishes basic interactions with the agent platform and a basic set of methods (e.g. send/receive messages). When an agent is created, it is given an identifier by the JADE platform, which is a globally unique name, and then its *setup*() method is executed. The *setup*() method is the point where the agent's activity starts, and users should add at least one behavior to the agent in the body of this method.

The functionality of each process is implemented as one behavior of the corresponding agent. The behavior is defined as a subclass of *jade.core.behaviours.SimpleBehaviour* class. An instance of the behavior is added to its agent in the *setup*() method by invoking the *addBehaviours(Behaviour)* method. The main routine of each process is located in the method *Behaviour.action()* of the behavior class. Finally, a behavior calls the *Agent.deDelete()* method to stop the execution of an agent when the task has completed. The *Agent.takeDown*() method is executed when the agent is about to be deleted.

5.2 Static Analysis

The static analysis carries out the extraction of static information from message passing programs. The output information is filtered and formatted to match the required input for

the dynamic predicate slicing algorithm. The workflow for the static analysis step is illustrated in Figure 5.2.1.

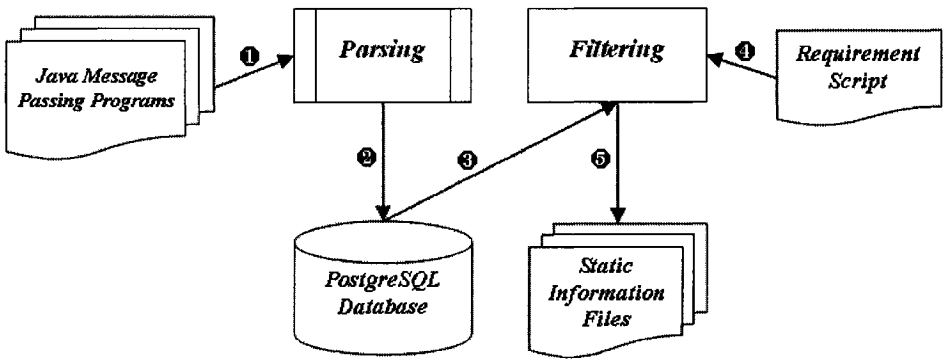

Figure 5.2.1 Parsing and Filtering

5.2.1 Parsing and Database

The CONCEPT environment supports Java source code parsing through a modified version of the standard Java compiler, *javac* from the Java Development Kit (JDK). The modified parser was developed by Zhang [Zha03], a member of the CONCEPT project. The *javac* is a fully implemented java language compiler. It not only performs lexical analysis, but also semantic analysis, which facilitates the identification of the relationships between the entities in source code And its source code is available from Sun Micro's website. These are the reasons why the *javac* is selected as a parser for the program static analysis purpose in the CONCEPT environment.

The modified Java compiler extends the original compiler from the standard JDK. It reads Java source code, and compiles it into bytecode files. During the compilation, after a symbol table is created by the original compiler, the javac extension will be invoked. It transfers the symbol table into a predefined Abstract Syntax Tree (AST) structures for all compiled classes. These ASTs are partially normalized and stored within a PostgreSQL database [Zha03].

PostgreSQL is adopted as the database management system in the CONCEPT environment. PostgreSQL is a SQL compliant, open source object-relational database management system, which is released under the BSD (Berkeley Software Distribution) license [Pos05]. The most important thing for the CONCEPT project is that PostgreSQL supports multiple queries and views.

As illustrated in Figure 5.2.1, Java message passing programs first are compiled by the parser, the modified *javac* (❶). Then the AST Trees are generated for all compiled classes from the symbol tables of each program. After that, the AST trees are normalized, and stored into the database (❷). Figure 5.2.2 shows an example of the result AST.

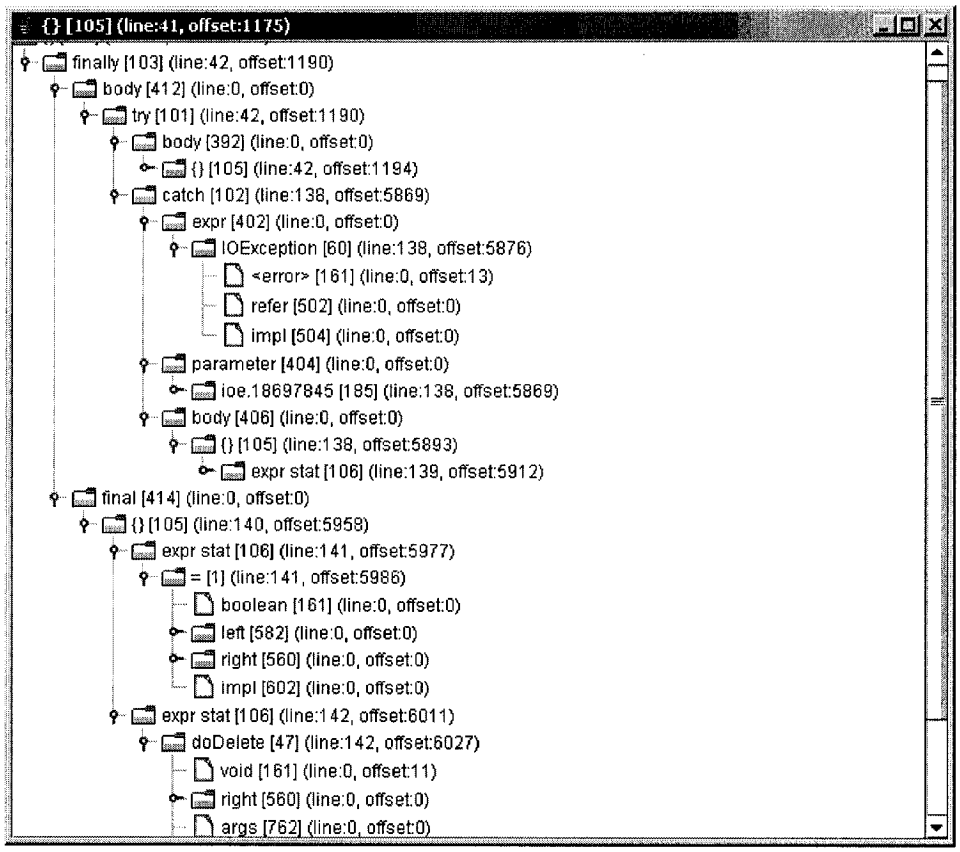

Figure 5.2.2 Example of AST

5.2.2 Filtering and Formatting

The parsing process generates the static knowledge about programs and their structures. Even though the resulting data has been normalized, it still contains much more information than we need for dynamic predicate slicing. For example, the dynamic predicate slicing may not be performed on all classes; instead it may only focus on specific classes or methods. Moreover, in order to improve the performance of the dynamic predicate slicing, the static information needs to be simplified and formatted to suit the

need of the dynamic predicate slicing in the current research.

The CONCEPT environment provides an easy way to access the static information stored in the database, through an Object Oriented model [Zha03]. The static data in the database is first retrieved into memory. Then an object-oriented model is formed so that the complexity of extracting information can be reduced. The OO Model also provides APIs to traverse the AST and basically analyze the raw static information. The Object Oriented Model of an AST is showed in Figure 5.2.3.

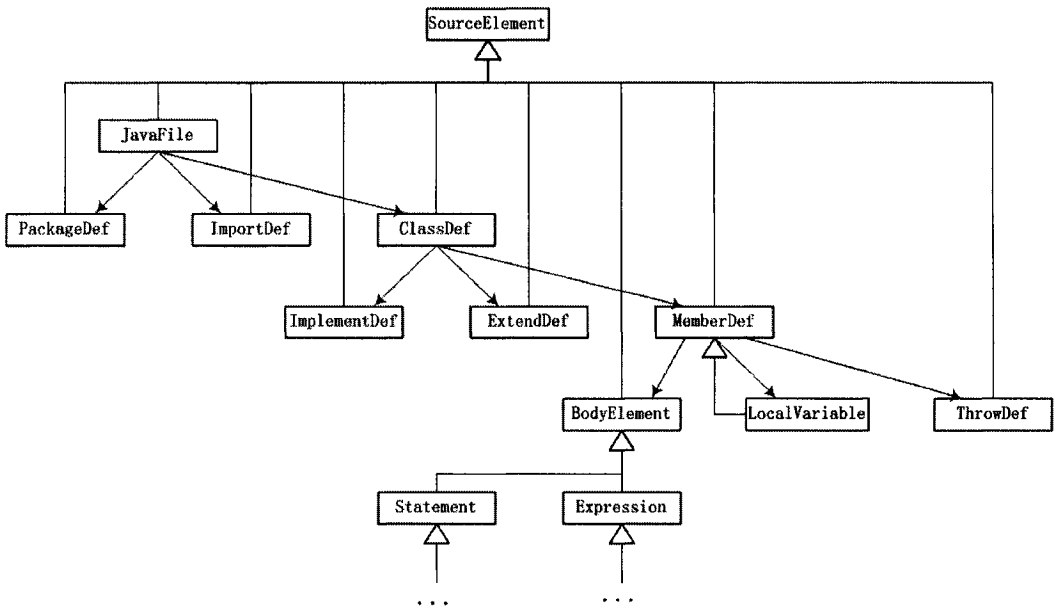

Figure 5.2.3 Object Oriented Model of the AST

From the database, each Java file in a project is loaded into a *JavaFile* class. In a *JavaFile* class, *PackageDef* class represents the package definition; *ImportDef* class

indicates the import list; *ClassDef* class represents the definition of a class in the java file. The *ClassDef* class contains *ImplementDef* classes, which refers to the implement interface list, an *ExtendDef* class, which represents the super class, and *MemberDef* classes, which may be attributes, constructors, methods, or inner classes. If *a MemberDef* class refers to a constructor or a method, the *MemberDef* class may contain *BodyElement* classes, in which *Statement* classes and Expression classes may be defined. The filtering process operates the OO model of the AST trees to obtain necessary static information for the dynamic predicate slicing.

According to the dynamic predicate slicing algorithm described in section 3.4, control and data dependencies are computed through the input variables, output variables, and control statements for each statements. Thus, the filtering process extracts this information from the raw static information obtained through source code parsing. For each statement in message passing programs, if a variable is modified by the statement, the variable is put into the *Out* set of the statement; if a variable is referenced by the statement, the variable is inserted into its *In* set. All control statements are put into the *Control* set of the statement.

As shown in Figure 5.2.1, the filtering process retrieves raw static information from the database (❸), extracts additional data according to the requirement script (❹) that indicates

the focused classes and methods, formats the data, and then generates static an information file (❺). The XML is adopted as the format of the static information file. Figure 5.2.4 demonstrates the result of the static analysis, the partial static information XML file, which indicates input variables, output variables, and control statements for statements in the message passing programs.

```
<?xml version="1.0" encoding="UTF-8" ?>
- <StaticInformation>
 - <Project Name="MessagePassing">
  - <JavaFile Name="D:\concept\MessagePassing\src\Process0.java">
   - <ControlDependencies>
     ...
     ...
     - <Line Number="91">
       <ControlLine Number="40" />
       <ControlLine Number="41" />
       <ControlLine Number="86" />
       <ControlLine Number="88" />
       <ControlLine Number="90" />
      </Line>
     ...
     ...
     </ControlDependencies>
   - <DataDependecies>
     ...
     ...
     - <Line Number="50">
      - <in>
        <Variable Line="49" Name="rr" Offset="1336" />
       </in>
      - <out>
        <Variable Line="50" Name="streamTokenizer" Offset="1427"
/>
       </out>
     ...
     ...
     </DataDependecies>
   </JavaFile>
  + <JavaFile Name="D:\concept\MessagePassing\src\Process1.java">
  + <JavaFile Name="D:\concept\MessagePassing\src\Process2.java">
  </Project>
</StaticInformation>
```

Figure 5.2.4 Static Information File

5.3 Dynamic Analysis

Through the dynamic analysis, we obtain runtime knowledge of the message passing programs. For dynamic predicate slicing runtime information that has to collect during the execution includes execution trace and communication trace. Source code instrumentation technique is used to accomplish the execution trace collection. The Java bytecode instrumentation is adopted to fulfill the communication trace gathering. The workflow of the dynamic analysis step is illustrated in Figure 5.3.1.

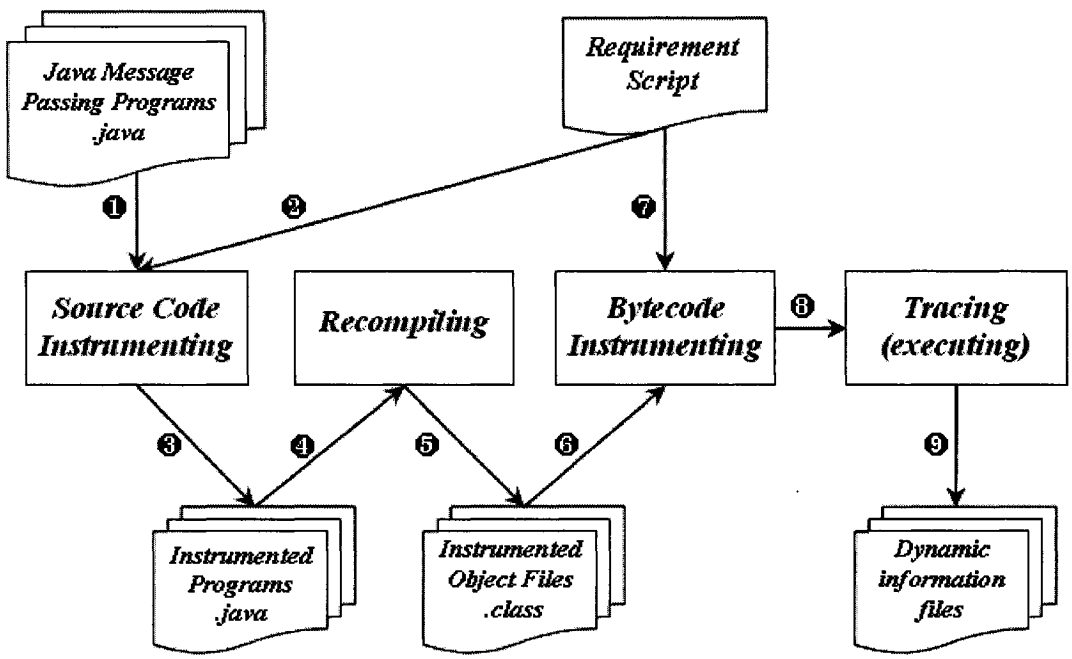

Figure 5.3.1 Instrumenting and Tracing

5.3.1 Source Code Instrumentation

The source code level instrumentation inserts extra source code artifacts, which we called tracing statements into the original source code of the message passing programs. Such tracing statements are inserted for each statement in the programs. When a statement is being executed, the corresponding tracing statement is activated, and a trace record for the executing statement is generated and stored into the trace file.

The whole source code instrumentation process as illustrated in Figure 5.3.1 is performed by the following steps: the Java source code of the message passing programs is first instrumented (❶), according to the requirement script (❷), and then the instrumented programs are generated (❸); the instrumented source code is then recompiled (❹) into *.class* files. (❺)

The instrumentation toolkits, "*query and instr*", introduced in section 2.5, are used in this research for the source code instrumentation. In these toolkits, *query* provides the function to parse Java source code into an internal tree structure. The *instr* package, which is based on the query package, operates on the tree representation of a program, and adds information to the tree. The *instr* package originally supplies several instrumentation functions [McC05]. *instr.add_counts* performs test coverage, which counts how many

times each statement is executed. *instr.instr_trace* provides statement level instrumentation used to trace individual source lines. *instr.instr_meth* instruments each methods at the point of entry to generate an record for each call of the method.

The *instr.instr_trace* instruments the Java source code at each statement line so that the instrumented code generates traces that are outputted to the console or a file at run-time. Each trace includes the Java file name, the line number, and the source code in this line. This statement instrumentation is accomplished by means of the following steps. First, the source file is read and parsed into a tree. Then the parse tree is annotated with the instrumentation, used to trace each executed statement. The annotated tree is written back to a file. The file contains original source code and some additions. The instrumented file then needs to be re-compiled. The annotations have to be removed before execution in order to show correct original source code. Otherwise, the displayed source lines by the instrumented code will have the annotations in them. The last step is to execute the compile instrumented program to show the execution traces.

Figure 5.3.2, Figure 5.3.3, and Figure 5.3.4 demonstrate the process of the instrumentation, and show respectively a sample program, instrumented program and the execution trace generated by the instrumented program. In the Figure 5.3.3, the code

between /*_I*/ and /*I_*/ is the instrumented code that is inserted by the *instr_trace*. In this example, the *instr_trace* inserts a method call of *instr.InstrUtil.showLine()*, which is a static method of the class *InstrUtil*. The method has two arguments, the first one is the file name, and the second one is the line number.

```
1  public class Loop {
2    public static void main(String args[]){
3      int n = 1;
4      for (int i = 1; i <= 10; i++)
5        n = n * i;
6      System.out.println(n);
7    }
8  }
```

Figure 5.3.2 Sample Program for Instrumentation

```
public class Loop {
 public static void main(String args[]){/*_I*/instr.InstrUtil.showLine("Loop.java", 2);/*I_*/
  /*_I*/instr.InstrUtil.showLine("Loop.java", 3);/*I_*/int n = 1;
  /*_I*/{instr.InstrUtil.showLine("Loop.java", 4);/*I_*/for (int i = 1; i <= 10; i++)
 /*_I*/{instr.InstrUtil.showLine("Loop.java", 5);/*I_*/n = n * i;/*_I*/}/*I_*//*_I*/}/*I_*/
  /*_I*/{instr.InstrUtil.showLine("Loop.java", 6);/*I_*/System.out.println(n);/*_I*/}/*I_*/
 }
}
```

Figure 5.3.3 Instrumented Sample Program

```
[Loop.java 2]    public static void main(String args[]){
[Loop.java 3]      int n = 1;
[Loop.java 4]      for (int i = 1; i <= 10; i++)
[Loop.java 5]        n = n * i;
[Loop.java 5]        n = n * i;
[Loop.java 5]        n = n * i;
[Loop.java 5]        n = n * i;
[Loop.java 5]        n = n * i;
[Loop.java 5]        n = n * i;
[Loop.java 5]        n = n * i;
[Loop.java 5]        n = n * i;
[Loop.java 5]        n = n * i;
[Loop.java 5]        n = n * i;
[Loop.java 6]      System.out.println(n);
3628800
```

Figure 5.3.4 Execution Trace of Sample Program

Although the *instr_trace* can generate execution traces, there are still some problems existing for our dynamic predicate slicing. First, the execution trace is only shown on the console, and the trace includes the source code for each line which is not required for the predicate slicing. So we need to build our own customized tracing instrumentation.

First, we design a *Tracer* class, which is used in both source code level and bytecode level instrumentations. The class *Trace* is shown in Figure 5.3.5. Its method *recordTrace()* is used in the source code level instrumentation in order to generate our customized trace, where each trace record consists of the name of a executing class and the line number of a executing statement.

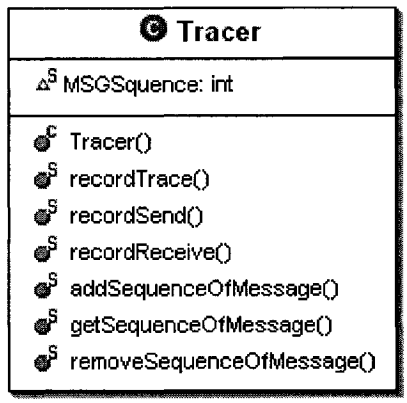

Figure 5.3.5 Tracer Class

The *instr_trace* class is modified in order to instrument our *tracer.recorderTrace()* method instead of the original *instr.InstrUtil.showLine()* method for each statement line. Figure 5.3.6 shows an excerpt from the instrumented message passing programs. The *Tracer.recordTrace(),* which is in the package, *concept.predicateSlcing.dynamicAnalysis*, is instrumented for each statement in the message passing programs.

```
...
public void action() {/*_I*/concept.predicateSlicing.dynamicAnlysis.Tracer.recordTrace("Process0.java", 40);/*I_*/
...
/*_I*/concept.predicateSlicing.dynamicAnlysis.Tracer.recordTrace("Process0.java", 45);/*I_*/int threshold0 = 0, r = 0;
...
/*_I*/{concept.predicateSlicing.dynamicAnlysis.Tracer.recordTrace("Process0.java", 80);/*I_*/send(msg);/*_I*/}/*I_*/
...
```

Figure 5.3.6 Excerpt of Instrumented Message Passing Program

5.3.2 Bytecode Instrumentation

Bytecode instrumentation provides efficient support for Java runtime dynamic analysis, especially for method-level dynamic information collection. As a result, bytecode level instrumentation is used to trace the communication between processes in the message passing programs. The communication is performed by *send*() and *receive*() methods. During the bytecode instrumentation, a tracing statements are inserted for each *send()* and *receive()* methods. When the message passing program is being executed and the methods are invoked, the inserted tracer is activated to generate a communication record for each *send()* or *receive()*. The bytecode instrumentation is performed during the loading time of a Java class. As shown in Figure 5.3.1, the recompiled bytecode is instrumented (❻) just before being loaded into the Java virtual machine. The requirement script indicates the object of the instrumentation (❼). After then, the instrumented bytecode (❽) is ready to be executed in order to generate execution and communication traces.

For the bytecode level instrumentation, we use a Java programming agent to access bytecode when a class is loaded into the JVM. The Java 1.5 has the new package *java.lang.instrument*, which provides services that allow java programming agents to instrument programs running on the JVM. A Java programming agent is a special class that

implements a public static *premain* method similar in principle to the *main* application entry point.

public static void premain(String agentArgs, Instrumentation inst);

An agent is launched by indicating the agent class through the *-javaagent* option when the JVM is started. After the JVM is initialized, each *premain* method will be called in the order the agents were specified, then the real application *main* method will be called. Figure 5.3.7 illustrated the instrumentation process through an agent. The dashed line is the common Java class loading process, and the real line shows the instrumenting process through a java programming agent before a class is loaded into the JVM.

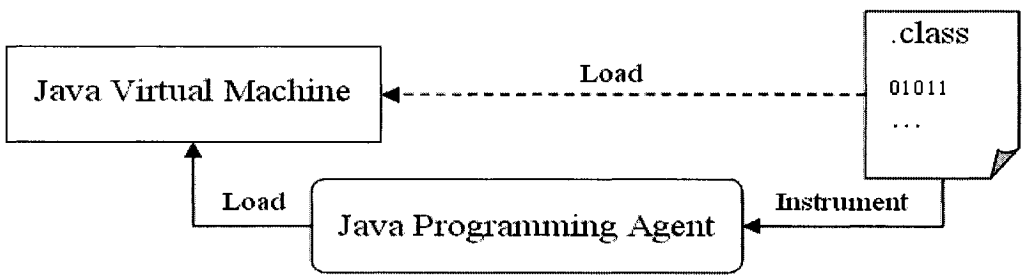

Figure 5.3.7 Java Programming Agent Instrumentation Process

The first parameter of *premain()* method, *agetnArgs,* is used to pass agent options. Via the second parameter, *inst*, an agent is passed an instance of the *Instrumentation* interface, which provides the services needed to instrument Java programming language code. The

addTransformer(ClassFileTransformer transformer) method of the interface is used to register a transformer. All class definitions will be transformed by the transformer. We define a transformer class, *ClassFileTransformerImpl,* to accomplish the bytecode instrumentation.

Through the Java agent class, we get access to all class definitions, that correspond to sequences of bytes in the Java class file format as defined by the JVM specification. The Javassist, introduced in section 2.5, is used to manipulate these bytecode sequences. It provides a class library for modifying the java bytecode in a high level, the source code level instead of the low bytecode instruction level.

In the Javassist, the class *javassist.CtClass* is an abstract representation of a Java class in bytecode format. An object of this class is a handle for operating the bytecode of a class. The object must be obtained from a *ClassPool* object, which is a container of *CtClass* objects, in order to keep the consistency. From a *CtClass* object, we can get *CtField* objects, which stand for the attributes of the class, and *CtMethod* objects, which represent the methods defined in the class, including constructors.

The *CtMethod* class is a subclass of the *CtBehavior* class, which is an abstract super class of the *CtMethod* and *CtConstructor*. The *CtBehavior* provides the methods to carry

out the instrumentation, including *insertBefore(), insertAt(), insertAfter(), instrument(), setBody()* etc. The instrument*()* is used to fulfill our instrumentation task.

The *instrument()* method takes an instance of the *ExprEditor* class as a parameter. The *ExprEditor* class is a translator of method bodies, which actually do the modification. In order to customize how to modify a method body, we need to define a subclass of the *ExprEditor* class. For the tracing the communication purpose, the defined *ExprEditor* subclass replaces all *send()* and *receive()* method calls.

As shown in Figure 5.3.5, the *Tracer* class is also used for the bytecode level instrumentation. The *Tracer.recordSend()* and *Tracer.recordReceive() methods* trace the communication between processes. The two methods are respectively instrumented into programs after *send()* and *receive()* method calls. When the *send()* or *receive()* method is invoked completely, the corresponding trace method will be activated, and generate a communication trace, which is saved into a trace file for each process.

In order to match a *send()* method with the *receive()* method that receive the message sent by the *send()* from a different process, the *Tracer* class creates a sequence number for each message that will be sent, and attaches a tag to the message. The tag shows the sender and the sequence number of the message. Then the message will be sent with the tag. After

a message is sent, the sender, the line number where the *send()* method is called, and the message sequence number will be passed to the tracer. The tracer then generates a send record, which consists of the sender, the line number, and the message sequence number.

When a process receives a message, the tracer detaches the tag of the message, reads the information, and then generates a receive record, which consists of the receiver, the line number, the sender and the sequence number of the received message. The sequence number and the sender identify each message, and work together as a sort of logical timestamp for the communication. At the sender side, the sender and message sequence number are recorded. On the receiver side, the tracer also gets the sender and the message sequence number for the message it receives. In this way, a *receive()* method call may be matched with the corresponding *send()* method call from the sending process. We obtain the communication independency between processes in the message passing programs.

5.3.3 Execution and Run-time Information Collection

After the message passing programs are instrumented, as shown in Figure 5.3.1, the last step of dynamic analysis is to execute the instrumented code (❸). While the instrumented programs are being executed, the dynamic information is generated by the inserted code

and stored into the dynamic information files (❾). The execution trace and communication trace will be saved in the same trace file for each process. The tracer instrumented in the source code level generates the execution trace for each statement, and the tracer instrumented in the bytecode level generates the communication trace. Figure 5.3.8 shows the part of dynamic information file for the process0.

```
...
<Trace><Runner>Process0.java</Runner><LineNumber>78</LineNumber></Trace>
<Trace><Runner>Process0.java</Runner><LineNumber>80</LineNumber></Trace>
<Send><Sender>process0@workstationsong:1099/JADE</Sender><LineNumber>80</LineNumber>
      <MSGSequence>1</MSGSequence></Send>
<Trace><Runner>Process0.java</Runner><LineNumber>81</LineNumber></Trace>
<Trace><Runner>Process0.java</Runner><LineNumber>85</LineNumber></Trace>

...
<Trace><Runner>Process0.java</Runner><LineNumber>100</LineNumber></Trace>
<Trace><Runner>Process0.java</Runner><LineNumber>102</LineNumber></Trace>
<Send><Sender>process0@workstationsong:1099/JADE</Sender><LineNumber>102</LineNumber>
      <MSGSequence>2</MSGSequence></Send>
<Trace><Runner>Process0.java</Runner><LineNumber>103</LineNumber></Trace>

...
<Trace><Runner>Process0.java</Runner><LineNumber>109</LineNumber></Trace>
<Receive><Receiver>process0@workstationsong:1099/JADE</Receiver><LineNumber>109</LineNumber>
        <Sender>process1@workstationsong:1099/JADE</Sender><MSGSequence>2</MSGSequence></Receive>
<Trace><Runner>Process0.java</Runner><LineNumber>110</LineNumber></Trace>
<Trace><Runner>Process0.java</Runner><LineNumber>111</LineNumber></Trace>

...
```

Figure 5.3.8 Dynamic Information File

In the trace file, a record is surrounded by <Trace> and </Trace> to indicate begin and end of the record in the execution trace. The record contains information about the process, filename and the statement number of an executed statement. These execution traces are

sorted in the execution order. A preceding execution trace indicates the statement line is executed before all statement lines shown in the succeeding traces.

There are two kinds of communication traces, the send record and the receive record. A send record is surrounded by <Send> and </Send>, and a receive record is surrounded by <Receive> and </Receive>. In Figure 5.3.8, the first send record indicates that process0 sends the first message on line 80, and the second send record indicates that process0 send the second message at line 103. There is only one *receive* record in the file, which indicates that process0 receives a message at line 109, the message is sent by process1 and the message sequence number is 2.

For the message passing sample program, we get three trace files for the three processes. The trace files include execution traces and communication traces. For a receive trace, we can easily follow back to the corresponding *send()* method. From the receive record, we know where the message comes from. In the sender process, we find out which *send()* method sends the message with the same sequence number of the received message. We consequently match the *send()* method with the *receive()* method. According to the communication traces in the three files, we can build a communication dependency graph for the execution of the message passing program, as shown in Figure 5.3.9.

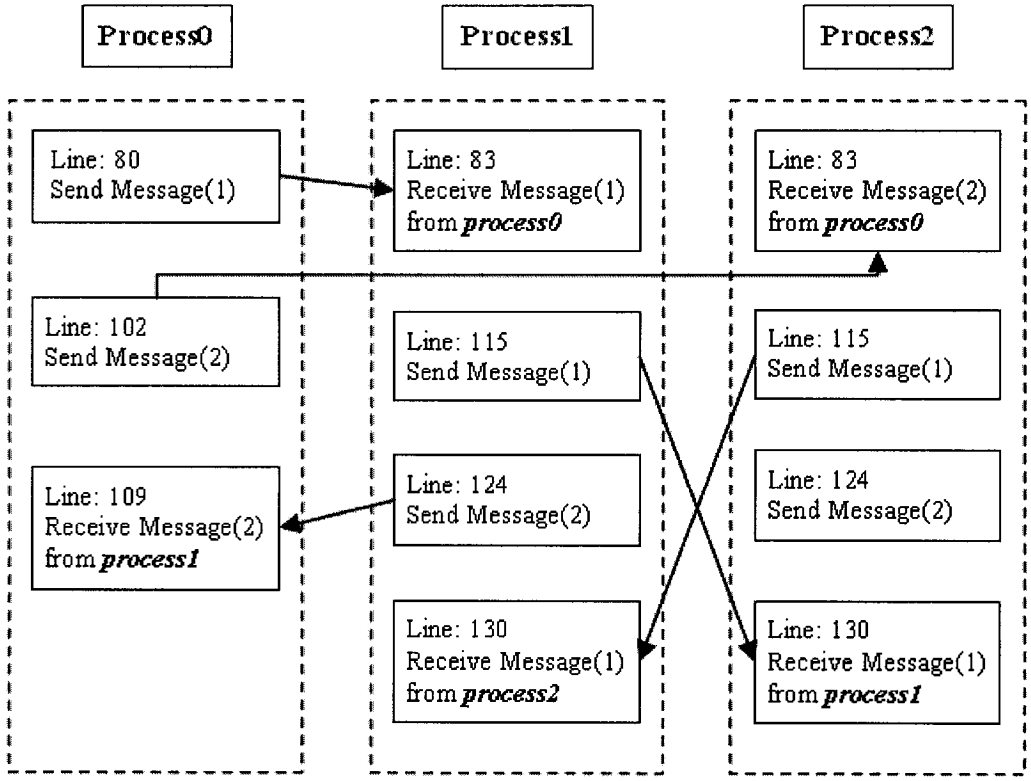

Figure 5.3.9 Communication Dependency

5.4 Computation of Coarse-Grained Predicate Slices

Finally, the predicate slices are computed based on the static information obtained through parsing and filtering, and the dynamic information generated by the instrumented code. The workflow of the computing predicate slice is illustrated in Figure 5.4.1.

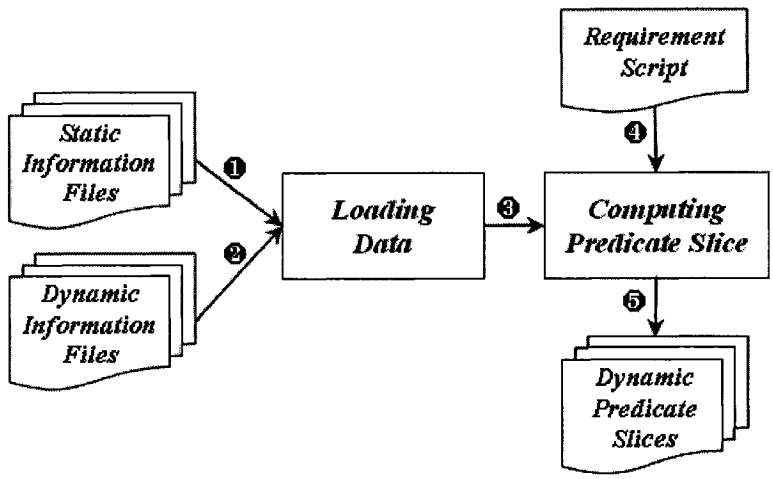

Figure 5.4.1 Computing Dynamic Predicate Slice

In order to enhance computation performance, static information (❶) and dynamic information (❷) are loaded into the computer's memory first. The static and dynamic information are merged, and a model representing the message passing programs is created in the memory. The later computation then will perform totally in the memory. After the static and dynamic information has been loaded into the memory, dynamic predicate slices are computed based on the in-memory mode of the message passing programs (❸). The requirement script indicates the process of computing (❹), and the result predicate slices are stored into files (❺), which may be used by further analysis and visualization.

5.4.1 Message Passing Program In-Memory Model

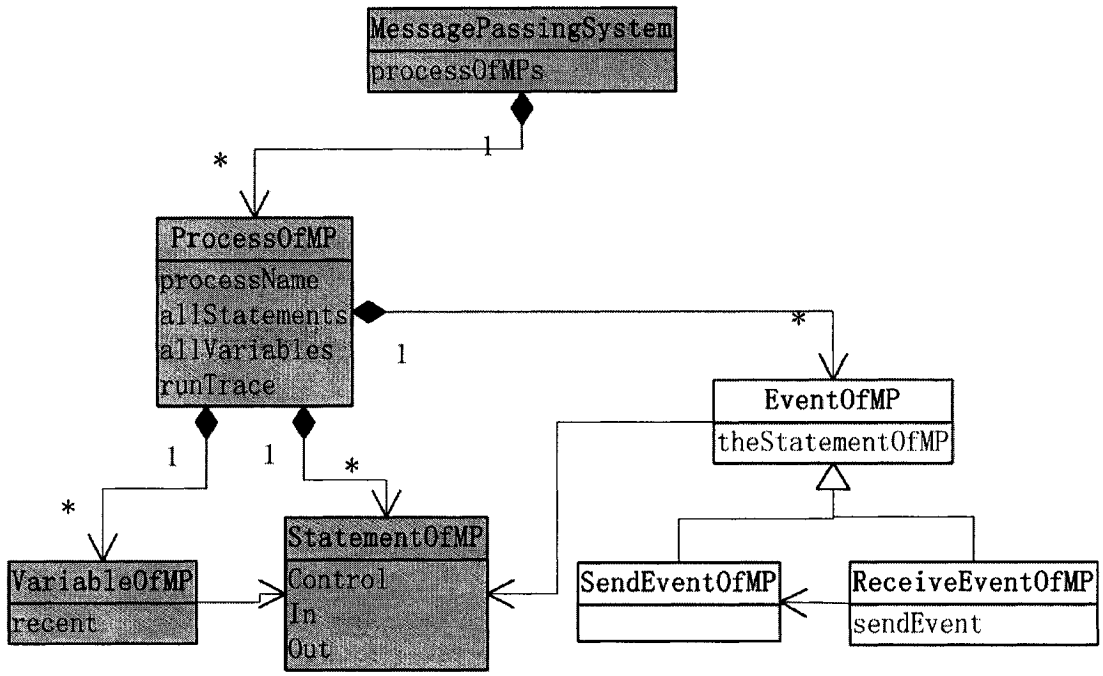

Figure 5.4.2 Message Passing Program In-Memory Model

Figure 5.4.2 shows the class diagram of a message passing program in-memory model. The model is used to present a message passing system in-memory in order to enhance the performance of the computation of dynamic predicate slices. It integrates both static and dynamic information, where, static information is loaded into the gray classes in the figure, and the other classes are used to loaded dynamic information.

A message passing system is modeled by the class *MessagePassingSystem,* which may contain several processes represented by the field, *processOfMPs.* A process in a message

passing system is modeled by the class *processOfMP,* which has fields, *processName, allStatememements, allVariables, and runTrace.* The *allStatements* and *allVariables* fields represent the static information. The *runTrace* filed represents the dynamic information.

Each statement in a process is represented by the class *StatementOfMP*. The class has fileds *Control*, *In*, and *Out*. All static information is loaded into these fields. The *Control* field is a vector of *StatementOfMp*. All control statements are put into the vector. The *In* and *Out* fields both are vectors of *VariableOfMp*. The *In* vector contains all input variables of the statement, and the *Out* vector contains all output variables.

The class *VariableOfMp* stands for a variable in the message passing system. In order to identify each variable, the class has fields, *name*, *line*, and *offset*. These three fields work together to resolve the name duplication problem. The class *VariableOfMP* has a *recent* filed, which is an instance of the class *StatementOfMp*. The *recent* field is used to record the statement, on which the variable was assigned the most recent value. The value of this field is changed dynamically during the computation of the algorithm according to the execution trace of the message passing system and the *out* set of the preceding statements.

Each process has the *runTrace* filed, which is a vector of *EventOfMP* instances. The *runTrace* represents an execution of the process in the message passing system. The

EventOfMp class stands for an event in the message passing system. As defined in the section 3.2, an event is an occurrence of a statement in the message passing system. The *EventOfMp* class has the *eventNumber* and *theStatementOfMp* fields. The *theStatemOfMp* field links the event to its corresponding statement.

The communication dependency in the message passing system is represented by the *SendEventOfMp* and *ReceiveEventOfMp* classes in the model. When trace files are being loaded, if a send record is found, the event for the occurrence of the statement is instanced as a *SendEventOfMp* instead of an *EventOfMp*. In the same way, if a receive record is met, the event is instanced as a *ReceiveEventOfMp*. After all trace records are loaded, a *ReceiveEventOfMp* object will be connected to its corresponding *SendEventOfMp* object. In this way, the communication dependencies are represented by the model.

5.4.2 Dynamic Predicate Slicing

After the static information and dynamic information are collected and loaded into the memory, the *PredicateSlicer* class implements the computation of a dynamic predicate slice for a message passing system. The class diagram of *PredicateSlicer* is shown in Figure 5.4.3.

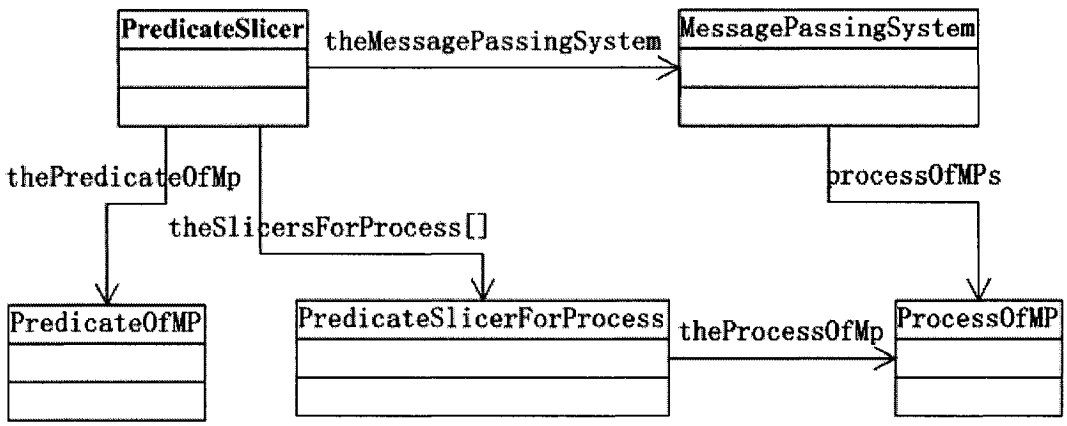

Figure 5.4.3 Class Diagram of Dynamic Predicate Slicing

Based on the predicate slicing algorithm described in section 3.2.2, the main routine of the algorithm is executed for each process upon its execution trace concurrently. Thus, the algorithm naturally is implemented by a concurrent multi-thread computation model.

The *PredicateSlicer* class, which implements the *Runnable* interface, is responsible for the computation of the whole message passing system. The *PredicateSlicer* starts the main thread for the computation of the algorithm. In the main thread, the initialization part is executed first, and then, the main thread starts one thread for one process. After the completion of the computations for all processes, the main thread computes the result predicate slice according to the result of computations in each process, and the provided predicate criterion.

When a *PredicateSlicer* is initialized, the message passing system is specified first. The

static and dynamic information is loaded into memory, and an in-memory model is created for the message passing system. In a next step, the *PredicateSlicer* class will generate the same number of instances of the *PredicateSlicerForProcess* class as the number of processes in the message passing system.

Each PredicateSlicerForProcess instance implements the computation on a process in the message passing system. Like the *PredicateSlicer* class, the *PredicateSlicerForProcess* class implements the *runnable* interface too. It executes the main routine in the algorithm for each process. An instance of the class has a link to its corresponding process. As described in section 5.4.1, when the message passing system model is formed, the execution trace of a process is loaded into the model. The *PredicateSlicerForProcess* will compute the slice for every statement based on its execution trace. The result slice will be put in the *Slice* field of each *StatementOfMP* object, which stands for a statement in the message passing system.

The dynamic predicate criterion is represented by the *PredicateOfMP* class. When a *PredicateSlicerOfMP* object is initialized, the criterion is specified through an instance of the *PredicateOfMP* class. The class has a *predicateString* field, which stores the original predicate as a string. Its *variableInPredicate* field stores all variables in the predicate. The

final computation of a predicate slice is based on the class.

According to the algorithm, the final slice is the union of all *Depend(S)* for each statement *S* belonging to *S(P)*, in which the *Depend(S)* is the *Slice* field of each statement and *S(P)* is the set of statements whose output variables contain a predicate variables (i.e. a variable in the field *variableInPredicate*).

5.4.3 Predicate Slicing Results

Based on the implementation described before, an initial experiment was conducted for the sample message passing program shown in Figure 5.1.1. As elaborated in the last section, the static information and the dynamic information are first loaded into the memory. The final computation is completed on the in-memory model of the message passing system. We give several various global predicates of the message passing system as dynamic slicing criteria, and compute the resulting slices for each predicate. The resulting slices are the following:

- For Predicate criteria only include variable, *threthod0*:

The final result of Predicate Slice for *threshold0* is:

< process0(37) process0(31) process0(39) process0(41) process0(40) process0(32)

process0(33) process0(36) >

- For Predicate criteria only include variable, *threthod1*:

The final result of Predicate Slice for threshold1 is:
< process1(33) process0(31) process1(32) process1(70) process0(32) process0(48) process1(71) process1(34) process1(66) process1(65) process0(66) >

- For Predicate criteria only include variable, *threthod2*:

The final result of Predicate Slice for threshold2 is:
< process0(31) process2(66) process2(32) process2(72) process0(32) process0(48) process2(65) process2(33) process2(34) process0(85) process2(70) >

- For Predicate criteria include variables, *threthod1* and *threshold2*

The final result of Predicate Slice for threshold1, threshold2 is:
< process1(33) process1(32) process0(31) process2(66) process1(70) process2(32) process2(72) process0(32) process2(65) process1(71) process2(33) process2(34) process1(34) process1(65) process0(85) process0(48) process1(66) process0(66) process2(70) >

5.5 Experimental Analysis

5.5.1 Static Information Collection

As described in section 5.2, the static information collection is accomplished through

two steps. The first step is parsing source codes and loading the resulting AST structures into a database. The second step is extracting necessary information from the database and forming static information files. Parsing and Loading source codes into database is implemented by Zhang [Zha03], and he also gave a basic measurement of this step in his research. Thus, we just measure the overhead of retrieval information from the database and the overhead of extraction of static information.

The experiments are performed on a PC with Pentium IV 2.40G CPU, 1 Gigabytes of main memory, and 120 Gigabytes hard disk. The operating system is Windows 2000 SP4. The DBMS we used is PostgreSQL 8.0, which is installed locally. The Java environment is the standard JDK 1.4.2 from Sun Microsystems.

The programs used in the experiments include an elevator program, which simulates an elevator in a building and are used in CONCEPT as a basic testing unit, the sample message passing programs described in section 5.1.2, and the *concept.java* package that is part of CONCEPT project.

Table 5.5.1 shows the overhead of static information collection. From the table, it is evident that the overhead of static information collection is directly related to program size. Analyzing a larger program consumes longer time than analyzing a small one. The

overhead of information retrieval is relatively low, compared to the overhead of information extraction.

Table 5.5.1 Overhead Related to Static Information Collection

	elevator	Message Passing Sample Programs	concept
Lines of Code (LOC)	261	370	14,768
Number of Classes	4	6	201
Number of Methods	15	12	2,621
Overhead of Information Retrieval	656ms	813ms	4,141ms
Overhead of Information Extraction	12,422ms	59,453ms	178,031ms
Number of Exceptions Thrown During Extraction	52	282	772

When we extract static information from in-memory AST models loaded from the database, the AST structures is traversed entirely in order to find out all dependencies between program elements. However, only AST models for the classes defined within the program is stored into the database when a program is parsed and its corresponding AST structures are stored into the database. All imported classes used in the program are not stored into the database. Consequently, when we traverse AST models, if an imported class is met, a *ClassNotFoundException* exception is thrown. Table 5.5.1 also gives the statistic numbers of exceptions thrown during extraction. Those exceptions cause the relative huge overhead of information extraction.

5.5.2 Instrumentation and Tracing

First, we use the same programs used in the previous section to measure the overhead caused by the instrumentation at the source code level. Table 5.5.2 shows this overhead and provides a comparison between the original source code file size and the instrumented source file size.

Table 5.5.2 Overhead related to Source Code Instrumentation

	elevator	Message Passing Sample Programs	concept
Lines of Code (LOC)	261	370	14,768
Number of Files	3	3	132
Overhead of Instrumentation	250ms	282ms	1360ms
Size of Original Source Files	9.64KB	16.6KB	222KB
Size of Instrumented Files	24.7KB	38.7KB	581KB

The time overhead used to instrument source codes depends on the size of the programs. When the size of the program increases, the time consumption for instrumentation will increase accordingly. During the source code instrumentation, an extra instruction is inserted for each statement in the programs. Thus, the size of instrumented files increases according to the LOC of the original files, and the size of instrumented files is approximately twice the size of the original files.

We use two programs to measure the overhead caused by tracing program executions One is an enhanced elevator example, in which, an elevator will run automatically many times, and for each operation, a random number will direct which floor it will go. In order to test different execution lengths, we introduced several loop iterations that correspond to several elevator moving cycles. Table 5.5.3 shows the overhead of execution tracing based on the elevator's executions.

Table 5.5.3 Overhead related to Execution Tracing - Elevator

	Elevator			
Loop Iterations	10	20	50	100
Original Programs Execution Time	31ms	62ms	172ms	422ms
Instrumented Programs Execution Time	2,375ms	4,250ms	9,672ms	19,578ms
Size of Execution (Number of Events in the Execution)	7,447	14,454	34,284	66,952
Size of Trace Files	555KB	970KB	2.31MB	4.52MB

From the table, it can be observed that the time consumption and the size of the trace files are directly related to the size of the execution.

The other example program is Ping-Pong, which is a typical program used to measure the round trip delay of a message between two message-passing processes [Mor02]. In our implementation of Ping-Pong the program is based on JADE, the message consists of a string where the length of the string varies as a function 2^x, where x is from 0 to 15. The

smallest message is a string containing one char (1 byte) and the largest message is a string containing 32,768 chars (32 Kilobyte). In the Ping-Pong program used in this section, each message is sent 2^k times between process0 and process1, where k is from 0 to 4, in order to produce execution traces in different size. Table 5.5.4 shows the statistics of the overhead of execution tracing based on the executions of the Ping-Pong program.

Table 5.5.4 Execution Overhead Tracing – Ping-Pong

	Ping-Pong			
Loop Iterations	16 * 2	16 * 4	16 * 8	16 * 16
Original Program Execution Time	2,906ms	3,484ms	4,125ms	5,390ms
Instrumented Program Execution Time (bytecode level)	3,019ms	3,609ms	4,531ms	6,031ms
Instrumented Program Execution Time (source code level)	20,969ms	21,719ms	22,578ms	24,281ms
Instrumented Program Execution Time (both levels)	21,375ms	22,204ms	23,329ms	25,121ms
Size of Execution (Number of Events in the Execution)	66,027	66,0243	66,666	67,512
Communication Events	72	144	288	576
Size of Trace Files	4.60MB	4.62MB	4.66MB	4.74MB

The above table shows the loop iterations, the execution time of the original programs, the execution time for the instrumented versions of the programs at the bytecode, source code and both levels respectively. The execution size is expressed by the number of events in a particular execution, which includes execution trace events collected by source code

instrumentation, and communication trace events captured by the bytecode instrumentation. The number of communication events is also provided. From the above table, we can see that the total overhead caused by tracing the program executions can mainly be attributed to the execution of the extra codes added through source code instrumentation.

5.5.3 Communication Overhead

In this section, we use the Ping-Pong program to measure the additional communication overhead caused by the instrumentation. We modified the original Ping-Pong program as follows. The message consists of a string where the length, L, of the string varies as a function:

$$L = \begin{cases} 2^x \quad where \quad x = 0...13 \\ L + 2^{11} \quad if \quad 2^{13} < L < 2^{15} \\ L + 2^{12} \quad if \quad 2^{15} < L < 2^{16} \end{cases}$$

Using this function, we send 23 different size messages. The smallest message is a string containing one char (one byte) and the largest message is a string containing 57,344 chars (57 Kilobytes). Each message is sent a total of 64 times between two processes. The process0 records the round-trip time for each message and produces an average round-trip

time for each string.

Table 5.5.5 Communication Overhead Caused by Instrumentation

Message Size	Round-Trip Time of the Message (ms)			
	Run on a single computer		Run on two computers	
	Original Program	Instrumented Program	Original Program	Instrumented Program
1	5	7	7	8
2	4	6	6	8
4	4	6	6	8
8	4	6	6	8
16	4	6	6	7
32	4	6	6	7
64	4	5	6	8
128	4	5	6	7
258	4	6	6	7
512	5	8	7	9
1024	4	5	6	7
2048	4	6	6	8
4096	5	7	7	10
8192	7	11	10	15
12288	11	16	16	20
16384	16	24	22	31
20480	21	30	27	38
24576	32	38	37	45
28672	38	47	50	54
32768	42	53	55	66
40960	53	66	68	80
49152	75	95	85	112
57344	96	106	126	129

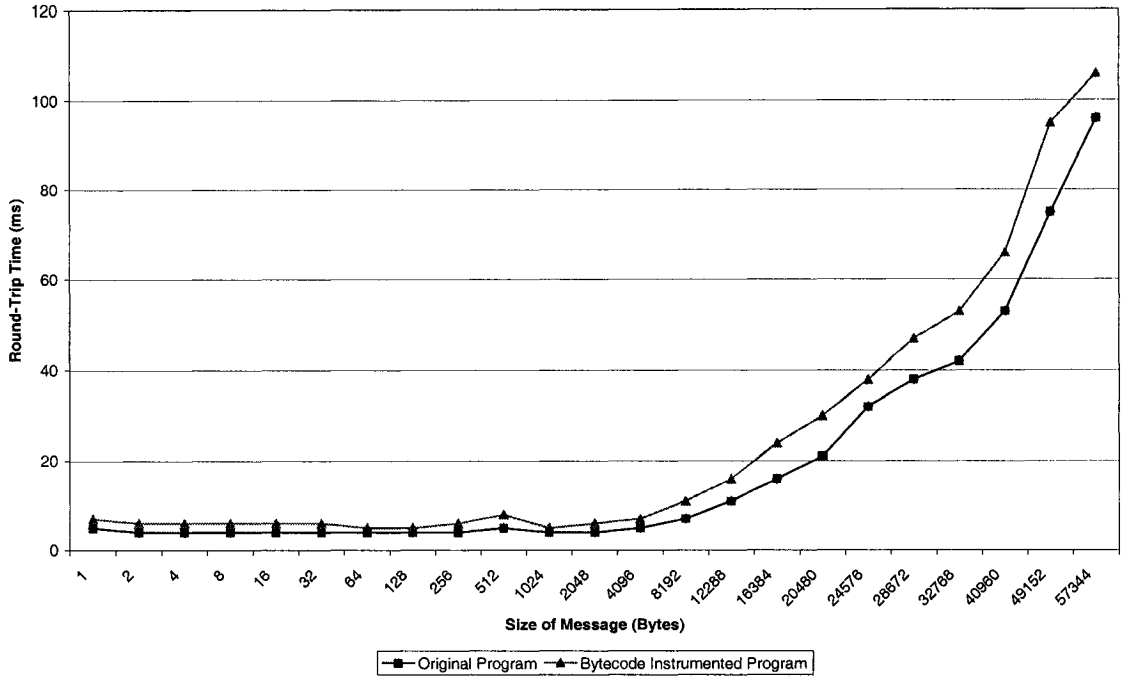

Figure 5.5.1 Communication Overhead - Ping-Pong executed on a single computer

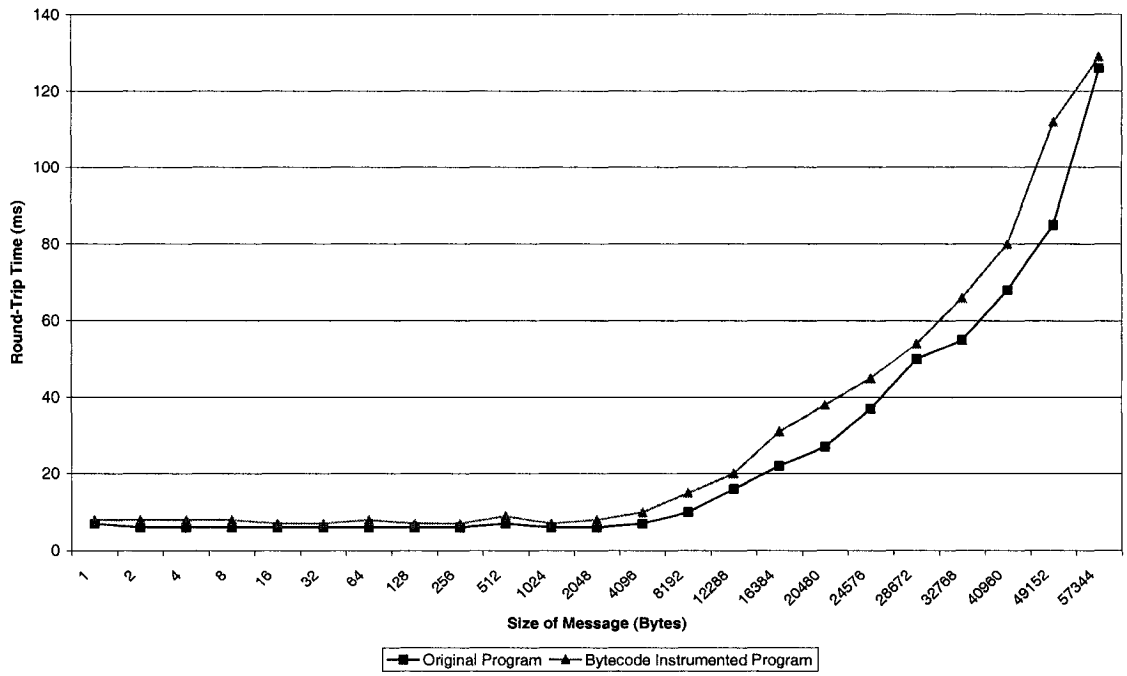

Figure 5.5.2 Communication Overhead - Ping-Pong executed on two computers

Table 5.5.5 shows the observed communication overhead during the execution of the Ping-Pong program (instrumented and non-instrumented). Figure 5.5.1 and 5.5.2 illustrate these differences of the round-trip communication times between the original program and the instrumented program. In Figure 5.5.1 the Ping-Pong program runs on a single computer whereas in Figure 5.5.2 the program runs on two computers connected via a fast Ethernet network. The message round-trip time between two processes running on a single computer is shorter than the round-trip time between two processes running on two separated computers. The comparison from the table and the two figures shows that the bytecode-level instrumentation causes a slight communication overhead compared to the non-instrumented version. The typical communication overhead is in the range of 5-10%.

5.5.4 Predicate Slicing Performance

According to the algorithm presented in section 3.2, the overhead for the computation of a dynamic predicate slice depends on the size of the recorded execution trace because the main routine of the algorithm is executed upon every executing statement, i.e. each event in a trace. In order to measure the performance of the predicate slicing algorithm, we compute predicate slices based on the executions of the message-passing sample program with

different iterations. Table 5.5.6 shows the time consumed to computer predicate slices.

Table 5.5.6 Performance of Predicate Slicing Algorithm

	Message-Passing Sample Program					
Iterations	1	5	10	20	50	100
Number of Events	250	620	1,089	2,017	4,890	9,459
Time Consumed for Loading Static and Dynamic Information	390ms	407ms	422ms	438ms	469ms	516ms
Time Consumed for Computing Dynamic Predicate Slice	169ms	673ms	1,132ms	2,200ms	4,899ms	8,146ms

As we described earlier in section 5.4, the predicate slicing algorithm requires that all static and dynamic information about the message passing program and its execution are first load into the memory to form an in-memory model, and then the computation is performed based on the in-memory model. In table 5.5.6 the overhead for loading this static and dynamic information is shown. One can observe an almost linear correlation between trace size and loading time; the time consumed for computing dynamic predicate slices increases linearly according to the number of events recorded in the executions, where the computation is performed by several threads concurrently in memory, and each thread computes slices for one process in the message passing program.

Chapter 6 CONCLUSIONS AND FUTURE WORKS

As distributed systems are ubiquitous today, the comprehension of distributed systems becomes critical to the development or maintenance of a system involving some distributed components. Due to the complexity of distributed systems, comprehending a distributed system is more challenging than comprehending a traditional single-process sequential system. Slicing as a well-known program decomposition technique has been widely used in the software comprehension field and adopted to support the comprehension of distributed systems.

Global predicates are used as filters to abstract behaviors of a distributed system, or to capture some requirements of a distributed system. Rilling et al [Ril02a] first applied the notion of global predicate into criteria of dynamic slicing for distributed message passing programs, and presented a novel predicate-based dynamic slicing. They also presented two kinds of granularity-driven dynamic predicate slicing [LiH04]. Traditional program slicing techniques focus on those parts of a program that influence a variable at a chosen position. Predicate slicing focus on all states of an execution of a message-passing program, in which the predicate might be changed. Therefore, the predicate slicing allows for a more

general slicing criterion and supports more general comprehension tasks.

In this thesis, we presented an instrumentation-based approach to implement the coarse grained dynamic predicate slicing algorithm presented by Rilling et al [Ril02a] [LiH04]. The approach is based on the instrumentation at both source code and bytecode levels. Through the source code instrumentation, we achieve execution tracing, and by using the bytecode instrumentation we accomplish communication tracing. For the coarse grained predicated slicing algorithm, we analyze both the execution and communication traces, and also use static information obtained through an existing source code parser. The combined dynamic and static information is used to create an in-memory model to abstract the message-passing program and its execution. The final coarse-grained dynamic predicate slicing algorithm is implemented based on this in-memory mode by parallel computation among threads, where each thread stands for the execution of a process in the message-passing program.

Although, the implemented algorithm can compute coarse – grained dynamic predicate slices for the sample message-passing program, there are still implementation limitations. The implemented algorithm is limited by the several factors. Firstly, there is a need to adopt the parser to different version of Java. At the current stage, the parser is limited to Java

version 1.4 and does not support Java 1.5. Secondly, both the instrumentation of the source and byte code can lead to change in the program behaviour due to the instrumentation. Thirdly, scalability with respect to recording execution traces and modeling these traces in the memory will become a major issue.

Because of the major limitations of the presented implementation, we expect as part of the future work that the static analysis will have to be extended, so that all static dependencies can be resolved, and more precise coarse-grained dynamic predicate slices can be computed. Secondly, there is a need to address the scalability issue related to tracing and analyzing these traces. Potential solutions might include more selective tracing, improved filtering of recorded information. Another future approach could be to minimize the trace complexity be investigating the possibility of using run-time information rather than recorded information to compute the coarse-grained slices.

REFERENCE

[And00] Andrews, Gregory R., *Foundations of Multithreaded, Parallel, and Distributed Programming*, Addison-Wesley, 2000.

[Agr90] Agrawal, Hiralal and Joseph R. Horgan, "Dynamic Program Slicing", *Proceedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation*, New York, USA, June 1990, pp.246-256.

[Att04] Attiya, Hagit, and Jennifer Welch, *Distributed Computing Fundamentals, Simulations, and Advanced Topics*, Second Edition, John Wiley & Sons, 2004.

[Bal99] Ball, Thomas, "The Concept of Dynamic Analysis", *ACM Conference on Foundations of Software Engineering*, Toulouse, France, September 1999, pp.216-234.

[Bel00] Bellifemine, Fabio, Agostino Poggi, Giovanni Rimassa, and Paola Turci, "An Object Oriented Framework to Realize Agent Systems", *Proceedings of WOA 2000 Workshop*, Parma, Italy, May 2000, pp.52-57.

[Bel03] Bellifemine, Fabio, Giovanni Caire, Agostino Poggi, and Giovanni Rimassa, "JADE – a White Paper", *the Special issue on JADE of the TILAB Journal, EXP - in search of innovation*, vol. 6, no. 3, September 2003, pp.6-19.

[Car92] Carriero, Nicholas, David Gelernter, *How to Write Parallel Programs: a First Course*, Massachusetts Institute of Technology Press, 1992.

[Che93] Cheng, Jingde, "Slicing Concurrent Programs - A Graph-Theoretical Approach", *Proceedings of the First International Workshop on Automated and Algorithmic Debugging*, Linkoping, Sweden, May 1993, pp.223-240.

[Chi90] Chikofsky, Elliot J., and James H. Cross II, "Reverse Engineering and Design Recovery: A Taxonomy", *IEEE Software*, vol. 7 no. 1, January 1990, pp. 13-17.

[Chi03] Chiba, Shigeru, and Muga Nishizawa, "An Easy-to-Use Toolkit for Efficient Java Bytecode Translators", *Proceedings of the Second International Conference on Generative Programming and Component Engineering*, Erfurt, Germany, September 2003, pp.364-376.

[Clo05] Clover, "Frequently Asked Questions", http://www.cenqua.com/clover/doc/faq.html, retrieved September 2005.

[Coh98] Cohen, A. Geoff, Jeffrey S. Chase, and David L. Kaminsky, "Automatic Program Transformation with JOIE", *Proceedings of the USENIX Annual Technical Symposium*, San Antonio, USA, January 1998, pp.167-178.

[Coo98] Cooper, Brian F., Han B. Lee, and Benjamin G. Zorn, *ProfBuilder: a Package for Rapidly Building Java Execution Profilers*, Technical report, University of Colorado, April 1998.

[Cou01] Coulouris, George F., Jean Dollimore, and Tim Kindberg, *Distributed systems: concepts and design*, Third Edition, Addison-Wesley, Pearson, 2001.

[Dah01] Dahm, Markus, *Byte Code Engineering with the BCEL API*. Technical Report B-17-98, Technical Report B-17-98, Freie Universit at Berlin, Institut fur Informatik, April 2001.

[Dai05] Daikon, "Daikon User Manual", http://pag.csail.mit.edu/daikon/download/doc/, retrieved September 2005.

[DeL01] De Lucia, Andrea, "Program Slicing: Methods and Applications", *the First IEEE International Workshop on Source Code Analysis and Manipulation*, Florence, Italy, November 2001, pp. 144-151.

[Due92] Duesterwald, Evelyn, Rajiv Gupta, and Mary Lou Soffa, "Distributed Slicing and Partial Re-execution for Distributed Programs", *Proceedings of the fifth Workshop on Language and Compilers for Parallel Computing*, New Haven, USA, August 1992, pp.497-511.

[Fac04] Factor, Michael, Assaf Schuster, and Konstantin Shagin, "Instrumentation of Standard Libraries in Object-Oriented Languages: the Twin Class Hierarchy Approach", *Proceedings of the nineteenth Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, Vancouver, Canada, October 2004, pp.288-300.

[Fre02] Freitag, lix, Jordi Caubet, and Jesús Labarta, "On the Scalability of Tracing Mechanisms", *Proceedings of the eighth International Euro-Par Parallel Processing Conference*, Paderborn, Germany, August 2002, pp.97-104.

[Ham01] Hamou-Lhadj, Abdelwahab, and Timothy C. Lethbridge, "A Survey of Trace Exploration Tools and Techniques", *Proceedings of the 2004 conference of the Centre for Advanced Studies on Collaborative research*, Markham, Canada, October 2004, pp.42-55.

[Har01] Harman, Mark, and Robert M. Hierons, "An overview of program slicing", *Software Focus* vol. 2, no. 3, 2001, pp.85-92.

[Hor90] Horwitz, Susan, Thomas W. Reps, and David Binkley, "Interprocedural slicing using dependence graphs", *ACM Transactions on Programming Languages and Systems*, vol. 12, no. 1, 1990, pp.26-60.

[IEE90] ANSI/IEEE Standard 610.12-1990, *IEEE Standard Glossary of Software Engineering Terminology,* the Institute of Electrical and Electronic Engineers, 1990.

[Kam95] Kamkar, Mariam, and Patrik Krajina, "Dynamic Slicing of Distributed Programs", *Proceedings of the International Conference on Software Maintenance*, Opio, France, October 1995, pp.222-229.

[Kor88] Korel, Bogdan, and Janusz W. Laski, "Dynamic program slicing", *Information Processing Letters*, vol. 29, no. 3, October 1988, pp.155-163.

[Kor92] Korel, Bogdan, and Roger Ferguson, "Dynamic Slicing of Distributed Programs", *Applied Mathematics and Computer Science Journal*, vol. 2, no. 2, 1992, pp.199-215.

[Kor98] Korel, Bogdan, and Juergen Rilling, "Program Slicing in Understanding of Large Programs", *Proceedings of the Sixth International Workshop on Program Comprehension*, Ischia, Italy, June 1998, pp.145-152.

[Kor94] Korel, Bogdan, and Satish Yalamanchili, "Forward Computation of Dynamic Program Slices", *Proceedings of the 1994 International Symposium on Software Testing and Analysis*, Seattle , USA, August 1994, pp.66-79.

[Kri98] Krinke, Jens, "Static slicing of threaded programs", *Proceedings of the 1998 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, Montreal, Canada, June 1998, pp.35-42.

[Kun94] Kunz, Thomas, "Reverse Engineering Distributed Applications: An Event Abstraction Tool", *International Journal of Software Engineering and Knowledge Engineering*, vol. 4 no. 3, September 1994, pp.303-323.

[Lam78] Lamport, Leslie, "Time, Clocks, and the Ordering of Events in a Distributed System", *Communications of the ACM,* vol. 21, no. 7, July 1978, pp.558-565.

[Lee97] Lee, Han Bok, and Benjamin G. Zorn, "BIT: A Tool for Instrumenting Java Bytecodes", *USENIX Symposium on Internet Technologies and Systems*, December 1997, pp.73-83.

[Leo01] Leopold, Claudia, *Parallel and Distributed Computing: A Survey of Models, Paradigms, and Approaches*, John Wiley & Sons, 2001.

[LiH04] Li, Hon Fung, Juergen Rilling, Dhrubajyoti Goswami, "Granularity-Driven Dynamic Predicate Slicing Algorithms for Message Passing Systems", *Automated Software Engineering*, vol. 11, no. 1, January 2004, pp 63-89.

[Lin99] Lindholm, Tim, and Frank Yellin, *the Java Virtual Machine Specification*, Second Edition, Sun Microsystems Inc., 1999.

[McC05] McCluskey, Glen, *Java Test Coverage and Instrumentation Toolkits*, Glen McCluskey & Associates LLC, http://www.glenmccl.com/instr/index.htm, retrieved September 2005.

[Mor02] Morin, Steven, Israel Koren, and C. Mani Krishna, "JMPI: Implementing the Message Passing Interface Standard in Java", *Proceedings of the 16th International Parallel and Distributed Processing Symposium*, Fort Lauderdale, USA, April 2002.

[Mul93] Mullender, Sape, *Distributed Systems*, Second Edition, Addison-Wesley, 1993.

[Nel96] Nelson, Michael L., "A Survey of Reverse Engineering and Program Comprehension", *ODU CS 551 – Software Engineering Survey*, April 1996.

[Nul03] Nulkar, Atul U., and Roger T. Alexander, "An Instrumentation Engine for Dynamic Program Analysis", *Fast Abstracts of International Symposium on Software Reliability Engineering*, Denver, USA, November 2003.

[Ott84] Ottenstein, Karl J., and Linda M. Ottenstain, "The Program Dependence Graph in a Software Development Environment", *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, vol. 19, no. 5, Pittsburgh, USA, May 1984, pp.177-184.

[Par94] Parnas, David Lorge, "Software Aging", *Proceedings of the 16th international conference on Software Engineering*, Sorrento, Italy, May 1994, pp.279-287.

[Pos05] PostgreSQL, PostgreSQL Technical Documentation, http://www.postgresql.org/, retrieved September 2005.

[Ril01] Rilling, Juergen, Ahmed Seffah, "MOOSE: A Task-Driven Program Comprehension Environment", *Proceedings of 25th International Computer Software and Applications Conference*, Chicago, USA, October 2001, pp.71-76.

[Ril02a] Rilling, Juergen, Hon Fung Li, and Dhrubajyoti Goswami, "Predicate-based Dynamic Slicing of Message Passing Programs", *The Second IEEE International workshop on Source Code Analysis and Manipulation*, Montreal, Canada, October 2002, pp.133-143.

[Ril02b] Rilling, Juergen, Ahmed Seffah, Christophe Bouthlier, "The CONCEPT project - applying source code analysis to reduce information complexity of static and dynamic visualization techniques", *Proceedings of the 1st International Workshop on Visualizing Software for Understanding and Analysis*, Paris, France, June 2002 pp.90-99.

[Sno92] Snow, C. R., *Concurrent Programming*, Cambridge University Press, 1992.

[Rug95] Rugaber, Spencer, "Program Comprehension," *Encyclopedia of Computer Science and Technology*, Draft -- to appear, April, 1995.

[Som01] Sommerville, Ian, *Software Engineering*, Sixth Edition, Addison-Wesley, 2001.

[Spi99] Spilipopoulou, Eleni, *Concurrent and Distributed Functional Systems*, University of Bristol Press, 1999.

[Tan02] Tanenbaum, Andrew S., and Maarten van Steen, *Distributed Systems Principles and Paradigms*, Prentice-Hall, 2002.

[Tip95] Tip, Frank, "A Survey of Program Slicing Techniques", *Journal of Programming Language,* vol. 3, no. 3, 1995, pp.121-189.

[Wei81] Weiser, Mark, "Program Slicing", *Proceedings of the 5th International Conference on Software Engineering*, San Diego, USA, March 1981, pp.439-449.

[Wei82] Weiser, Mark, "Programmers Use Slices When Debugging", *Communications of the ACM*, vol. 25, no. 7, July 1982, pp.446-452.

[Wei84] Weiser, Mark, "Program slicing", *IEEE Transactions on Software Engineering*, vol. 10, no. 4, July 1984, pp.352-357.

[XuB05] Xu, Baowen, Ju Qian, Xiaofang Zhang, Zhongqiang Wu, and Lin Chen, "A Brief Survey of Program Slicing", *ACM SIGSOFT Software Engineering Notes*, Vol. 30, No. 2, 2001, pp.1-16.

[Zha96] Zhao, Jianjun, Jingde Cheng, and Kazuo Ushijima, "Static Slicing of Concurrent Object-Oriented Programs", *Proceedings of the 20th Computer Software and Applications Conference*, Seoul, Korea, August, 1996, pp.312-320.

[Zha99a] Zhao, Jianjun, "Multithreaded Dependence Graphs for Concurrent Java Programs", *International Symposium on Software Engineering for Parallel and Distributed Systems*, Los Angeles, USA, May 1999, pp.13-23.

[Zha99b] Zhao, Jianjun, "Slicing Concurrent Java Programs", *Proceedings of the 7th International Workshop on Program Comprehension*, Pittsburgh, USA, May 1999, pp.126

[Zha03] Zhang, Yonggang, *Automatic Design Pattern Recovery, Thesis of Master*, Concordia University, Montreal, Canada, 2003.

APPENDICES

Message Passing Sample Program on JADE

▪ Processs0.java

```
import jade.core.*;
import jade.domain.*;
import jade.lang.acl.*;
import jade.core.behaviours.SimpleBehaviour;
import jade.domain.FIPAAgentManagement.AMSAgentDescription;

import java.io.*;

public class Process0 extends Agent {
  // Put agent initializations here
  protected void setup() {
    // Add the behaviour
    addBehaviour(new MyBehaviour());
  }

  // Put agent clean-up operations here
  protected void takeDown() {
    // Printout a dismissal message
    System.out.println("Process0 " + getAID().getName() + " terminating.");
    System.exit(AP_DELETED);
  }

  /**
   * Inner class Mybehaviour.
   */
  private class MyBehaviour extends SimpleBehaviour {

    private boolean finished = false;

    public void action() {
      try {
        int threshold0 = 0, r = 0;

        // input(threshold0);
        Reader rr = new BufferedReader(new InputStreamReader(System.in));
        StreamTokenizer streamTokenizer = new StreamTokenizer(rr);
        System.out.print("Enter threshold0 , a integer please: ");
        while (true)
          if (streamTokenizer.nextToken() == StreamTokenizer.TT_NUMBER) {
            threshold0 = (int) streamTokenizer.nval;
```

```
      System.out.println("User input threshold0 = " + Integer.toString(threshold0));
      break;
    } else
      System.out.print("Error Reading from user! \nEnter threshold0, a integer again please: ");

AMSAgentDescription template = new AMSAgentDescription();
ACLMessage msg = new ACLMessage(ACLMessage.INFORM);

// send(threshold0, 1);
template.setName(new AID("process1", false));
try {
  AMSAgentDescription[] result;
  if ((result = AMSService.search(myAgent, template)).length == 0) {
    System.out.println("Waiting for begining of process1!!!");
    while ((result = AMSService.search(myAgent, template)).length == 0)
      doWait(100);
    System.out.println("process1 has begun!!!");
  }
} catch (FIPAException fe) {
  fe.printStackTrace();
}
msg.reset();
msg.addReceiver(new AID("process1", false));
msg.setContent(Integer.toString(threshold0));
send(msg);
System.out.println("send(threshold0, 1)");

// send(threshold0, 2);
template.setName(new AID("process2", false));
try {
  AMSAgentDescription[] result;
  if ((result = AMSService.search(myAgent, template)).length == 0) {
    System.out.println("Waiting for begining of process2!!!");
    while ((result = AMSService.search(myAgent, template)).length == 0)
      doWait(500);
    System.out.println("process2 has begun!!!");
  }
} catch (FIPAException fe) {
  fe.printStackTrace();
}
msg.reset();
msg.addReceiver(new AID("process2", false));
msg.setContent(Integer.toString(threshold0));
send(msg);
System.out.println("send(threshold0, 2)");

// receive(r);
MessageTemplate msgFromProcess1 = MessageTemplate.MatchSender(new AID("process1", false));
MessageTemplate msgFromProcess2 = MessageTemplate.MatchSender(new AID("process2", false));
ACLMessage msgReceive = blockingReceive(MessageTemplate.or(msgFromProcess1, msgFromProcess2));
System.out.println("receive(r) from " + msgReceive.getSender().getName());
r = Integer.parseInt(msgReceive.getContent());

//output("The largest element below threshold is ", r);
```

```
            System.out.println("The largest element below threshold is " + Integer.toString(r));
          } catch (IOException ioe) {
            ioe.printStackTrace();
          } finally {
            finished = true;
            myAgent.doDelete();
          }
        }

        public boolean done() {
          return finished;
        }
      }
}
```

▪ Processs1.java

```
import jade.core.*;
import jade.domain.*;
import jade.lang.acl.*;
import jade.core.behaviours.SimpleBehaviour;
import jade.domain.FIPAAgentManagement.AMSAgentDescription;

import java.io.*;

public class Process1 extends Agent {

  // Put agent initializations here
  protected void setup() {
    // Add the behaviour
    addBehaviour(new MyBehaviour());
  }

  // Put agent clean-up operations here
  protected void takeDown() {
    // Printout a dismissal message
    System.out.println("Process1 " + getAID().getName() + " terminating.");
    System.exit(AP_DELETED);
  }

  /**
   * Inner class Mybehaviour.
   */
  private class MyBehaviour extends SimpleBehaviour {

    private boolean finished = false;

    public void action() {
      try {
        int threshold1 = 0, n = 0, x, a[], p = 0, sum1 = 0;
        a = new int[10];
```

```
// input(n,a);
Reader readerIn = new BufferedReader(new InputStreamReader(System.in));
StreamTokenizer streamTokenizer = new StreamTokenizer(readerIn);
System.out.print("Hom many integer will be input: n = ");
while (true)
  if (streamTokenizer.nextToken() == StreamTokenizer.TT_NUMBER) {
    n = (int) streamTokenizer.nval;
    System.out.println("User input n = " + Integer.toString(n));
    break;
  } else
    System.out.print("Error Reading from user! \nEnter n, a integer again please: ");
System.out.println("Please input " + Integer.toString(n) + " integers, a[0] to a["
    + Integer.toString(n - 1) + "] !");
for (int i = 0, errorInput = 0; i < n; i++) {
  while (true)
    if (streamTokenizer.nextToken() == StreamTokenizer.TT_NUMBER) {
      a[i] = (int) streamTokenizer.nval;
      break;
    } else
      errorInput++;
  if ((errorInput != 0) & (errorInput + i == n - 1))
    System.out.println("Error Reading from user! \nEnter " + Integer.toString(errorInput)
        + " more integer(s) again please!");
}
for (int i = 0; i < n; i++)
  System.out.println("a[" + Integer.toString(i) + "] = " + Integer.toString(a[i]) + ";");

AMSAgentDescription template = new AMSAgentDescription();
MessageTemplate msgFromProcess0 = MessageTemplate.MatchSender(new AID("process0", false));
MessageTemplate msgFromProcess2 = MessageTemplate.MatchSender(new AID("process2", false));
ACLMessage msg = new ACLMessage(ACLMessage.INFORM);

// receive(threshold1);
ACLMessage msgReceive1 = blockingReceive(MessageTemplate.or(msgFromProcess0, msgFromProcess2));
threshold1 = Integer.parseInt(msgReceive1.getContent());
System.out.println("receive(threshold1) from " + msgReceive1.getSender().getName());
System.out.println("threshold1 = " + Integer.toString(threshold1));

// (8) to (11)
while ((a[n - 1] > threshold1) && (n > 1) ) {
  sum1 = sum1 + a[n - 1];
  n = n - 1;
}
p = a[n - 1];

// send(sum1, 2);
template.setName(new AID("process2", false));
try {
  AMSAgentDescription[] result;
  if ((result = AMSService.search(myAgent, template)).length == 0) {
    System.out.println("Waiting for begining of process2!!!");
    while ((result = AMSService.search(myAgent, template)).length == 0)
      doWait(500);
```

```
            System.out.println("process2 has begun!!!");
          }
        } catch (FIPAException fe) {
          fe.printStackTrace();
        }
        msg.reset();
        msg.addReceiver(new AID("process2", false));
        msg.setContent(Integer.toString(sum1));
        send(msg);
        System.out.println("send(sum1, 2)");

        //send(p, 0);
        msg.reset();
        msg.addReceiver(new AID("process0", false));
        msg.setContent(Integer.toString(p));
        send(msg);
        System.out.println("send(p, 0)");

        //receive(x, 2);
        MessageTemplate msgOfINFORM = MessageTemplate.MatchPerformative(ACLMessage.INFORM);
        ACLMessage msgReceive2 = blockingReceive(MessageTemplate.or(msgFromProcess2, msgOfINFORM));
        System.out.println("receive(x, 2) from " + msgReceive2.getSender().getName());
        x = Integer.parseInt(msgReceive2.getContent());
        System.out.println(" x = " + Integer.toString(x));

        //sum1 := sum1 + x;
        sum1 = sum1 + x;
      } catch (IOException ioe) {
        ioe.printStackTrace();
      } finally {
        finished = true;
        myAgent.doDelete();
      }
    }

    public boolean done() {
      return finished;
    }
  }
}
```

▪ Processs2.java

```
import jade.core.*;
import jade.domain.*;
import jade.lang.acl.*;
import jade.core.behaviours.SimpleBehaviour;
import jade.domain.FIPAAgentManagement.AMSAgentDescription;

import java.io.*;
```

```
public class Process2 extends Agent {

  // Put agent initializations here
  protected void setup() {
    // Add the behaviour
    addBehaviour(new MyBehaviour());
  }

  // Put agent clean-up operations here
  protected void takeDown() {
    // Printout a dismissal message
    System.out.println("Process2 " + getAID().getName() + " terminating.");
    System.exit(AP_DELETED);
  }

  /**
   * Inner class Mybehaviour.
   */
  private class MyBehaviour extends SimpleBehaviour {

    private boolean finished = false;

    public void action() {
      try {
        int threshold2 = 0, m = 0, y, b[], q = 0, sum2 = 0;
        b = new int[10];

        // input(m,b);
        Reader readerIn = new BufferedReader(new InputStreamReader(System.in));
        StreamTokenizer streamTokenizer = new StreamTokenizer(readerIn);
        System.out.print("Hom many integer will be input: m = ");
        while (true)
          if (streamTokenizer.nextToken() == StreamTokenizer.TT_NUMBER) {
            m = (int) streamTokenizer.nval;
            System.out.println("User input m = " + Integer.toString(m));
            break;
          } else
            System.out.print("Error Reading from user! \nEnter m, a integer again please: ");
        System.out.println("Please input " + Integer.toString(m) + " integers, b[0] to b["
            + Integer.toString(m - 1) + "] !");
        for (int i = 0, errorInput = 0; i < m; i++) {
          while (true)
            if (streamTokenizer.nextToken() == StreamTokenizer.TT_NUMBER) {
              b[i] = (int) streamTokenizer.nval;
              break;
            } else
              errorInput++;
          if ((errorInput != 0) & (errorInput + i == m - 1))
            System.out.println("Error Reading from user! \nEnter " + Integer.toString(errorInput)
                + " more integer(s) again please!");
        }
        for (int i = 0; i < m; i++)
          System.out.println("b[" + Integer.toString(i) + "] = " + Integer.toString(b[i]) + ";");
```

```
AMSAgentDescription template = new AMSAgentDescription();
MessageTemplate msgFromProcess0 = MessageTemplate.MatchSender(new AID("process0", false));
MessageTemplate msgFromProcess1 = MessageTemplate.MatchSender(new AID("process1", false));
ACLMessage msg = new ACLMessage(ACLMessage.INFORM);

// receive(threshold2);
ACLMessage msgReceive1 = blockingReceive(MessageTemplate.or(msgFromProcess0, msgFromProcess1));
System.out.println("receive(threshold2) from " + msgReceive1.getSender().getName());
threshold2 = Integer.parseInt(msgReceive1.getContent());
System.out.println("threshold2 = " + Integer.toString(threshold2));

// (18) to (21)
while ((b[m - 1] > threshold2) && (m >1)) {
  sum2 = sum2 + b[m - 1];
  m = m - 1;
}
q = b[m - 1];

// send(sum2, 1);
template.setName(new AID("process1", false));
try {
  AMSAgentDescription[] result;
  if ((result = AMSService.search(myAgent, template)).length == 0) {
    System.out.println("Waiting for begining of process1!!!");
    while ((result = AMSService.search(myAgent, template)).length == 0)
      doWait(500);
    System.out.println("process1 has begun!!!");
  }
} catch (FIPAException fe) {
  fe.printStackTrace();
}
msg.clearAllReceiver();
msg.addReceiver(new AID("process1", false));
msg.setContent(Integer.toString(sum2));
send(msg);
System.out.println("send(sum2, 1)");

//send(q, 0);
msg.clearAllReceiver();
msg.addReceiver(new AID("process0", false));
msg.setContent(Integer.toString(q));
send(msg);
System.out.println("send(q, 0)");

//receive(y, 1);
MessageTemplate msgOfINFORM = MessageTemplate.MatchPerformative(ACLMessage.INFORM);
ACLMessage msgReceive2 = blockingReceive(MessageTemplate.or(msgFromProcess1, msgOfINFORM));
System.out.println("receive(y, 1) from " + msgReceive2.getSender().getName());
y = Integer.parseInt(msgReceive2.getContent());
System.out.println(" y = " + Integer.toString(y));

//sum2 := sum2 + y;
sum2 = sum2 + y;
} catch (IOException ioe) {
```

```
            ioe.printStackTrace();
        } finally {
            finished = true;
            myAgent.doDelete();
        }
    }

    public boolean done() {
        return finished;
    }
  }
}
```