Regression Test Selection for Distributed Java RMI Programs by Means of Formal Concept Analysis

Hong Fei Zhu

A thesis
in
The Department
of
Computer Science and Software Engineering

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

December, 2008

Your file Votre référence
ISBN: 978-0-494-63211-6
Our file Notre référence
ISBN: 978-0-494-63211-6

ABSTRACT

Regression Test Selection for Distributed Java RMI Programs by Means of Formal Concept Analysis

Hong Fei Zhu

Software maintenance is the process of modifying an existing system to ensure that it meets current and future requirements. As a result, performing regression testing becomes an essential but time consuming aspect of any maintenance activity. Regression testing is initiated after a programmer has made changes to a program that may have inadvertently introduced errors. It is a quality control approach to ensure that the newly modified code still complies with its specified requirements and that unmodified code has not been affected by the maintenance activity. In the literature various types of test selection techniques have been proposed to reduce the effort associated with re-executing the required test cases. However, the majority of these approach has been focusing only on sequential programs, and provide no or only very limited support for distributed programs or database-driven applications.

The thesis presents a lightweight methodology, which applies Formal Concept Analysis to support a regression test selection analysis, in combination with execution trace collection and external data sharing analysis, for distributed Java RMI programs. Two Eclipse plug-ins were developed to automate the regression test selection process and to evaluate our methodology.

Acknowledgements

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

My sincere appreciation is extended to my father, Zhu Heng Lin, my mother, Liu Mei Yu and my family in China, for their unconditional love and support. I would also like to give the special thanks to my wife, Zhang Zhi Yu, for always loving me and keeping me motivated.

Table of Contents

List of Figures

List of Tables

1. Introduction

Software maintenance is the process of modifying an existing system to ensure that it meets current and future requirements. Regression testing is initiated after a programmer has made changes to a program that may have inadvertently introduced errors. It is a quality control measure to ensure that the newly modified code still complies with its specified requirements and that unmodified code has not been affected by the maintenance activity [HAR00]. The easiest method for performing regression testing is the reuse of an original test suite and rerun all the test cases in it. However, when the change to a system is minor, a complete rerun of the full test suit is not only often unnecessary but also expensive. As a result an alternative approach is needed that allows for the selection and re-execution of only the test cases that are relevant to the specific modification.

Regression test selection is such an approach that attempts to reduce the cost of retesting, by identifying and re-executing only a subset of the existing test suite in order to re-test the code potentially affected by a modification request [GRA01]. Regression test selection involves the recording of program elements exercised by tests used in previous releases, and selecting these test cases that exercise elements changed in the current release. The coverage matrix between the code entities and test cases needs to be identified by tracing the actual execution paths of the test cases through the code [SNE04]. Selective regression testing is a well established research domain with a wide range of existing approaches, varying from the use of control flow information and/or data flow dependencies for procedural, object-oriented and aspect-oriented programs [CHE94, HSI97, HAR01B, ROT97, ROT00, WHI92, WHI97, XU07, ZHA06]. However, these existing approaches have focused primarily only on sequential programs, with none of them providing support for regression test selection for distributed programs (such as Java RMI applications). Performing selective regression test for distributed programs is clearly a more challenging task compared to performing it on sequential programs, since it not only requires to trace client/server activities across multiple threads and processors, but also to merge local and remote calls by examining causality relationship between them.

Another shortcoming of these existing regression test selection techniques is that they only deal with the manipulation of internal program states, and ignore typically external program states (persistent variables, e.g., database and files) in their analysis. However, these persistent states not only play an important role in modern software, especially database-driven system, but also might affect a selective regression testing analysis. Through the persistent states, the change effect could be transmitted from one code entity (i.e. function, component, or even program) to other code entities. The omission of the persistent states could lead to scenarios where test cases affected by the modifications might not be selected and re-executed.

The research is motivated by the need to provide software maintainers and managers with the ability to estimate early on during the maintenance cycle, the testing effort associated with a modification request. In this research, we address this issue of predicting the regression testing effort, by proposing a lightweight methodology, which applies Formal Concept Analysis to support a regression test selection analysis, in combination with execution trace collection and external data sharing analysis, for distributed Java RMI programs. A toolkit was developed, consisting of two Eclipse plug-ins that are used to automate the regression test selection process and allow us to validate our approach. The toolkit is able to collect distributed execution traces, implement external data sharing analysis algorithm to generate the test case dependency table, and visualize the selection result.

The remainder of the thesis is organized as follows. Chapter 2 provides the background related to program comprehension, including dynamic analysis, aspect oriented programming, Java RMI, Formal Concept Analysis, and regression testing. Chapter 3 states the main contributions of this thesis, including the motivation, research hypothesis, and research goals. Chapter 4 shows the lightweight regression test selection approach for distributed Java RMI applications. In Chapter 5, the implementation of the tool to support our methodology for automating the analysis process is introduced. Chapter 6 elaborates on each step of the problem solving approach through prepared case studies. The discussions of the advantages as well as limitations of the introduced approach, and the related researches are also presented. Finally, Chapter 7 concludes the thesis and discusses potential future work.

2. Background

In the following sections, we will introduce background information relevant to this research. In particular we focus in our review on dynamic analysis (section 2.1), Aspect Oriented Programming (AOP) and AspectJ (section 2.2), Java RMI (section 2.3), Formal Concept Analysis (section 2.4), and regression testing (section 2.5).

2.1 Dynamic Analysis

Program Comprehension is the process of acquiring knowledge about a computer program [RUG95]. It 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. Program comprehension plays a significant role in software maintenance and evolution. A significant proportion of the time required for maintaining, debugging, and reusing existing code is spent in understanding existing programs [STO97].

Program comprehension can be performed through two types of analysis: static analysis (reading the code) and dynamic analysis (running the code) [COR89]. Static analysis collects its information statically through fact extraction from artifacts such as the source code, design documents, etc. and then analyzes these collected facts to abstract and interpret the program properties. In contrast dynamic analysis collects knowledge about system properties by executing a software system for various inputs [BAL99].

Dynamic analysis supports program comprehension in particular by providing additional insights with respect to behavioral aspects of a software system, which are often not well documented in system documentation [GSC03]. Using dynamic program analysis requires some form of instrumentation of the original software application or its underlying runtime system to generate traces of real program executions. Through the analysis of these traces, it is typically possible to identify those parts of the program that implement the functionality of interest and hence, need to be understood.

Object-oriented systems are difficult to understand by relying only on static analysis, due to object oriented specific features such as inheritance, dynamic binding and polymorphism. These language features tend to obscure the relationships among the system artifacts [STV05]. As a result, the behavior of OO systems can often only be completely determined through the use of runtime (dynamic) information. Since dynamic analysis can take advantage of run-time information, it can overcome many of the shortcomings of static analysis. Also through the use of run-time information (such as object instantiation and communication, method calls, and branching decisions), dynamic analysis can provide additional insights on the life cycle of objects, the sequences of interactions, and the flow of control between components at run-time. Furthermore, given the more detailed information available for the analysis, dynamic analysis can be more precise and sensitive to the input data [BAL99], and hence improve the comprehension process.

Figure 2-1 shows a general overview of the major steps involved while performing dynamic analysis. Firstly, the program under test or its underlying runtime system is instrumented in order to put probes collecting the dynamic information. Then, the destination program is executed with a set of test cases. The trace data is produced and transmitted into some type of repository/data store. Due to the fact that important interactions are mixed with low-level implementation details, traces can be very large and hard to understand [HAM03]. Therefore, in the third step, depending on the analysis focus, traces are either compressed and/or abstracted to remove unnecessary data (i.e. utility functions, repetitive and recursive calls, redundancy patterns, etc.). At last, the filtered traces are processed to present the program's high-level behavioral view (i.e. sequence diagram), or for further analyses (i.e. feature identification).

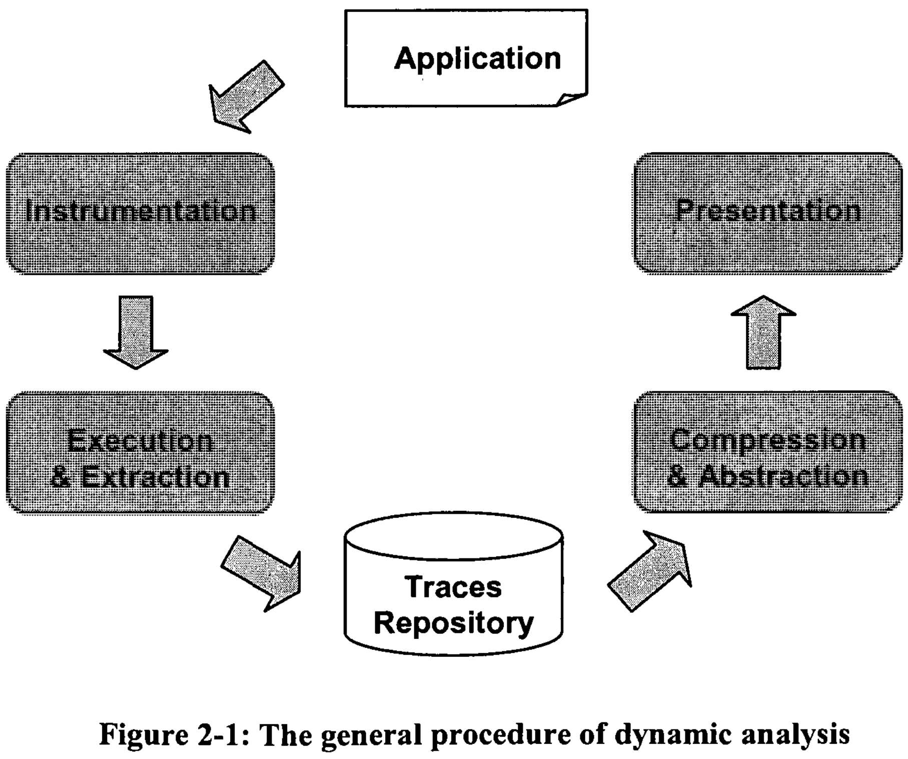


Figure 2-1: The general procedure of dynamic analysis

2.1.1 Instrumentation

As mention previously, dynamic analysis involves some form of instrumentation of the system to be analyzed, to allow for the collection of certain run-time states and program properties. For Java programs, instrumentation is typical performed through one of two approaches. (1) Code instrumentation; which requires inserting additional statements (probes) into a program (source or byte code) to allow collecting dynamic behavior information [HUA78]. (2) Leverage capabilities of the runtime environment, by monitoring and tracking the runtime behavior of an application through debugging, profiling or modifying the Java Virtual Machine [SEE05]. In what follows, we provide a more detailed overview of some major techniques used to collect dynamic information from Java programs.

2.1.1.1 Source Code Instrumentation

One way for instrumenting a program is to simply add code (probes) needed for the instrumentation into the source code. These probes become part of the program build and the resulting object code contains code corresponding to the instrumentation code which was added to the source code. When executing the program, the code for the probes will be executed and dynamic information for the application can be obtained. The advantages of the source code instrumentation are: (1) it supports for statement level source code instrumentation as it is typically used by source code coverage tools, statements and branch coverage; (2) it does not require a specialized runtime environment, the instrumented applications can run within the same program environments as the original programs. On the other hand, the source code instrumentation also has drawbacks: firstly, the source code of an application must be available; moreover, the instrumentation even being semantically and syntactically correct alters the original source code. As a result the source code being executed and analyzed might no longer reflect the behavior of the source code.

There exist several tools for the source code instrumentation. *Clover* is a commercial code coverage analysis tool, developed by Cenqua Pty Ltd. [CLO07]. It copies and instruments a set of Java source files, and then measures three types of coverage analysis: statement, branch and method coverage. The Java test coverage and instrumentation toolkits, *query and instr* [MCC07], are used to parse Java source programs into an internal tree form, and perform method and statement source instrumentation. The toolkits are suited for applications, such as test coverage, metrics, instrumentation, extraction of information, documentation tools, etc.

2.1.1.2 Bytecode Instrumentation

Java source code is normally compiled into a binary format consisting of a bytecode instruction set (i.e. the class file) as an intermediate format. After instrumenting the bytecode, the bytecode instructions are executed by the Java Virtual Machine (JVM) [LIN99]. Java bytecode instrumentation, also called bytecode injection, is the process of directly inserting or manipulating Java bytecode. It generally inserts a special, short sequence of bytecode at the designated points within a Java class file. The introduced bytecode controls the message passing. The Java bytecode instrumentation can be performed either *statically* at the compile time or *dynamically* at the runtime when the bytecode of the class is being loaded into the JVM.

Static bytecode instrumentation inserts all instrumentation-code before the program under instrumentation starts execution. The main advantage of this approach is that it causes less runtime overhead, as all classes are instrumented before the program is executed. The major drawback of static instrumentation is that dynamically generated or loaded code is not instrumented. Some high-level bytecode engineering libraries can be leveraged to perform static bytecode instrumentation. *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. *Bytecode Instrumenting 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. *Java programming assistant* (Javassist) [CHI03] is a reflection-based toolkit for developing Java bytecode translators. The main feature of Javassist is that it allows users to access Java bytecode in the high source code level, instead of in the low bytecode instruction level. This means that programmers can modify a class file with source-level vocabulary.

Dynamic bytecode instrumentation is interleaved with the execution of the program under instrumentation; an instrumentation agent is invoked each time a class is loaded and may augment the loaded bytecode with instrumentation code. The weakness of this approach is that it introduces extra overhead and may perturb measurements due to the runtime instrumentation process. The advantage of this method is that it ensures that all loaded classes will be instrumented and avoids the often tedious bytecode instrumentation prior to the program startup. Dynamic instrumentation is applied not to all library classes, but only to those classes that are actually being loaded. Furthermore, it also prevents problems, such as forgetting to instrument classes after modification and recompilation. *org.jmonde.debug.Trace* [JMO07] is an on-the-fly runtime method tracing tool for Java applications based on the Byte Code Engineering Library. Its working mechanism is as follows: a custom class loader reads the class file and instruments each method with tracing code. The class loader also adds a static field to each class. This field has two states, 'on' and 'off'. The tracing code checks this field prior to printing. The command line options access and modify this static field to control tracing output.

2.1.1.3 Interfacing with Java Virtual Machine

Another approach for instrumenting Java source code is by interfacing with the Java Virtual Machine through the debugging (JVMDI) and profiling (JVMPI, JVMTI) interfaces. They provide ways to inspect the state and to control the execution of applications running in the Java virtual machine (JVM). The Java Virtual Machine Debugging Interface (JVMDI) [SUN99] defines the services a VM provides for debugging. It includes requests for information (for example, current stack frame), actions (for example, set a breakpoint), and notification (for example, when a breakpoint has been hit). The performance penalty using the JVMDI is so significant that its applicability is limited only for very short program executions. The Java Virtual Machine Profiler Interface (JVMPI) [SUN02] is a two-way method call interface between the JVM and an in-process profiler agent. JVMPI provides hooks into the JVM that can be used without modifying the user program or the JVM itself. A profiler agent instructs the virtual machine to send it the relevant JVMPI events, such as method enter and exit, and processes the event data into profiling information. The Java Virtual Machine Tool Interface (JVM TI) [SUN04] is a new profiling interface, which was introduced in J2SE 5.0 and replaced JVMDI and JVMPI. JVMTI provides both a way to inspect the state and to control the execution of applications running in the Java virtual machine (JVM). It supports the full breadth of tools that need access to JVM state, including but not limited to: profiling, debugging, monitoring, thread analysis, and coverage analysis tools. Profiling tools based on JVMPI or JVMTI can obtain a variety of information for a comprehensive performance analysis task. Whereas these tools have to be written in platform native code, and become less transportable.

Currently, most profiling tools are based on profiler agents that use JVMPI or JVMTI. *OptimizeIt* [OPT07] is a commercial tool and it allows local and remote profiling of Java programs on multiple platforms. Its main work of the instrumentation is assigned to JVMPI. OptimizeIt collects all the information generated by JVMPI and stores it in his internal structures. OptimizeIt contains a graphic visualizer of all information, and with this information it offers different types of profiling: CPU profiling, Memory debugging, Object allocations etc. *jProf* [JPR07] is a non-commercial profiler developed using JVMPI, it was constructed to identify the typical problems that appear in java application developing: excessive memory usage, excessive synchronization and excessive processor usage. The profiler gets the information generated by JVMPI, produces XML profiling result, and presents the results of a profiling in HTML format.

2.1.1.4 Instrumented Java Virtual Machine

The Java virtual machine (JVM) is instrumented for monitoring and management, providing built-in management capabilities for both remote and local access. In particular, the JVM also can be instrumented statically or dynamically in order to export more specific and detailed information, such as start and exit time of methods, client/server interactions, etc. Statically instrumented JVM approaches instrument the JVM program in order to export some state information available while it executes the bytecode. The Dynamically Instrumented JVM approach generates and inserts instrumentation code into the JVM, or removes it from the JVM at runtime. An instrumented JVM does not require the source code of applications under test, and it can provide more flexibility to users. However, the development effort of this approach is much higher than using profiling interfaces such as JVMPI and JVMTI. In addition, the evolution of a supported JVM, or supporting more JVMs, can induce a high maintenance cost.

In what follows, we will introduce some Java profilers that use instrumented JVM to collect runtime information. The project of *JaViz* [KAZ00] started in 1997 at the University of Minnesota with the idea of providing Java software developers an easy way to collect performance data and analysis. JaViz uses an instrumented JVM capable of collect information about start and exit time of methods and to record client/server interactions. When a program is executed with the instrumented JVM, the trace files are generated. These files are then post processed to create an execution tree. JaViz has a visualizer that presents the information in an execution tree, with callers being parents of callees. *Jinsight 2.0* [PAU01] is a profiler developed by IBM to show performance bottlenecks, object creation, garbage collection, thread interaction, deadlocks and program execution patterns. It offers a modified JVM with which the application must be executed to obtain the profile. All the performance data extraction is done inside the instrumented JVM. The trace files are then visualized in different views: the histogram view, which shows the program's use of resources; the execution view, which shows the program execution sequence. Jinsight 2.0's subsequent version 2.1 supplies a profiling agent using the JVMPI for Java 2 instead of using an instrumented JVM.

2.1.2 Dynamic Analysis Applications

Dynamic analysis plays a critical role during program comprehension and is supported through techniques such as program slicing, visualizing the behavior of the system, identifying design pattern, feature to code assignment, etc. In what follows, we describe some of these approaches in more details.

2.1.2.1 Debugging and Program Comprehension

Program slicing is a method of program decomposition, and the process of it deletes those parts of the program that can be determined to have no effect upon the semantics of interest. The result of program slicing is 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 [WEI82, WEI84]. Dynamic Slicing was originally introduced in [KOR88], which aims to reduce the size of a slice and get more accurate slice based on program executions. In order to compute a dynamic slicing, an execution trace is recorded first, and then the trace is traversed backwards to derive data and control dependencies to compute the dynamic slice [AGR90]. Using the run-time information, the approach may significantly reduce the size of a program slice, and is possibly able to resolve some of the conservative assumptions that have to be made by static slicing regarding the control flow.

2.1.2.2 Dynamic Views

There exist several approaches to explore execution traces [BRI03, GUE05, HAM05] to support the understanding of a program behavior by reconstructing its dynamic views, such as sequence/scenario, statechart diagram to show program interactions at different abstraction levels [SAL06]. Also UML v.2 supports the use of composition operators to combine dynamic diagrams from traces]. Leveraging these methods, maintainers are provided with diagrams at various abstraction levels, allowing them to check the conformance between produced diagrams and documented diagrams.

2.1.2.3 Design Pattern Identification

For precise design pattern recognition, especially for object-oriented languages, static analysis might not be sufficient, due to structural similarities among patterns. Patterns often rely on polymorphism and dynamic method binding. As a result these patterns are not distinguishable from each other using static analysis, since they often differ only in their behavior (e.g. method invocations) [HEU02]. Dynamic analysis on the other hand supports the analysis of runtime behavior. However, the amount of data gathered during runtime (in the form of traces) used for pure dynamic analysis is often very large. Depending on the level of information detailed needed for dynamic analysis, the analysis can become very expensive and in some cases even unfeasible. Therefore, most of these approaches combine static and dynamic analysis techniques [WEN03, WEN04]. They use static analysis identify pattern instance candidates to reduce the search space, and then use dynamic analysis to confirm or weaken the results from static analysis. By this way, the quality of design pattern identification is highly improved.

2.1.2.4 Feature Location

Software developers are constantly required to modify and adapt application features in response to changing requirements. However, relying only on static analysis is difficult to determine how software entities contribute to the runtime behavior of features and how these features interact. Comparing with static analysis, dynamic analysis is a reliable means of associating behaviors of a system with the internal components of its implementation. Based on dynamic analysis, these approaches [EIS01, EIS03] leverage extracted execution traces to achieve an explicit mapping between the system's externally visible behavior (features) and the relevant parts of the source code. In these approaches, features are defined as units of behavior of a system; techniques such as concept analysis, data mining [GRE05] are used to identify the groups of software entities (i.e. classes, functions) that implement software features.

2.1.2.5 Other Dynamic Analysis Approaches

[ZAI05, WAN05] identify key classes and utility classes in a system by using web-mining principles or dynamic fan-in, fan-out metrics. Helping software engineers to start their reconnaissance of the software from important classes, these approaches alleviate their program comprehension task. Zaidman's work [ZAI04] is centered on the idea that the relative execution frequency of methods or procedures can tell something about which methods or procedures are working together to reach a common goal. An iterative approach using dynamic information to support the recovery and understanding of collaborations was presented in [RIC02]. In Richner's work [RIC02] collaboration abstractions are extracted without reliance on visualization techniques. Dynamic analysis implies large amounts of data. [HAM06] addressed this issue by summarizing the content of large execution traces. It first identifies utility routines and consequently summarizes these routine.

2.2 AOP and AspectJ

In what follows we provide a brief introduction to Aspect Oriented programming (AOP). One of the key elements differentiating the AOP programming paradigm from traditional object-oriented programming is it support for separation concerns. AOP's support for separation of concerns, specifically cross-cutting concerns, through additional language constructs. A program can be broken down into distinct parts that overlap in functionality by separating concerns. All programming methodologies—including procedural programming and object-oriented programming—support some separation and encapsulation of concerns into single entities, such as procedures, packages, classes, and methods [BJO06]. However, some concerns, named as crosscutting concerns, defy these forms of encapsulation "cut" across multiple modules in a program.

AOP provides language mechanisms that explicitly capture crosscutting. It extracts scattered concerns from classes and turns them into aspects, which are well modularized crosscutting concerns. By decoupling these concerns and placing them in aspects, the original classes are relieved of the burden of managing functionalities orthogonally related to their purpose. Later, the aspect code is injected into appropriate places by a process known as weaving. A direct consequence of aspect use is to program crosscutting concerns in a modular way, and achieves the usual benefits of improved modularity: simpler code that is easier to develop and maintain, and that has greater potential for reuse. Logging is one example of a crosscutting concern, because a logging strategy necessarily affects every single logged part of the system. Logging thereby crosscuts all logged classes and methods. One of the key advantages of AOP is that it provides native language support for logging and tracing of program execution [ELR01].

AspectJ [ECL07] is a simple and practical aspect-oriented extension to Java. It helps to manage crosscutting concerns by augmenting Java language with number of new structures, such as pointcuts and advice. In AspectJ's dynamic join point model, a set of identifiable points in the execution of the program, called join points, are collected though pointcuts. Code defined in advice is attached to these poinctcuts and executed when join points are reached. Aspects are class-like modular units of crosscutting implementation, comprising pointcuts, advice, and ordinary Java member declarations. AspectJ files are compiled together with standard Java source files into standard Java byte code via AspectJ compiler so that platform-independence is assured henceforward.

2.2.1 AspectJ Semantic

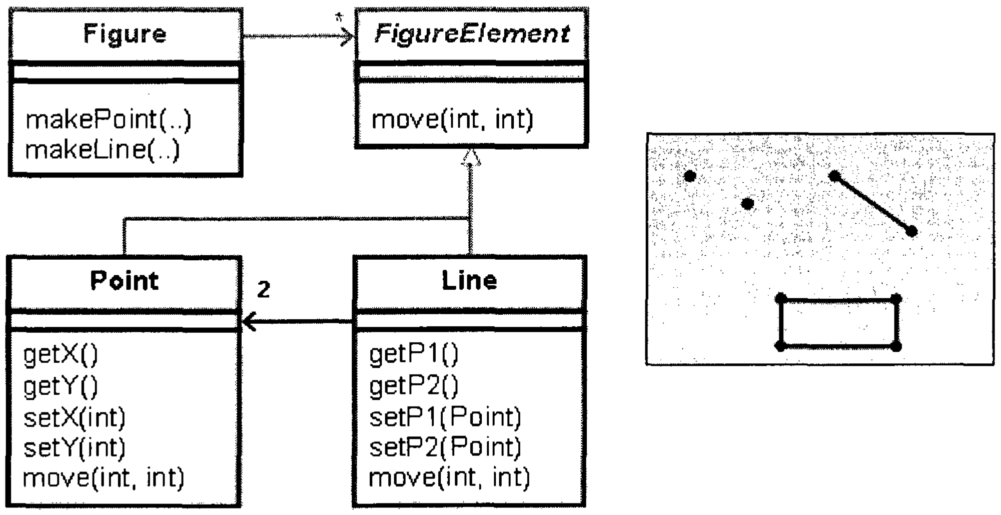


Figure 2-2: A simple figure editor

The semantics are presented using a simple figure editor system shown in Figure 2-2. In this example a `Figure` class provides factory services and it consists of a number of `FigureElements`, which can be either `Points or Lines.`

2.2.1.1 The Join Point Model

A critical element in the design of any aspect-oriented language is the join point model. The join point model provides the common frame of reference that makes it possible to define the structure of crosscutting concerns. The dynamic crosscutting elements of AspectJ are now based on a model in which join points are certain well-defined points in the execution of the program. In this model join points can be considered as nodes in a simple runtime object call graph. These nodes include points at which an object receives a method call and points at which a field of an object is referenced. The edges are control flow relations between the nodes. In this model control passes through each join point twice, once on the way in to the sub-computation rooted at the join point, and once on the way back out. The different kinds of join points provided by AspectJ are stated as follows: a call or an execution to a method or a constructor, an exception handler, an initialization to a class or an object, a field access, etc.

2.2.1.2 Pointcut Designators

In AspectJ, *pointcut designators* identify collections of join points in the program flow. They can be categorized to *name-based pointcut designators*, *property-based pointcut designators* and *control-flow based pointcut designators.*

(1) Name-based pointcut designators

The pointcut designators are all based on explicit enumeration of a set of method. For example, the following pointcut designator identifies all calls to the method `getP1()` defined on `Line` objects:

```
call(Point Line.getP1())
```

Pointcut designators can be combined using a set algebra semantics, such as and, or and not operators ('&&', '||', '!'); and it can crosscut classes and identify join points from many different classes. For example:

```
pointcut moves():
    call(void FigureElement.move(int, int)) ||
    call(void Point.setX(int)) ||
    call(void Point.setY(int)) ||
    call(void Line.setP1(Point)) ||
    calls(void Line.setP2(Point));
```

defines a pointcut named "`moves`" that designates calls to any of the methods that move figure elements.

(2) Property-based pointcut designators

AspectJ also provides mechanisms that enable specifying a pointcut in terms of properties of methods other than their exact name. The simplest of these involve using wildcards in certain fields of the method signature. For instance:

```
call(void Point.set*(int))
```

identifies calls to any method defined on `Point`, whose name begins with `"set"` and it needs one integer parameter and has no return value, specifically the methods `setX(int)` and `setY(int)`; and

```
call(public * Point.* (..))
```

identifies calls to any public method defined on `Point` with any parameters and return value.

(3) Control-flow based pointcut designators

These pointcuts capture join points based on the control flow of join points captured by another pointcut. A control-flow pointcut always specifies another pointcut as its argument. There are two control-flow pointcuts. The first pointcut is expressed as `cflow(Pointcut)`, and it captures all the join points in the control flow of the specified pointcut, including the join points matching the pointcut itself. The second pointcut is expressed as `cflowbelow(Pointcut)`, and it excludes the join points in the specified pointcut. For instance:

```
cflow(call(void Line.setP1(Point)))
```

identifies all the join points in the control flow of any `setP1(Point)` method in `Line` that is called, including the call to the `setP1(Point)` method itself.

```
cflowbelow(call(void Line.setP1(Point)))
```

identifies all the join points in the control flow of any `setP1(Point)`method in `Line` that is called, but excluding the call to the `setP1(Point)`method itself.

2.2.1.3 Advice

Advice is a method-like construct that provides a way to express crosscutting action at the join points that are captured by a pointcut. AspectJ has three different kinds of advice that define additional code running at join points. (1) *Before advice* runs when a join point is reached and before the computation proceeds, i.e. that runs when computation reaches the method call and before the actual method starts running. (2) *After advice* runs after the computation 'under the join point' finishes, i.e. after the method body has run, and just before control is returned to the caller. (3) *Around advice* runs when the join point is reached, and has explicit control over whether the computation under the join point is allowed to run at all. In the following code snippet, the advice prints the log string prior to the execution of any set method in the `Point` class:

```
before() : call(void Point.set*(int)) {
      System.out.println("Before setting point x or y value.");
}
```

2.2.1.4 Aspect

Aspects are modular units of crosscutting implementation. They are defined by aspect declarations, which have a form similar to that of class declarations. Aspect declarations may include pointcut declarations, advice declarations, as well as all other kinds of declarations permitted in class declarations. The following declaration defines an aspect that implements the behavior of updating display of a line moved recently.

```
aspect DisplayUpdating {

      static boolean movedFlag = false;

      pointcut move():
            call(void Line.setP1(Point)) ||
            call(void Line.setP2(Point));

      after() : move() {
            movedFlag = true;
            Display.update();
      }
}
```

2.2.2 Tracing with AspectJ

Tracing involves recording an execution of a software system in order to debug, analyze, and modify the software system. In fact, tracing is a valid example of a crosscutting concern since this concern cuts orthogonally across a number of classes and requires coding in a number of places to perform the same task. The points at which we have to perform tracing are typically method calls, event invocation etc. are all join points. Therefore, AOP can be used to solve this problem through the following step:

- Identify individual groups of join-points of interest for tracing activities
- Design pointcuts to filter out these groups
- Associate advice with these pointcuts to perform the logging activities

AspectJ itself provides an efficient programming language environment to create traces for Java programs. The three main elements of AspectJ, pointcut, advice, and join point, powerfully support a flexible extraction of the information of source codes. Pointcut addresses packages, classes, methods, and variables that could be interesting for developers. Advice arranges appropriate information for the pointcut. Join-point filters out the information in execution time. Using these three elements, AspectJ enables people to extract the execution information they want to know from source codes and delve into important parts iteratively.

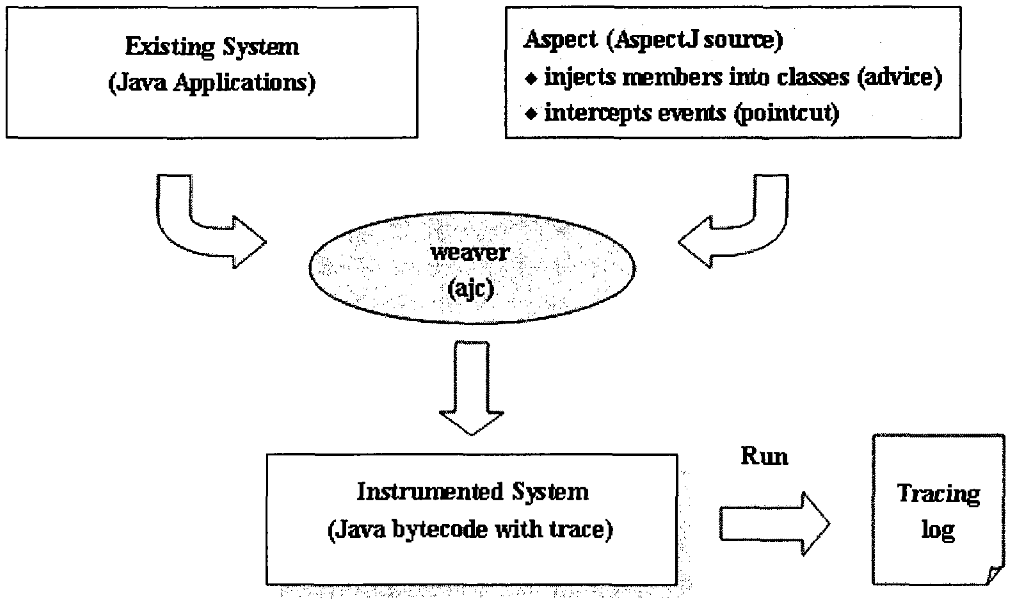


Figure 2-3: Tracing process for Java applications through AspectJ

Given the advanced tracing capabilities of AspectJ, it can be applied for analyzing and understanding of existing software systems. Its tracing capabilities can also support reverse engineering by capturing key execution points, identifying the core execution path, without requiring instrumentation or modification of the destination source code, etc. Figure 2-3 illustrates the use of AspectJ for tracing existing Java programs. What follows summarizes the major advantages and disadvantages of current tracing capabilities in AspectJ.

Advantages:

- **Non-intrusive instrumentation**

  AspectJ works at the bytecode level and does not equip source code with any instrument code. It facilitates configuration management and maintenance: tracing functionality can be easily added, modified or deleted in a non-intrusive manner giving complete control on tracing for the entire application.

- **Flexible expression**

  AspectJ offers various expressions (pointcuts) for describing source code locations to check at run-time. These pointcuts could be generic or specific depending upon how specific is the filtering criteria. For example, pointcuts can be designed to take care of join-points associated with discrete points in an inheritance hierarchy.

- **Easy refinement**

  AspectJ allows users to refine the tracing log by adding another condition to pointcuts. The combination of several pointcuts conditions also reduces the amount of tracing log; users can reduce the amount of tracing records until they find the precise information.

Limitations:

- **Lack of support for trace control flow**

  AspectJ does not currently provide mechanisms to intercept method control flow, such as repetitions of message and conditions under which messages are exchanged by objects. The alternative would be to manually instrument the code or to use debuggers or profilers.

- **Lack of support for tracing outside the package**

  AspectJ does not allow aspects to be woven into Java's library packages. That means that if a class is an extension of a class in Java's library, whenever an event is caught from the former class, the trace will show a method call that seems to come out of nowhere.

- **Requires rebuilding process**

  In order to weave aspects into Java byte code of the destination, AspectJ files need to be compiled together with standard Java source files or Java compiled files (class file, jar file) via AspectJ compiler

2.3 Java RMI

2.3.1 Overview of Java RMI

Java Remote Method Invocation (RMI) is an object model for creating distributed Java-based applications. Simplifying the communication between two objects in different Java Virtual Machines (JVM), Java RMI enables objects in one JVM to invoke methods on objects in other JVMs, in the same way as methods of local objects. On one hand, Java RMI is capable to work as a stand-alone middleware platform. On the other hand, it also acts as the foundation for other high level frameworks, such as Enterprise JavaBeans and Jini.

The general Java RMI architecture is shown in Figure 2-4. Java RMI includes three independent layers:

(1) The stub/skeleton layer is the interface between the application layer and the rest of the RMI system. A stub for a remote object is the client-side proxy, which forwards the request from the client to the actual remote object. A skeleton is a server-side entity, which dispatches calls to the actual object in the server.

(2) The remote reference layer is responsible for carrying out the semantics of the invocation and sits on top of the low-level. It has the client-side and the server-side components.

(3) The transport layer is based on TCP/IP connections among different machines in the network. It is responsible for the set-up and management of the connection and dispatching the requests to the remote objects.

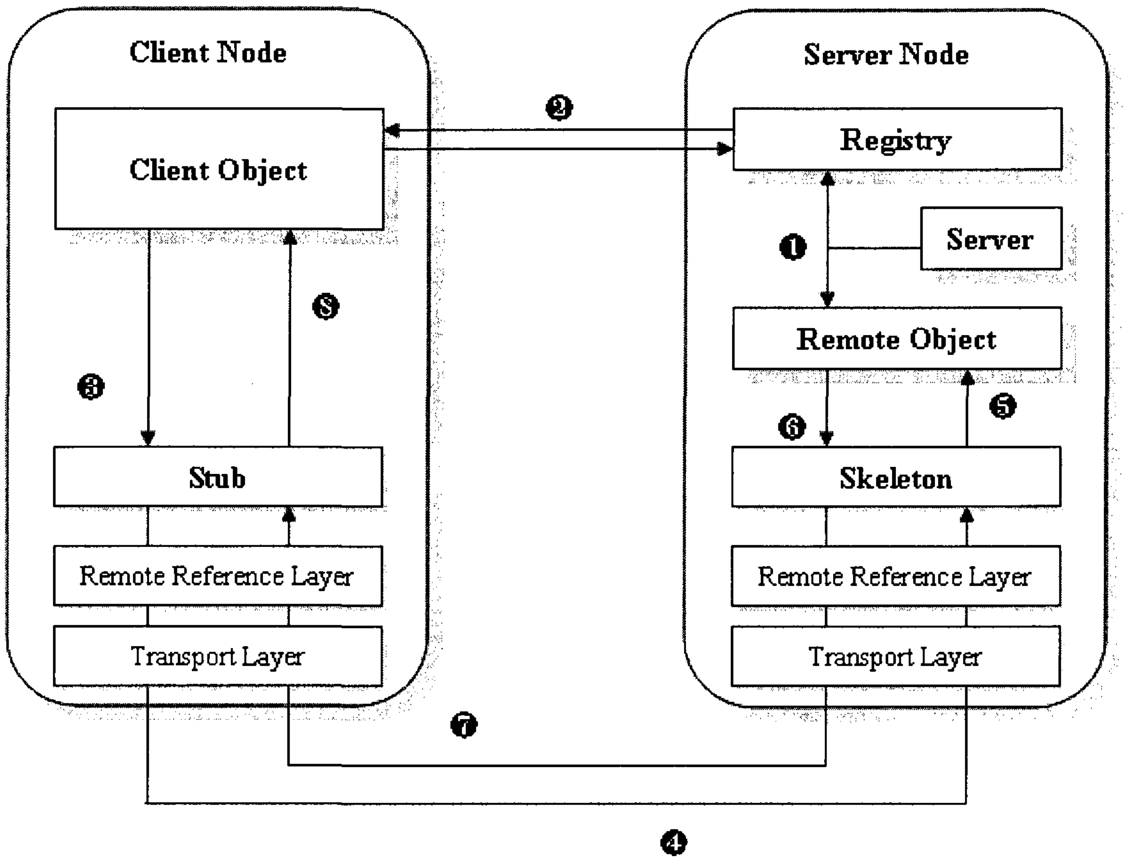


Figure 2-4: Java RMI Architecture

The basic procedure a client uses to communicate with a server is as follows: ❶ First a server creates a remote object and registers it to a local registry. ❷ The client obtains the reference of the remote object in the registry, and receives an instance of the local stub class. The stub class is transferred from the remote JVM, and automatically pre-generated from the target server class and implements all the methods that the server class implements. ❸ When the client invokes a method on the remote object, the method is actually invoked on the local stub. The stub marshalls all the information associated to the method call, including the name of the method and the arguments, and sends this information to the associated skeleton on the server side ❹. ❺ The skeleton demarshalls the data and makes the method call on the actual remote object. The remote object executes the method and passes the return value back to the skeleton ❻, the skeleton marshalls the return value, and sends it to the associated client-side stub ❼. At last, the stub demarshalls the return value and passes it to the client object ❽.

2.3.2 Implementation Details

2.3.2.1 Server Side

One of the requirements for a server process to be visible to a client object is that the server must implement the java.rmi.Remote interface. Any methods which are intended to be called by a remote object must be placed in an interface that extends the java.rmi.Remote interface. That interface must be implemented by the class whose methods will be called remotely. In addition, each method that will be called remotely must fulfill the following requirements:

(1) Must include the exception java.rmi.RemoteException (or one of its super classes such as java.io.IOException or java.lang.Exception) in its throws clause, in addition to any application-specific exceptions (application-specific exceptions do not have to extend java.rmi.RemoteException).

(2) A remote object declared as a parameter or return value (either declared directly in the parameter list or embedded within a non-remote object in a parameter) must be declared as the remote interface and not the implementation class of that interface.

Furthermore, a server class is required to implement an interface that extends the java.rmi.Remote interface, by extending the java.rmi.server.UnicastRemoteObject class. By extending the UnicastRemoteObject (in the java.rmi.server package) the class is given access to the remote behavior of both, the java.rmi.server.RemoteObject and java.rmi.server.RemoteServer. A server must also bind its unique name to the RMI registry, allowing clients to be able to "find" the server through the RMI registry. Once the server code is completed, the code must be compiled with the RMI compiler. By doing this, the skeleton code for the server is generated. The skeleton code handles all of the underlying networking needs of the communication. This includes, but is not limited, to setting up a connection, accepting the marshalled method invocation and potentially accompanying parameters and sending a response.

2.3.2.2 Client Side

A client can send a reference to the server by using the *java.rmi.Naming* class. The java.rmi.Naming class also provides access to services such as binding (already mentioned for the server process) unbind, lookup and listing the name-object pairings maintained on the host. Upon completion of the client code, the code must be compiled with the RMI compiler, thus generating the client stub code. The client stub code is used to send the marshalled messages to the server process, to receive and demarshall the response from the server.

2.4 Formal Concept Analysis

Formal Concept Analysis is a mathematical technique that provides insights into binary relations. It is a branch of lattice theory that provides a way to identify maximal groupings of objects that have common attributes [WIL81]. The mathematical foundation of formal concept analysis was laid by Birkhoff in 1940 [BIR67]. It is now gaining wide acceptance and has been applied to various application domains, such as to evaluate class hierarchies, explore configuration structures of preprocessor statements, for redocumentation, and to recover components.

2.4.1 Definitions

Formal concept analysis is based on a relation $\mathcal{R}$ between a set of objects O and a set of attributes $\mathcal{A}$, hence $\mathcal{R} \subseteq O \times \mathcal{A}$. The triple $C = (O, \mathcal{A}, \mathcal{R})$ is called formal context. For a set of objects $O \subseteq O$ the set of common attributes σ is defined as:

$$\sigma(O) \equiv \{a \in \mathcal{A} \mid \forall o \in O : (o,a) \in \mathcal{R}\}$$

Analogously, the set of common objects τ for a set of attributes $A \subseteq \mathcal{A}$ is defined as:

$$\tau(A) \equiv \{o \in O \mid \forall a \in \mathcal{A} : (o,a) \in \mathcal{R}\}$$

The mappings are antimonotone:

$$O_1 \subseteq O_2 \Rightarrow \sigma(O_2) \subseteq \sigma(O_1)$$

$$A_1 \subseteq A_2 \Rightarrow \tau(A_2) \subseteq \tau(A_1)$$

and extensive:

$$O \subseteq \tau(\sigma(O)) \quad \text{and} \quad A \subseteq \sigma(\tau(A))$$

To illustrate concept analysis, we use the binary relation between a group of stars and their characteristics shown in Table 2-1 as an example.

Context $(\mathcal{O}, \mathcal{A}, \mathcal{R})$								
		Attributes $\mathcal{A}$						
		small	medium	large	near	distant	moon	no moon
Objects $\mathcal{O}$	Merkur	×			×			×
	Venus	×			×			×
	Earth	×			×		×	
	Mars	×			×		×	
	Jupiter			×		×	×	
	Saturn			×		×	×	
	Uranus		×			×	×	
	Neptune		×			×	×	
	Pluto	×				×	×	

Table 2-1: An example relation table [LIN00] (a characterization of stars)

In the example, the objects are the different kinds of stars; the attributes are the characteristics small, no moon, etc. An object star has attribute characteristic if row *i* and column *j* is marked with a ×. For this relation table, the following equations hold:

$$\sigma\ (\{\text{Merkur}\}) = \{\text{small, near, no moon}\}$$

$$\tau\ (\{\text{distant, moon}\}) = \{\text{Jupiter, Saturn, Uranus, Neptune, Pluto}\}$$

A pair (O, A) is called **concept** if $A = \sigma(O) \wedge O = \tau(A)$ holds, i.e., all objects share all attributes. For a concept $c = (O, A)$, O is the extent of c, denoted by *extent*(c), and A is the intent of c, denoted by *intent*(c). Informally, a concept corresponds to a maximal rectangle of filled table cells modulo row and column permutations. In the example, ({Earth, Mars}, {*moon, small, near*}) is a concept, whereas ({Earth, Pluto}, {*moon, small*}) is not a concept: σ ({Earth, Pluto}) = {*moon, small*}, τ ({*moon, small*}) = {Earth, Mars, Pluto}. The following table contains the concepts for the relation in Table 2-1.

Concept 1	({}, {*moon, medium, distant, small, large, near, nomoon*})
Concept 2	({Pluto}, {*moon, distant, small*})
Concept 3	({Uranus, Neptune}, {*moon, medium, distant*})
Concept 4	({Merkur, Venus}, {*small, near, nomoon*})
Concept 5	({Jupiter, Saturn}, {*moon, distant, large*})
Concept 6	({Jupiter, Saturn, Uranus, Neptune, Pluto}, {*moon, distant*})
Concept 7	({Earth, Mars}, {*moon, small, near*})
Concept 8	({Earth, Mars, Pluto}, {*small, moon*})
Concept 9	({Merkur, Venus, Earth, Mars}, {*small, near*})
Concept 10	({Merkur, Venus, Earth, Mars, Pluto}, {*small*})
Concept 11	({Earth, Mars, Jupiter, Saturn, Uranus, Neptune, Pluto}, {*moon*})
Concept 12	({Merkur, Venus, Earth, Mars, Jupiter, Saturn, Uranus, Neptune, Pluto}, {})

Table 2-2: The Concepts for Table 2-1

The set of all concepts of a given formal context forms a partial order via:

$$(O_1, A_1) \leq (O_2, A_2) \Leftrightarrow O_1 \subseteq O_2$$

or equivalently with

$$(O_1, A_1) \leq (O_2, A_2) \Leftrightarrow A_1 \supseteq A_2$$

If $c_1 \leq c_2$ holds, then c_1 is called a **subconcept** of c_2 and c_2 is called **superconcept** of c_1.

For instance, ({Jupiter, Saturn}, {*moon, distant, large*}) is a subconcept of ({Jupiter, Saturn, Uranus, Neptune, Pluto}, {*moon, distant*})

2.4.2 Concept Lattice

The set L of all concepts of a given formal context and the partial order $\preceq$ form a complete lattice, called **concept lattice**:

$$L(C) = \{(O, A) \in 2^{O} \times 2^{\mathcal{A}} \mid A = \sigma(O) \wedge O = \tau(A)\}$$

Concept lattices are usually visualized as hierarchical graphs, often with non-redundant labeling (presents each object and each attribute only once) to improve their readability. Each node represents a different concept. The node with an attribute $a \in \mathcal{A}$ represents the most general concept that has a in its intent, called the **top element**; on the other hand, the node with an object $o \in O$ represents the most specific concept that has o in its extent, called the **bottom element**. Figure 2-5 shows an example of concept lattice graph derived from the previous context table in Table 2-2. In this figure, the bottom element, Concept 1, ({}, {*moon, medium, distant, small, large, near, nomoon*}) contains the empty set of objects coupled with all the attributes. The top element, Concept 12, ({Merkur, Venus, Earth, Mars, Jupiter, Saturn, Uranus, Neptune, Pluto}, {}) includes an empty set of attributes coupled with all the objects.

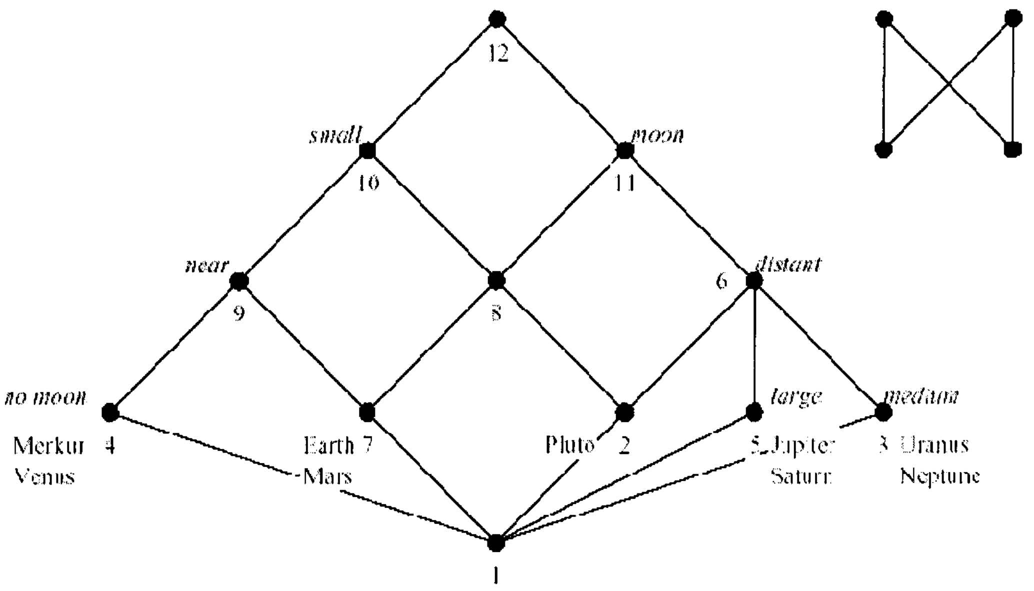


Figure 2-5: The concept lattice for the Table 2-2 (stars and their characters)

The content of a node *N* in this representation can be derived by passing attributes down and passing objects up [LIN00]:

- the objects of *N* are all objects at and below *N*,
- the attributes of *N* are all attributes at and above *N*.

For example, to read Concept 8, one must pass all the objects from the lower level up to Concept 8, and one will get {Earth, Mars, Pluto} as the object list of Concept 8. Then, one must pass all the attributes from the upper levels down to Concept 8, and one will get {*small, moon*} as the attribute list of Concept 8. Therefore, Concept 8 represents ({Earth, Mars, Pluto}, {*small, moon*}) which corresponds to the list of concepts in Table 2-2.

2.5 Regression Testing

2.5.1 Overview

Throughout the software life cycle, the cost of maintenance activities dominates the overall cost of a software product. A significant part of this maintenance cost is spent on testing to be performed after a modification request was performed. Among the different types of testing performed during maintenance, regression testing plays an important role. Regression testing attempts to validate modified software to ensure that no new errors are introduced into previously tested code [HAR00].

Regression testing can be defined as the process of reusing (parts of) a test suite that was used for testing the original version of the software. One approach to regression testing is to rerun all test cases in the test suite. However, due to the size of software products and the associated size of the test suite, re-executing an entire test suite may require days or even weeks. Therefore, retest-all approach are considered often too expensive, especially when only a small portion of a system was modified as part of a performed maintenance request. Due to the substantial effort associated with the re-test all approach, kinds of regression test selection techniques have been developed to reduce the cost of regression testing [ROT96].

Regression test selection (RTS) techniques attempt to reduce the testing cost by selecting and running only a subset of the test cases from a program's existing test suite, exercise the software entities that have been changed or are most likely to be affected by the change [GRA01]. Rothermel and Harrold provide the following selective retest strategy for regression testing [ROT96]. The strategy is presented as a solution to the following problem:

> Given program P, its modified version P', and test set T used previously to test P. Find a way of making use of T, to gain sufficient confidence in the correctness of P'.

(1) Select $T' \subseteq T$, a set of tests to execute on P'.

(2) Test P' with T', to establish the correctness of P' with respect to T'.

(3) If necessary, create T'', a set of new functional or structural tests for P'.

(4) Test P' with T'', to establish the correctness of P' with respect to T''.

(5) Create T''', a new test suite and test history for P', from T, T', and T''.

2.5.2 Regression testing selective techniques

A significant body of existing work on regression test selection exists in the literature. These regression testing techniques can be differentiated by their programming languages support, e.g., procedural languages [ROT97], object-oriented languages [HAR01B], aspect-oriented languages [XU07], as well as by the type of applications they support, e.g., COT-based applications [ZHE06], database-driven applications [HAR04]. In general, RTS approaches can be used for code or specifications. Specification-based selection techniques focus on changes at the specification level. They are independent of the code, but require properly-maintained specifications. Code-based selection techniques are based on the available source code. They record the program elements exercised by the tests during previous releases and select based on the existing information, the test cases that exercise elements changed in the current release. A number of code-based selection techniques focus on different programming elements: control-flow [ROT97], data-flow [HAR89], program slices [GUP96], firewall concept [WHI92], and they operate at different granularity levels, such as fine-grained [ROT97] or coarse-grained [CHE94].

[GRA01] presents a typical classification of regression test selection techniques. In terms of the testing goal, RTS approaches are grouped into five families: *Retest-All Technique*, *Ad Hoc/Random Techniques*, *Minimization Techniques*, *Dataflow Techniques, Safe Techniques*. Among them, the later three methods are often used in practice. Dataflow-coverage-based techniques select tests that cover those program entities, which are modified or affected by modifications. Like the dataflow methods, minimization techniques are fundamentally coverage based analysis approaches. However, the minimizations techniques attempt to identify a minimal set of tests from the set of all test cases T. If the coverage of two test cases is exactly the same, the new test suite will only keep one of them. Both minimization and dataflow techniques are not designed to be safe, and they can fail to select a test case that would have revealed a fault in the modified program. In contrast, safe RTS techniques make certain that they will not omit any test cases in the original test suite T that can reveal faults in P'.

In what follows, we provide a more detailed review of some of these regression test selection techniques.

- [ROT97] proposes a safe regression test selection technique which supports statement level analysis for procedural programs. In this approach, a *Control flow graph* (CFG) is used as program representation to select tests, which contains nodes for each simple or conditional statement, and edges between nodes representing the flow of control between statements. At first, a CFG is established for the original program, a test history table is also constructed to record which test cases correspond to each traversed edge in the CFG. In the following, another CFG is built for the modified program. Then simultaneous traversals are performed on two CFGs through each node and edge to identify the difference between them; and the test cases in the history table being related to the changed entities are selected. After comparison is finished, system gathers all test cases that need to be rerun.

- Using the similar method introduced above to perform selective retesting for C program, the tool TestTube [CHE94], on the other hand, employs relatively coarse-grained analysis of the system under test, and produces a reasonable and practical tradeoff between granularity of analysis and time/space complexity. A technique for safe regression test selection for Java programs is described in [HAR01B]. This technique is an adaptation of the method of [ROT97], which uses *Java Interclass Graph* (JIG) extended of CFG to explicitly represent various specific features in Java programs and then detect dangerous arcs on it. Based on the research reported by

[HAR01B], [ZHA06] develop their approach for AspectJ programs, utilizing *AspectJ Inter-module Graph* (AJIG). With this control-flow representation, they determine a set of dangerous AJIG edges corresponding to semantic source-code-level changes made by a programmer.

- An incremental testing system that can also be used for regression testing is described in [HAR89]. This tool leverage incremental data flow analysis to provide reuse of original test cases. Data flow analysis employs definitions and uses of variables to compute def-use associations. Uses are classified as either *computation* uses (c-uses) or *predicate* uses (p-uses). A c-use occurs whenever a variable is used in a computation statement, and a p-use occurs whenever a variable is used in a conditional statement. During the initial testing, the system stores the previous data flow analysis result and test cases. After program changes, the system analyzes the effect of the modifications on the test history, and reruns the test cases traversing every definition-use pair that is deleted from the original program, new in the changed program, or modified for the changed program.

- [GUP96] presented an approach to data flow based regression testing that uses slicing algorithms to explicitly detect definition-use associations that are affected by a program change. The slicing algorithms include backward and forward walk algorithms, both of them require no past history of data flow information. Given a program point, the backward walk algorithm identifies statements containing definitions of variables that will affect the point when the program execution reaches at it; the forward walk algorithm identifies uses of variables that are directly or indirectly affected by either a change in a value of a variable at the program point or a change in a predicate. This approach does not need to maintain a test suite and also can achieve the same testing coverage as a complete retest of the program with respect to the affected definition-use associations.

- A testing firewall methodology for regression testing has been developed by White and Leung in [WHI92], which considers both control-flow dependencies and data-flow dependencies. The firewall concept is defined to represent affected areas that include changed modules and all other modules affected by them. When one program entity is changed, then all test cases being related to it and to other entities in its "firewall" will be identified and re-executed.

3. Research Contributions

In this chapter, we present our research contribution, a relative lightweight regression test selection approach for Java RMI applications. The section will introduce the motivation behind this research, its research hypothesis, research goals and the approach developed to address the problem.

3.1 Motivation

The goal of regression testing is to ensure that the modified software still satisfies its intended requirements. Due to the cost associated with regression testing, regression test selection (RTS) techniques can be applied to reduce the overall cost for re-running test cases. A variety of RTS techniques have been introduced for many kinds of programs, such as procedural programs [CHE94, ROT97, WHI92], object-oriented programs [HSI97, ROT00, WHI97, HAR01B], and aspect-oriented programs [XU07, ZHA06]. RTS methods are also being applied to component-based [MAO05] and COTS-based applications [ZHE06]. However, most of these approaches focus on sequential programs, with none of them providing support for regression test selection for distributed programs (such as Java RMI applications).

Firstly, identifying distributed code entities (e.g. classes, methods), which are exercised by a particular test case is one of the key issues for RTS. Using dynamic analysis can provide a more complete and reliable analysis, and it can be achieved by using traces that correspond to the actual execution paths of the test cases through the code [SNE04]. As introduced earlier, Java RMI applications distribute objects and execute across different hosts. The methods of remote Java objects can be invoked from other Java virtual machines on different hosts. These features of Java RMI make it difficult to establish the relation table between test cases and program components that is typically required for regression test selection approaches. In this case, all individual execution events from multiple machines need to be collected separately and then merged together properly into a single, complete trace for an entire application. There exist some tools for profiling Java RMI applications, such as Jinsight [PAU00], JaViz [KAZ00], VisOK [LEE00], JRastro [SIL03]. However, some of these tools do not provide a sufficient level of detail to allow for a more detailed analysis of the traces, and/or they are closed source (commercial) tools, which cannot be customized.

Secondly, existing RTS approaches rely typically on one or several of the following information resources: control flow information [ROT97], data flow information [HAR89] or the firewall concept [WHI92] to identity which test cases are associated with modifications. Nevertheless, these techniques only consider internal data states (program state) to select test cases, external data states (such as databases, files) are not considered in their analyses. For many applications working with databases or files, this omission could lead to scenarios where test cases affected by the modifications might not be selected and re-executed.

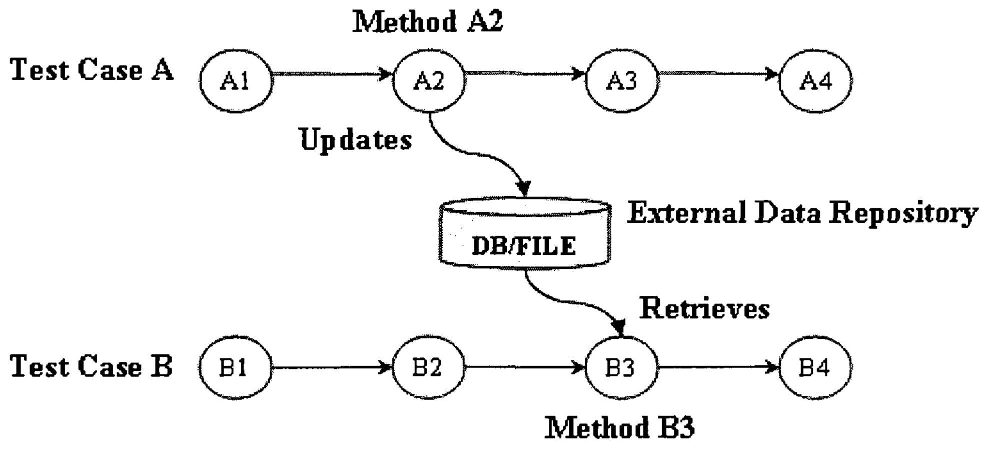


Figure 3-1: Test cases dependency on sharing external data states

As depicted in Figure 3-1, test case A, method A2 updates an external data state; while for test case B, method B3 retrieves the same data state. As a result one can observe that there exists a write/read access between test case A and B, and in case of a modification to method A2, both scenarios, test case A and B should be retested due to the data dependency existing between the methods A2 and B3. However, most of the existing regression testing techniques will fail to include test to cover these external data dependencies.

There are several approaches addressing RTS for database-driven applications by using database states [HAR04, WIL05]. However, common to these techniques is that they rely on complete statement level instrumentation. Therefore, they require the recording of huge amounts of data in the execution traces, making the analysis of these traces expensive and often not feasible.

Furthermore, in many cases maintainers or project managers might want to perform an initial (more lightweight) RTS prior to actually performing the modification, to identify an estimate of the testing effort associated with the modification. This information might be applicable to determine the level of testing and evaluation required, and the estimated management cost to implement a modification request. We refer the reader for more details on applying modification analysis activities to the IEEE maintenance standard [ISO/IEC 14764:2006(E) IEEE Std 14764-2006].

Given these limitations and restrictions of existing tools and approaches in tracing and performing regression test selection analysis for Java RMI programs, we decided to implement our own tracing tool to collect the corresponding execution data and to perform our own regression test analysis for Java RMI programs

3.2 Research Hypotheses and Research Goals

3.2.1 Research Hypotheses

In this research we present a lightweight regression test selection approach for Java RMI applications that combines both execution trace collection and external data sharing analysis. In particularly, we focus on estimating the potential testing efforts involved in a change request during modification analysis. Our research hypothesis can therefore be defined as follows:

Research Hypothesis:

A methodology can be developed to collect traces from distributed Java RMI applications that allows for performing a lightweight regression test selection analysis on these traces during the modification request analysis.

We expect our research hypothesis to hold if the following acceptance criteria can be validated:

1. Automated tracing of distributed Java programs implemented using RMI

There exist a number of tracing approaches, which mainly focus on profiling sequential programs [LEE97, GOL03, SEE05, SYS01]. As state earlier, to perform regression test selection for RMI based distributed Java applications, execution traces from these distributed systems have to be collected. However, tracing of distributed systems is more complex and requires the tracing environment itself to be distributed, to allow for data collected not only within an individual node but also from the distributed nodes.

In the literature, several approaches for tracing distributed Java RMI programs have been proposed [KAZ00, GHO02, BRI05, CHE04]. After execution data from each individual machine are captured, the data has to be transferred to a centralized repository. In [ZOL04], different modes of trace transportation are described. In the local mode approach, interceptors write the collected information (including timestamps) to local files, and these files are later merged in one tracing file. An example implementation for the local mode approach is JaViz [KAZ00]. For the buffer mode approach, events are buffered locally in each component's name space, and then propagated to the center. The communication may either be arranged using the common channels or through dedicated channels. An example for the buffer mode approach is VisOK [LEE00].

Merging both local execution data and remote invocation between components, can be achieved by matching the corresponding entries on the server and client profiles. One approach addressing this issue adopts a similar method as JaViz, which records unique identifier for remote objects and methods, the machine names of client and server, as well as the client-side TCP/IP connection port number to support a consistent merging of the execution traces. Rather than developing a stand-alone tracing tool we plan to integrate our tracing tool as an Eclipse plug-in within the Eclipse IDE. Based on the existing work on tracing sequential and distributed programs we anticipate that a RMI based Eclipse plug-in tracing tool can be developed.

2. Implement a lightweight regression test selection method for distributed Java RMI applications to estimate the testing effort involved prior to performing a program change.

RTS techniques have been applied previously to verify that the applications still complies with its specified requirements after a program change. As part of this research we focus on the analysis of applying selective regression testing technique to provide some guidance in estimating the potential testing effort involved during the modification request analysis. The goal is to provide decision makers with some guidelines with respect to the number of test cases that have to be retested, prior to actually performing the modification.

A number of regression test selection techniques that use dynamic system traces to build coverage matrices between test cases and program entities (e.g. statements, methods, classes, or modules) have been proposed for procedural, object-oriented and aspect-oriented applications. However, most of these test selection approaches focus on identifying test cases to be re-run for sequential applications. This is due to the fact that their underlying tracing approach is limited to the collection of runtime communications within components, and do not examine causality relationship between local invocations and remote calls. Therefore these methods are typically not suitable for analyzing distributed systems such as RMI based programs.

Moreover, existing RTS approaches have mainly focused only on the change propagation through the internal program state (i.e. variables) manipulation, and do not consider change impacts involving persistent states (i.e. databases, files). Although several papers [HAR04, WIL05] have addressed RTS for database applications, these approaches are typically heavy weight approaches, requiring fine grained traces at the statement level, making them very precise but also computational expensive.

Based on the above criteria, we can then also define our Null-Hypothesis when to reject our research hypothesis.

Null-Hypothesis:

The research hypothesis will be rejected if it is not possible to collect a consistent set of distributed Java RMI applications traces or no tool can be developed to support a selective regression testing analysis for these traces.

3.2.2 Research Goals

In what follows we further refine the research hypothesis to specify our primary research goals as follows:

Research-Goal 1:

Develop an Eclipse Plug-in to trace and collect run-time information of distributed Java RMI applications at different levels of granularity, including external data states, without the need for any major user involvement.

This research goal can be further decomposed into several sub-goals:

1. **Define a methodology for tracing and merging trace information**

   For tracking dynamic behaviour of distributed Java RMI programs, the methodology is required not only to trace local calls within a node, but also to capture remote method invocations between different machines. It should allow to specify different levels of granularities (i.e. function-level, class-level) at which the information is collected, as well as to select which component (i.e. a method with a specific name) in an execution to be monitored. Moreover, the approach should not require the user to manually modify any source code for the collection of the execution traces.

2. **Develop an Eclipse plug-in to automate the tracing environment for distributed Java RMI applications**

   The plug-in should support the extraction and merging of execution information from different running nodes as well as associate these traces to the execution of specific test cases. Furthermore, messages of remote method invocations and external data states (e.g. databases, files) access must also be intercepted and included as parts of the collected tracing information.

Research-Goal 2:

Apply execution traces to support a lightweight regression test selection approach.

This research goal can be further decomposed into several sub-goals, they are:

1. **Apply Formal Concept Analysis (FCA) to support a lightweight RTS analysis**

   Formal Concept Analysis (FCA) is capable to perform sensible grouping of objects that have common attributes, and helps extract dependency information. Using test cases as the objects and execution trace elements that are executed by a particular set of test cases as the attributes, an *execution dependency lattice* resulting from FCA can identify all the test cases that execute a particular software component, and then can be used to estimate the test cases that should be rerun after the software change is made.

2. **Include the external data state analysis to improve the FCA based RTS analysis**

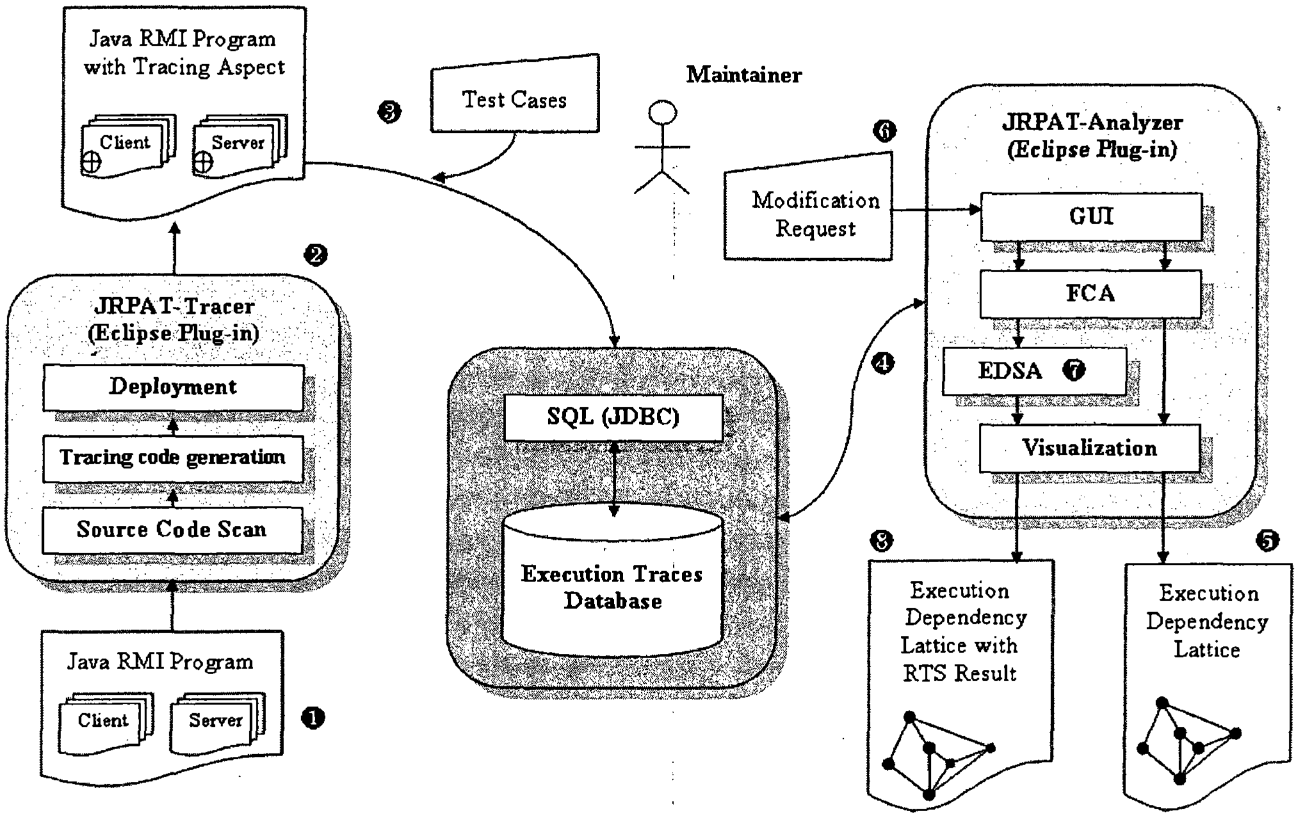
   The test case selection method is expected to enrich our previous FCA based RTS approach by taking into account external data sharing relationship among the program entities of different test cases. When program components of several test cases accessing the same external data, if one test case is selected to be retested, the others will be analyzed whether or not to be re-executed. Some test cases which are omitted by the previous approach will now be complementally added.

3. **Implement an Eclipse plug-in to evaluate the improved RTS analysis**

   Based on the execution traces, the plug-in should have the capability to establish an external data sharing table among system test cases, and use the table to conduct the proposed RTS method. It is also expected to support both textual information and graphical representations of the dependency structure between execution traces and test cases. Evaluate the approach through some initial case studies.

4. A Selective RTS Methodology for Distributed Systems

In what follows we introduce a general overview of our novel regression testing methodology for Java RMI programs. The methodology overview is followed by a more detailed description of the various parts of our methodology in the subsequent sections and subsections.



Java RMI Program with Tracing Aspect
Client
Server
❸
Test Cases
Maintainer
❻
JRPAT-Analyzer (Eclipse Plug-in)
Modification Request
GUI
FCA
❷
JRPAT-Tracer (Eclipse Plug-in)
Deployment
Tracing code generation
Source Code Scan
SQL (JDBC)
Execution Traces Database
❹
EDSA ❼
Visualization
❽
❺
Execution Dependency Lattice with RTS Result
Execution Dependency Lattice
Java RMI Program
Client
Server
❶

Figure 4-1: The RTS Methodology for Distributed Systems

The RTS methodology is briefly described in step ❶ to step ❽.

❶ – *Select a destination program*:

Select a Java RMI-based distributed program to be analyzed.

❷ – *Perform instrumentation on the destination program*:

Utilize the tracing plug-in (the JRPAT-Tracer) to instrument the Java RMI program on both client and server sides for collecting runtime data.

❸ – *Run the instrumented destination program with test cases*:

The execution traces are collected and stored in the server-side central database by the JRPAT-Tracer.

❹ – *Perform analysis on the tracing information*:

The analysis plug-in, the JRPAT-Analyzer, consists of three major components. (1) An External Data Sharing Analysis (EDSA) component to analyze the external states accessing. (2) A formal concept analysis (FCA) component to perform the logical grouping of the traces. (3) A visualization component to represent textual and graphical information. In this step, the JRPAT-Analyzer merges the client/server execution traces and uses them to build the external data sharing table for test cases.

❺ – *Visualize FCA result*:

The JRPAT-Analyzer invokes the FCA component to compute the FCA concepts and the relations among them, and then calls an external tool GraphViz to generate a graph file of the result. After that, the JRPAT-Analyzer visualizes the graph file, an execution dependency lattice, in its specific view.

❻ – *Input modification request*:

Given the execution dependency lattice from step ❺, a modification request can now be specified at the concept level.

❼ – *Conduct regression test case selection*:

The JRPAT-Analyzer performs the selective regression testing analysis and identifies the test cases that have to be potentially retested as part of the modification request.

❽ – *Visualize test case selection results*:

Both the changed node and the test cases that required re-testing are highlighted in the concept lattice.

4.1 Tracing Process

4.1.1 Instrumentation

For the instrumentation of Java RMI programs, we utilize AspectJ [KIC01], an extension of the Java language to support aspects. AspectJ instruments the bytecode of Java applications and thus does not require the modification of the source code. We selected AspectJ due to its additional flexibility, compared to other byte code instrumentation approaches during monitoring the program execution and logging of the trace information. It supports both the collection of trace information for classes, methods, packages and threads, as well as the collection of run-time objects and actual arguments. In addition, AspectJ allows for parameterization, to specify tracing which entities (e.g. classes, methods and packages), which interactions (e.g. non-static, static, constructor and remote calls) and which positions (e.g. before, after and around executions).

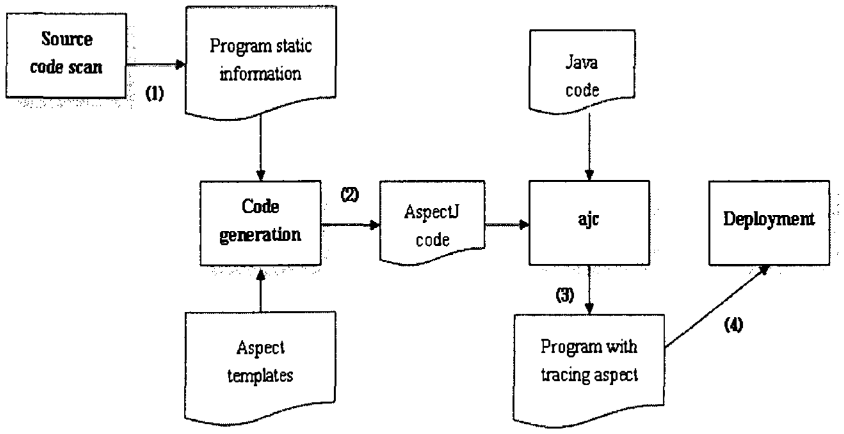


Figure 4-2: The instrumentation workflow

The instrumentation workflow is illustrated in Figure 4-2. **(1)** A fact extraction of source code is first performed to collect static information about the distributed Java RMI program analyzed (e.g. interface classes in which RMI remote methods are defined, package names, etc.). **(2)** Then, based on the derived source code information, the AspectJ tracing codes are generated and inserted into the destination project automatically. In the next step **(3)**, the Java codes and the AspectJ codes are compiled (weaved) to create the tracing enable version of the program. Those steps stated above are performed on both client and server sides. In the last step **(4)** the RMI system is deployed. The stub and skeleton classes required by RMI are automatically constructed on the server side, and the stub classes along with the interface classes are deployed from the server to the client through TCP/IP socket transfer.

4.1.2 Tracing Remote Invocations

The RMI middleware has previously been used to provide extended services for the intercept of remote invocations [CHE04]. In what follows we use AspectJ for the interception of these remote procedure calls. As description in Section 2.3, a remote method call includes the invocation to the reference remote method of stub instance on the client side, and the invocation to the real remote method on the server side. Leveraging AspectJ, both the client and server-side information of a remote call can be collected separately. However, the collected information is not sufficient to establish a mapping between the server-side and the client-side tracing records. We remedy this problem by exchanging the remote invocation record between the client and the server to establish the traceability among them (Figure 4-3). The invocation record on the client (i.e. the sender host name, the name of the method invoked, etc.) is transmitted to the server, and also the server-side method call information (i.e. the receiver host name, the receiver class name, etc.) is sent to the client.

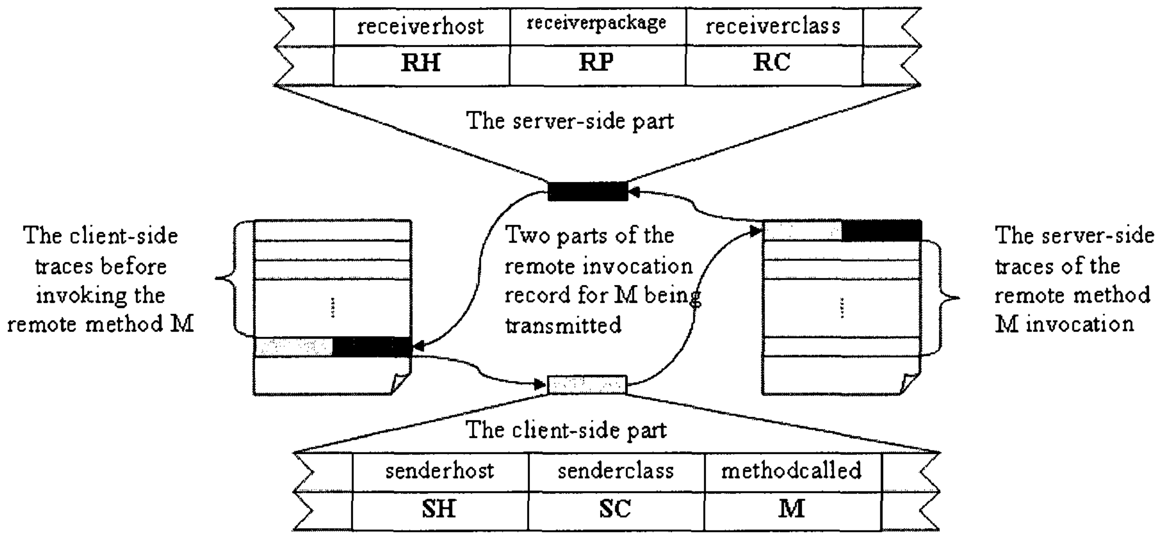

receiverhost
receiverpackage
receiverclass
RH
RP
RC
The server-side part
The client-side traces before invoking the remote method M
Two parts of the remote invocation record for M being transmitted
The server-side traces of the remote method M invocation
The client-side part
senderhost
senderclass
methodcalled
SH
SC
M

Figure 4-3: Exchanging the remote invocation records between Client and Server

4.1.3 Tracing External Data States

In database-driven applications, program components typically utilize different external data state (elements). This is in particular of interest for scenarios such like when one test case involves a write access to some external data element and another test case performs a read access to the same external data element. During the RTS analysis, there is a need to identify these often indirect data dependencies with external data elements to determine the appropriate set of test cases that have to be re-run after a modification request.

Through AspectJ, we are able to trace external data access information by monitoring the corresponding access operations in the source code at runtime. Java provides several classes in the "java.io" package for file handling shown in Table 4-1. These classes can be monitored to generate the file sharing relation among test cases.

class **FileInputStream**	A file input stream obtains input bytes from a file.
class **FileOutputStream**	A file output stream for writing data to a file.
class **FileReader**	Convenience class for reading character files
class **FileWriter**	Convenience class for writing character files.
class **RandomAccessFile**	Instances of this class support both reading and writing to a random access file.

Table 4-1: Java file handling classes in the "java.io" package

The following classes (Table 4-2), being parts of the "java.sql" package, are designed for database processing, and hence we can trace these classes to establish the database sharing relation among test cases.

interface **Statement**	An object used for executing a static SQL statement and returning the results it produces.
interface **PreparedStatement** (extends Statement)	An object that represents a precompiled SQL statement. The SQL statement is precompiled and stored in a `PreparedStatement` object, and the object can then be used to efficiently execute this statement multiple times.
interface **ResultSet**	A table of data representing a database result set, which is usually generated by executing a statement that queries the database.

Table 4-2: Java database processing classes in the "java.sql" package

4.2 Selective Regression Testing Analysis

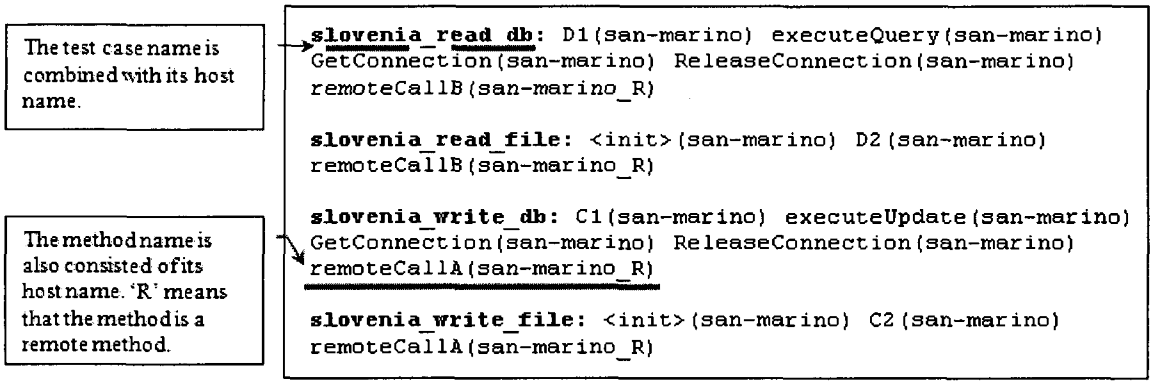
The RTS analysis presented in this research extends a previous FCA-based regression test selection approach [PAB06] to distributed programs (especially for Java RMI applications), and complements it with external data states analysis to provide the ability to estimate testing effort more precisely prior to performing a modification request.

4.2.1 Combining RMI traces with FCA

In the existing FCA-based regression test selection approach [PAB06], an *execution dependency lattice* is generated to represent the test case coverage based on runtime traces collected. In this concept lattice, test cases are objects and the execution traces accounting for each test case are their attributes. Given the lattice representation one can identify which test cases execute which software component. Starting from the node (represents a method exercised by test cases) to be modified; we can traverse the execution dependency lattice downward and identify all the reachable leaf nodes (represent test cases) and therefore the associated test cases that need to be retested.

In this research, we perform the FCA-based regression test selection analysis on Java RMI programs. Runtime data of the distributed Java applications is collected from multiple hosts and transmitted to the central database. The collected execution traces for each test case are then merged to provide the input for the *execution dependency lattice.* In the lattice, methods and test cases can be identified by their unique names, which are created by combining the test case names and the name of the host they were executed on.

In Figure 4-4, the FCA context contains four execution traces captured from a Java RMI distributed program.


The test case name is combined with its host name.
The method name is also consisted of its host name. 'R' means that the method is a remote method.

```
slovenia_read_db: D1(san-marino) executeQuery(san-marino)
GetConnection(san-marino) ReleaseConnection(san-marino)
remoteCallB(san-marino_R)

slovenia_read_file: <init>(san-marino) D2(san-marino)
remoteCallB(san-marino_R)

slovenia_write_db: C1(san-marino) executeUpdate(san-marino)
GetConnection(san-marino) ReleaseConnection(san-marino)
remoteCallA(san-marino_R)

slovenia_write_file: <init>(san-marino) C2(san-marino)
remoteCallA(san-marino_R)
```

Figure 4-4: The FCA context of a sample Java RMI program

In the context, test cases are the *objects* and the methods in execution traces accounting for each test case are the *attributes*. In this example, test cases are started on the client *slovenia*, and they invoke remote methods (i.e. `remoteCallA`, `remoteCallB`) on the server *san-marino*. Figure 4-5 shows the resulting concept lattice generated from the FCA context.

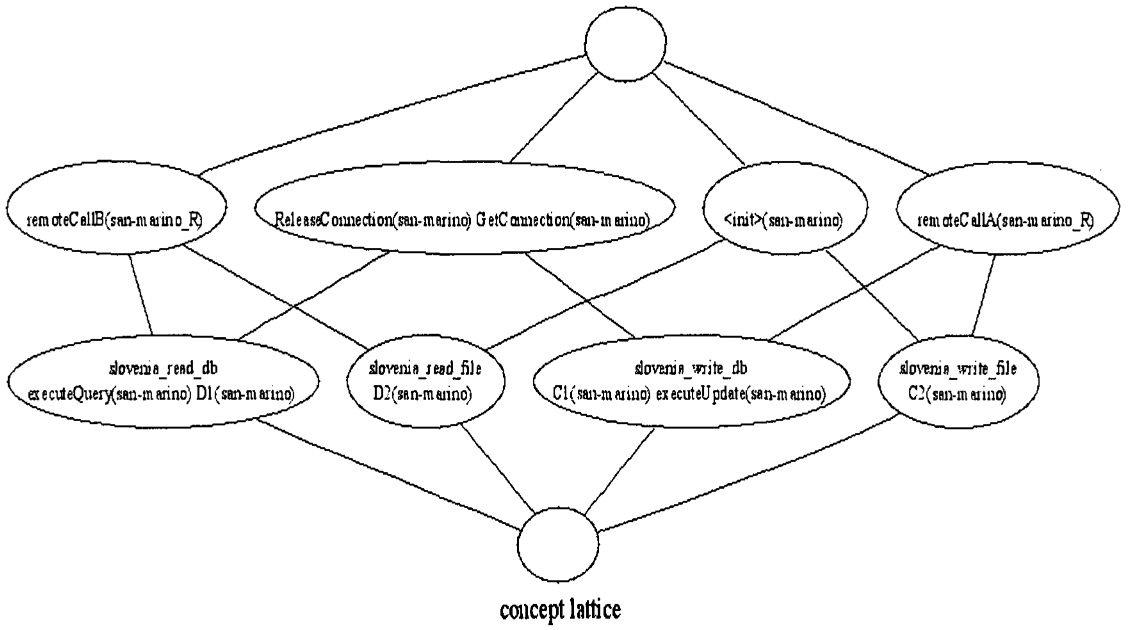

remoteCallB(san-marino_R)
ReleaseConnection(san-marino) GetConnection(san-marino)
<init>(san-marino)
remoteCallA(san-marino_R)
slovenia_read_db
executeQuery(san-marino) D1(san-marino)
slovenia_read_file
D2(san-marino)
slovenia_write_db
C1(san-marino) executeUpdate(san-marino)
slovenia_write_file
C2(san-marino)
concept lattice

Figure 4-5: The concept lattice of a sample Java RMI program

4.2.2 External Data Sharing Analysis combined with FCA

A limitation of our previous approach is that it does not consider external data flow; the define-use relationship that might exist among program components executed by different test cases. Therefore, it might ignore some test cases that need to be rerun based on a particular change. To overcome the limitation of the former FCA-based approach, the improved regression test case selection methodology will extend the approach through the use of external data sharing as follows:

Build the external data sharing table among test cases based on execution traces gathered. Analyze the retesting test cases selected by the FCA-based RTS method with the external data sharing table, identify all other test cases which have define-use relationship with the FCA selected test cases, and add them into the retesting list.

Next we present our lightweight RTS approach that leverages external data sharing relations to further refine the RTS analysis. The approach considers each file or table in a database as a variable, and traces their usages. External data in files are normally accessed through some type of *read* or *write* access. We refer to them as *r-use* and *w-use* operations. For the external data in database tables we concentrate on SQL-based systems and identified the following four main access strategies: *select*, *delete*, *insert*, and *update*. *Select* usage is used to retrieve data from tables and is denoted as *r-use*. *Delete*, *insert*, and *update* usages are employed to modify data in tables and are denoted as *w-use*. If both *read* and *write* operations are performed to the same external data, then the usage of the data is denoted as *rw-use*. Test case A will affect test case B only if A modifies (*w-use* or *rw-use*) an external data that is retrieved (*r-use* or *rw-use*) by B.

Test Cases / **External Data**	T1	T2	T3	T4	**T5**
D1	R	R	R		
D2		W		R	W
D3			*RW*	*W*	
D4	W				R

Table 4-3: An example external data sharing table

Table 4-3 illustrates such an external data sharing table. In the example, T5 is initially identified as a test case that needs to be re-executed, it reads (*r-use*) D4 and writes (*w-use*) D2. In the next step a further analysis is performed, since T5 may affect other test cases accessing the same external data D2. T2 and T4 all use D2, with T2 has write access (*w-use*) on D2, and T4 has read access (*r-use*) on D2. Based on the above definitions, only T4 has define-use relationship with T5 and therefore needs also to be retested. We can define this external data sharing analysis more formally as follows:

Given is d_k, an external data shared by program components executed by different test cases. We use a notational convention *usage(d_k)* to denote the usage of the external data d_k, and its value is defined as the table below:

The usage of d_k	**The value of *usage(d_k)***
Empty	ϕ
r-use	0
w-use	1
rw-use	2

Table 4-4: The usages and corresponding values of an example external data

Assume that t'_j is a test case selected to be retested; t_i is a test case being checked to see whether it is affected by t'_j. Then for an external data d_k, if the following equation holds:

$$(t_i.usage\ (d_k)\neq \phi\ )\cap(\ t'_j.usage(d_k)\neq \phi\ )\cap(\ t_i.usage(d_k)\neq 1\ )\cap(\ t'_j.usage(d_k)\neq 0\ ) = 1$$

There exists a define-use relationship between the two test cases, t_i is affected by t'_j and also need to be re-executed.

Note that a define-use relationship among test cases is transitive. Whenever a modification is made to one test case, this transitivity can result in a ripple effect. For example, in Table 4-3, test case T4 is selected as being affected by the selected test case T5. As part of the ripple effect analysis, we have to check now whether there exists another test case which has define-use relationship with T4. Since T4 writes (*w-use*) D3 and test case T3 reads and writes (*rw-use*) D3, T3 is also added to the retesting list. Then we need to continue examine T3, T3 will not affect any other test cases because it only has two *r-use* usages of external data. Till now, the analysis for the effect of T5 is finished.

Let T be the original test suite the program under test. Let $T'(T'\subseteq T)$ be a set of test cases which are selected to be retested. A typical regression test selection through external data sharing analysis proceeds as follows.

(1) Select a set of test cases $T1$, $T1 \in T$ but $T1 \notin T'$.

(2) Analyze $T1$ and T' with the equation discussed above to find out $T1'$, a set of test cases in $T1$ that are affected by T', add $T1'$ to T'.

(3) Select a set of test cases $T2$, $T2 \in T$ but $T2 \notin T'$.

(4) Analyze *T2* and *T1'* with the equation discussed above to find out *T2'*, a set of test cases in *T2* that are affected by *T1'*, add *T2'* to *T'*.

The pseudo code for the algorithm performing the RTS approach is shown as Figure 4-6.

```
Algorithm: PerformEdsaRTS
Input: The RTS result list selected by the FCA-based approach
Output: The complemented RTS result list including both the result of FCA and the result of
External Data Sharing Analysis (EDSA)

Denote TFCA to be the list of test cases which are selected by the FCA-based approach
Denote TE to be the list of test cases which are identified accessing external data
Denote TA to be the list of test cases which are selected and able to affect other test cases
Denote TC to be the list of test cases being checked if they are affected by the selected test cases
Denote TR to be the list of test cases which are in TC and affected by the test cases in TA
Denote TEDSA to be the list of test cases including both the FCA result and the EDSA result

Save all test cases from TFCA into TEDSA
Select test cases which are in both TFCA and TE, and have w-use or rw-use usages of external
data, save them in TA
Identify the test cases which are in TE but NOT in TFCA, and have r-use or rw-use usages of
external data, save them into TC

NoNewTestCaseFound = false

WHILE NOT NoNewTestCaseFound, DO
        FOR each test case $tc_i$ in TC, DO
                Check $tc_i$ with every test case $ta_j$ in TA
                IF they use the same external data, THEN
                        Add $tc_i$ into the result list TR

        IF TR is not empty, THEN
                Save all test cases from TR into TEDSA
                Clear TA
                Identify the test cases which are in TR, and have w-use or rw-use usages of
                external data, save them into TA
                Identify the test cases which are in both TR and TC, remove them from TC
                Clear TR
        ELSE
                NoNewTestCaseFound = true

RETURN TEDSA
```

Figure 4-6: The algorithm for RTS through External Data Sharing Analysis

5. Implementation

In what follows we discuss the implementation details of our Java RMI-based Programs Analysis Toolkit (JRPAT) that was developed as part of this research. In the first part of the chapter, we focus on the Eclipse plug-in designed to trace distributed Java RMI applications. We refer to this plug-in as the JRPAT-Tracer. This plug-in is used to establish the link between test cases and execution traces required for the RTS approach. In the second part of this chapter we introduce the analysis Eclipse plug-in, which performs the actual RTS analysis. The plug-in is referred to as the JRPAT-Analyzer.

5.1 The Tracing Plug-in JRPAT-Tracer

The JRPAT-Tracer plug-in was developed to instrument distributed Java RMI programs, collect runtime data from both local calls and remote invocations, and store the collected information in a central persistent data storage. Figure 4-1 shows a general overview of the JRPAT-Tracer plug-in:

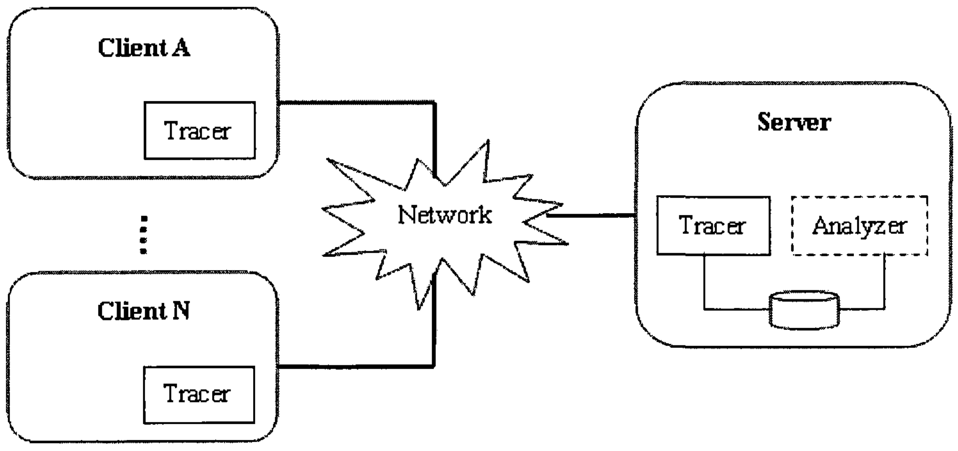
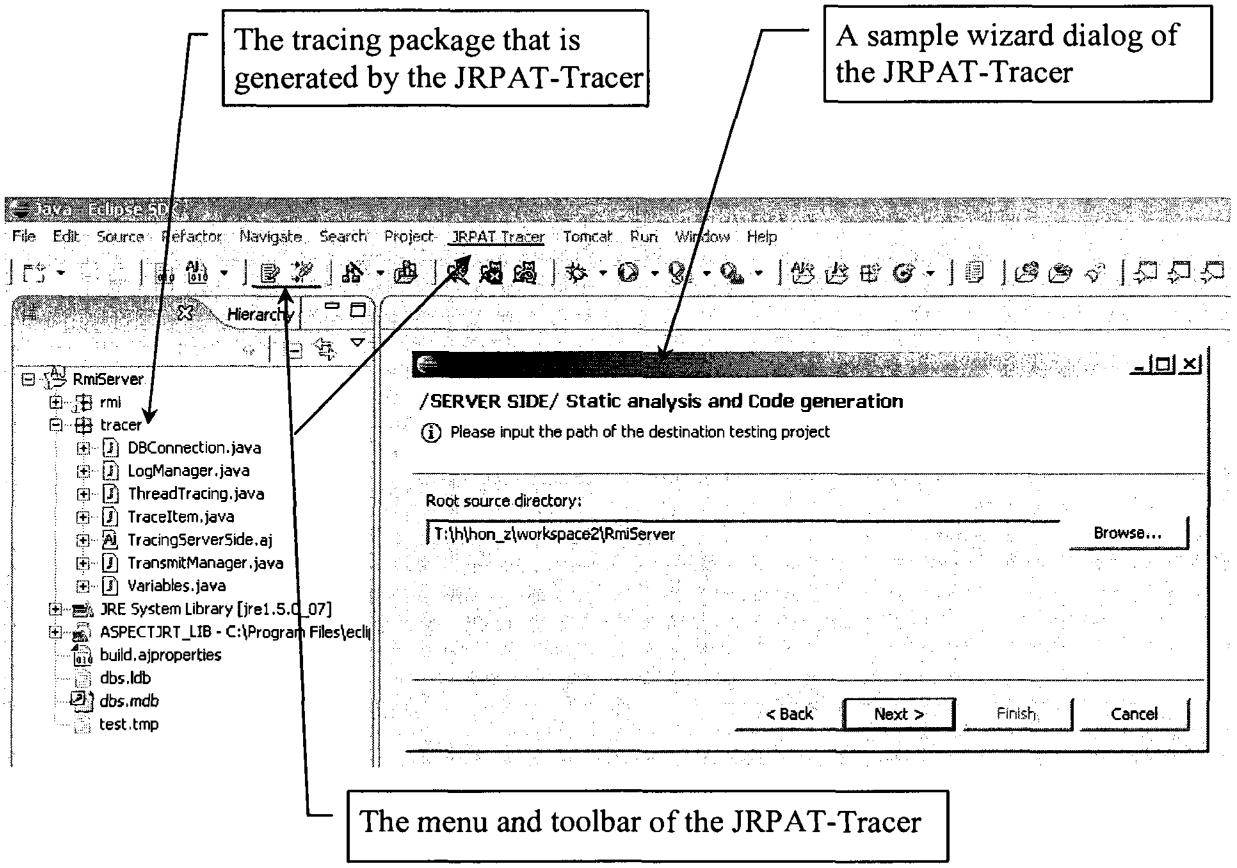


Figure 5-1: The general overview of the JRPAT-Tracer plug-in

A wizard like approach was implemented to guide users during the instrumentation process, by providing a sequence of dialogs. These dialogs include project selection, tracing code generation and instrumentation, RMI stub/skeleton class deployment, etc. Figure 5-2 shows the JRPAT-Tracer interface used to support a wizard based instrumentation of the programs to be analyzed.


The tracing package that is generated by the JRPAT-Tracer
A sample wizard dialog of the JRPAT-Tracer
File Edit Source Refactor Navigate Search Project JRPAT Tracer Tomcat Run Window Help
Hierarchy
RmiServer
rmi
tracer
DBConnection.java
LogManager.java
ThreadTracing.java
TraceItem.java
TracingServerSide.aj
TransmitManager.java
Variables.java
JRE System Library [jre1.5.0_07]
ASPECTJRT_LIB - C:\Program Files\ecli
build.ajproperties
dbs.ldb
dbs.mdb
test.tmp
Browse...
< Back
Next >
Finish
Cancel
The menu and toolbar of the JRPAT-Tracer

/SERVER SIDE/ Static analysis and Code generation

Please input the path of the destination testing project

Root source directory: T:\h\hon_z\workspace2\RmiServer

Figure 5-2: The JRPAT-Tracer interface and a sample wizard dialog

Next, we illustrate in more detail, how the actual client and server side instrumentation is performed through the JRPAT-Tracer plug-in.

5.1.1 Tracing Local Calls

Local calls, which correspond to method call sequences between components running on the same Java Virtual Machine, are traced through Aspect templates. These calls can be categorized into four groups:

(1) Non-static method calls. This type of method calls are invoked on an instance of a class. These calls are captured through the following Aspect template:

```
Object around (Object targetObj):
      call(public * PACKAGE_NAME..*(..)) &&
      target(targetObj)
      {
            ADVICE_BODY
            return proceed(targetObj);
      }
```

(2) Static method calls. Static methods are invoked directly within a class, the `target()` pointcut will not automatically match calls to such a method. These calls are traced by using the following Aspect template:

```
Object around():
      call(public static * PACKAGE_NAME..*(..))
      {
            ADVICE_BODY
            return proceed();
      }
```

(3) Constructor method calls. Constructor method call is a specific case of non-static method call. A constructor method is automatically called when an object is created using the keyword `new`. The following template is used to collect the execution of the constructor methods.

```
Object around():
        call(PACKAGE_NAME.new(..))
        {
                ADVICE_BODY
                return proceed();
        }
```

(4) Thread starting calls. Threads are created by calling the `start()` function on objects whose class implements the interface `java.lang.Runnable`. The template below is used for collecting the executions involved in these calls:

```
Object around (Object targetObj):
        call(* java.lang.Runnable+.start(..)) &&
        target(targetObj)
        {
                ADVICE_BODY
                return proceed(targetObj);
        }
```

Given this Aspect template we can now identify the starting calls of thread to allow for the tracing of the method invocation sequences in multiple threads.

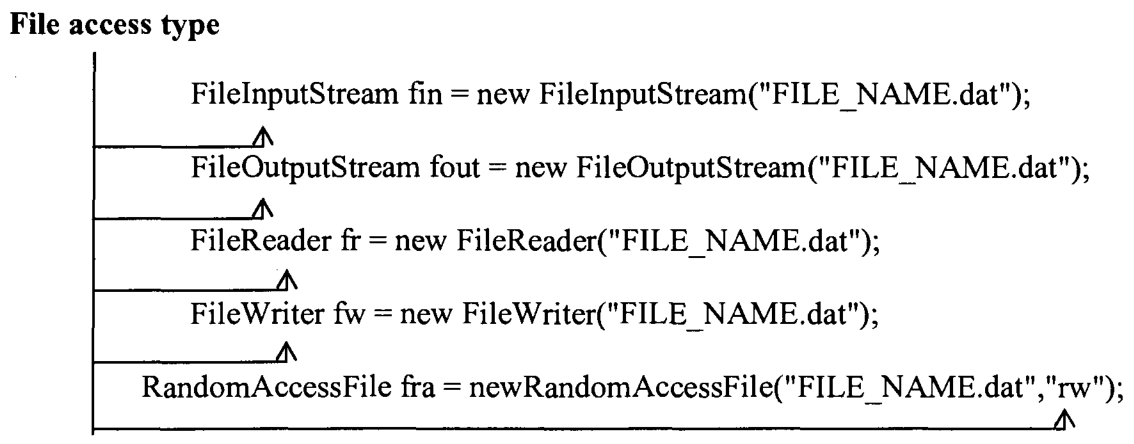
For single threaded programs, class names, object identifiers and timestamp are sufficient to identify a trace record. However, for distributed multithreaded programs, a thread identifier is needed to identify each trace record. Moreover, since the same thread identifier might be assigned to several threads operating on different nodes at the same time, the node name on which the thread is running has to be included in each trace record.

5.1.2 Monitoring External Data Interactions

As previously introduced (Section 4.1.3) we consider in our approach external data (e.g. files or databases) elements. While monitoring external data interactions, we in particular interested not only in identifying and tracing the source code involved in the external data access but also the type of data access performed (i.e. read, write access).

5.1.2.1 File Access

In the context of this research we restrict the monitoring of external file to the I/O access types shown in Table 4-1, and the supported file access types to read, write, and random access. Common to Java is that both the access type and I/O name that are accessed are specified during the object instantiation. Figure 5-3 shows some of these file access instantiation we support in our approach.



```
FileInputStream fin = new FileInputStream("FILE_NAME.dat");
FileOutputStream fout = new FileOutputStream("FILE_NAME.dat");
FileReader fr = new FileReader("FILE_NAME.dat");
FileWriter fw = new FileWriter("FILE_NAME.dat");
RandomAccessFile fra = newRandomAccessFile("FILE_NAME.dat","rw");
```

Figure 5-3: File access instantiation

We can now trace these file I/O through the following AspectJ pointcut called `file_mutators`. This pointcut monitors all the constructor calls to these file access classes.

```
pointcut file_mutators():
(
          call(java.io.FileInputStream.new(..))
       || call(java.io.FileOutputStream.new(..))
       || call(java.io.FileReader.new(..))
       || call(java.io.FileWriter.new(..))
       || call(java.io.RandomAccessFile.new(..))
);
```

5.1.2.2 Database Access

External data accessed in database tables can be monitored through the following Java classes, `Statement`, `PreparedStatement` and `ResultSet`. In what follows, we describe the tracing of these classes using AspectJ.

(1) Statement

There are several methods provided by the class `Statement` to provide database access:

- `executeQuery()`, which retrieves data from a table using a SELECT statement (*r-use*).
- `executeUpdate()`, which can be used to INSERT, UPDATE, or DELETE records in a table by executing SQL statements (*w-use*).
- `execute()`, it can work as `executeQuery()` or `executeUpdate()` specified by the given SQL statement (*r-use* or *w-use*).

If users require frequent insertions/updates/deletions of a database, they can improve the database performance by using the `addbatch()` and `executebatch()` methods of the `Statement` objects. Utilizing batch statements to process the database, the tracing related information is collected when the method `addbatch()` is invoked with a SQL statement as its parameter, for example:

```
stmt.addBatch("UPDATE TableName SET ColumnName = *");
```

accordingly, the pointcuts for tracing these two methods are

```
     call(* java.sql.Statement+.execute*(..))
  || call(* java.sql.Statement+.*Batch(..))
```

Each of the pointcuts collects invocations to any method in the `Statement` class or its subclasses and supports any argument and returns type and their name either begins with `execute` or ends on `Batch`.

(2) PreparedStatement

The prepared statement provides database table operations, through the support of SQL

```
 PreparedStatement pstmt = con.prepareStatement
("UPDATE TableName SET ColumnName = ? WHERE ColumnName = ?");
```

The invocation of these operations can be captured through the following pointcut

```
call(* java.sql.Connection+.prepareStatement(..))
```

The pointcut captures all calls to the method `prepareStatement` in the `Connection` class or its subclasses.

(3) ResultSet

The `ResultSet` is a table corresponding to the results returned from a database access. It is generated by executing the method `executeQuery()` or `execute()` to query the database. The following are two examples for creating such `ResultSet` instances:

```
ResultSet rs = stmt.executeQuery(SQL statement);
```

or

```
stmt.execute(SQL statement);
ResultSet rs = stmt.getResultSet();
```

The `ResultSet` class defines the `insertRow()` method that inserts a row into a table, the `deleteRow()` method that deletes a row from a table, and the `updateRow()` method that updates a row in a table. The execution of these methods can be captured by using the pointcuts below:

```
   call(* java.sql.Statement+.execute*(..))
|| call(* java.sql.ResultSet+.*Row(..))
```

The first pointcut is used to trace the database table associated with a `ResultSet` object, the second pointcut captures information about the type of table access that is being performed by the `ResultSet` object.

5.1.3 Tracking Remote Invocations

Within our tracing environment both client and server message exchanges through remote invocations are traced. The tracing of these remote invocation is performed by using the `around` advice Aspect template shown in Figure 5-4 for the client side, and the `wrapper` method template shown in Figure 5-5 for the server side.

```
Object around(Object targetObj):
      call(* java.rmi.Remote+.*(..)) &&
      target(targetObj)
      {
      ◆  Collects the client-side trace of the remote invocation,
         including the name of the remote method and the arguments passed
         to it;
      ◆  Redirect the call to the wrapper method with above data as
         parameters;
      ◆  Gets the result from the wrapper method and passes the result
         and control back to the invocation.
      }
```

Figure 5-4: The client-side `around` advice Aspect template

The `around` advice Aspect template captures the call to a remote method that is intercepted by the following pointcut:

```
call(* java.rmi.Remote+.*(..))
```

This pointcut intercepts all calls to methods defined in any of the classes that implement the `java.rmi.Remote` interface. Whenever the call is advised by the `around` advice, the method's context information, such as the target object on which the method is called (**target**`(targetObj)`) and the method's arguments are collected. Leveraging the Java reflection mechanism, the advice can determine the wrapper method at runtime (`targetObj.getClass().getMethod()`), and redirects the call to it (`invoke()`). The redirected call includes the original method's arguments and additional arguments (i.e. the name of the real remote method, the client-side trace of the remote invocation). Finally, the advice receives the result from the `wrapper` method and passes both the result and the control back to the method invocation.

```
public Object PACKAGE_NAME.INTERFACE_NAME.WrapperMethod (
      clientside_invocation_record, remote_method_name,
      remote_method_arguments) throws RemoteException
      {
      ◆  Processes the clientside_invocation_record;
      ◆  Finds the original remote method  specified by
         remote_method_name and invokes it with remote_method_arguments;
      ◆  Gets the result of the call to the original remote method, adds
         with the server-side invocation record and returns them back to
         the client side.
      }
```

Figure 5-5: The server-side `wrapper` method template

Figure 5-5 shows the server side wrapper method that wraps all methods defined in the interface class extending `java.rmi.remote`. This wrapper method is added to the server-side interface class, and the body of the wrapper is implemented in the server-side tracer. The wrapper method works like a regular RMI remote method; it receives the remote call from the client side with parameters including the client-side remote method invocation trace information, the name and arguments of the original remote method. The wrapper method uses the latter two parameters to get and invoke the original remote method, and then returns the invocation result, as well as the server-side remote method invocation trace back to the client.

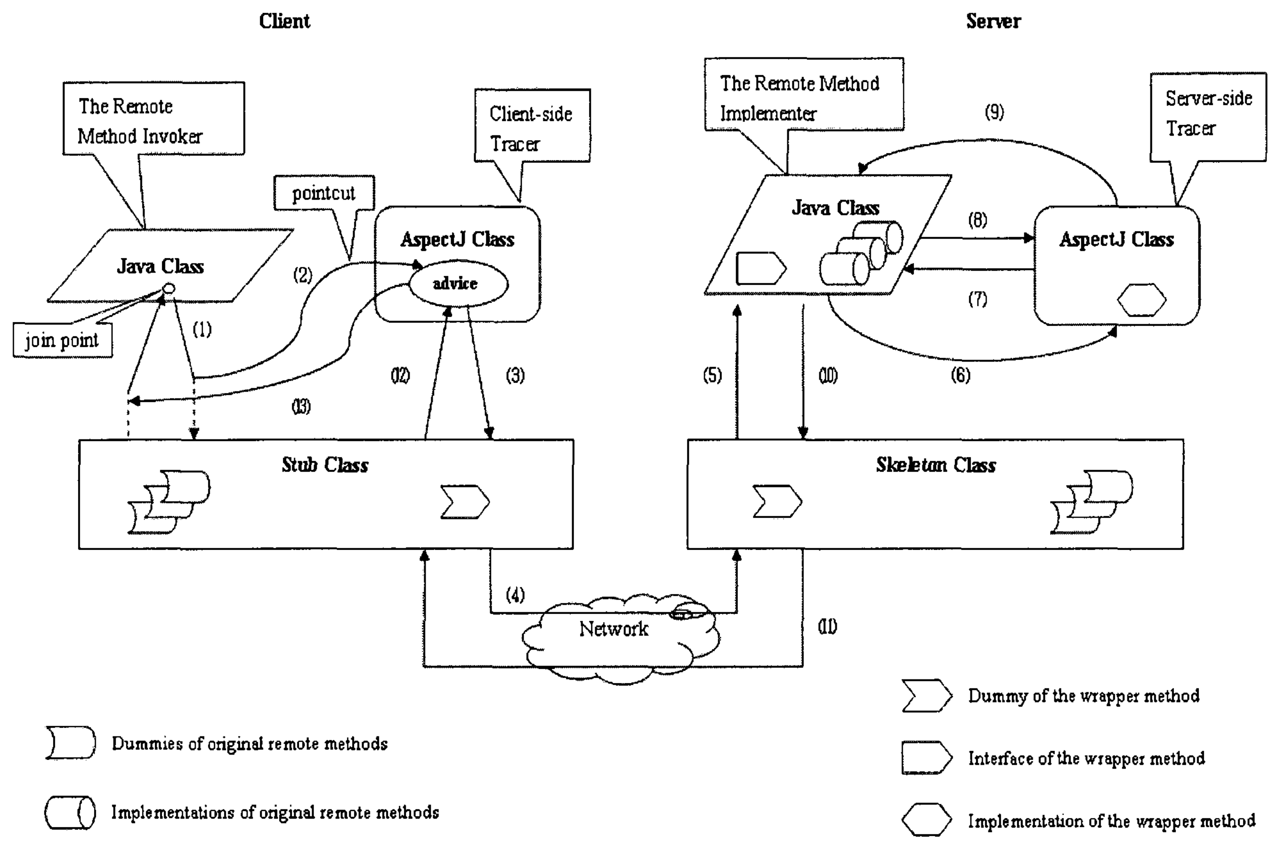

Client
Server
The Remote Method Invoker
Client-side Tracer
pointcut
Java Class
AspectJ Class
advice
join point
(1)
(2)
(3)
(12)
(13)
Stub Class
The Remote Method Implementer
Server-side Tracer
(9)
Java Class
(8)
AspectJ Class
(7)
(6)
(5)
(10)
Skeleton Class
(4)
Network
(11)
Dummy of the wrapper method
Dummies of original remote methods
Interface of the wrapper method
Implementations of original remote methods
Implementation of the wrapper method

Figure 5-6: The working process of tracing remote invocations through the Aspect template and the wrapper method

Figure 5-6 shows the process on how the Aspect template and the wrapper method work together in tracing remote method calls. The involved steps (1) to (13) are described below in chronological order:

(1) The client-side method invokes the remote method.

(2) The invocation is captured as a join point through a pointcut defined by the AspectJ class (advice) as part of the client-side tracer.

(3) The advice calls the wrapper method on the instance of the client-side stub class (STUB), passes the client-side tracing record, the name and the arguments of the remote method as parameters.

(4) STUB then communicates with the skeleton class on the server side (SKEL) through the object serialization protocol for the remote invocation.

(5) SKEL invokes the wrapper method; its interface is compiled in the remote method implementation class (IMPL) of the destination application.

(6) This invocation is transmitted to the AspectJ class in the server-side tracer, which has the whole body of the wrapper method.

(7) The server-side AspectJ class gets parameters listed in (3), and calls the actual remote method in IMPL specified by the method name and arguments it received.

(8) The result of the invocation to the actual remote method is returned back to the server-side AspectJ class from IMPL.

(9) The AspectJ class passes the result of the wrapper method, which includes the actual remote method and the server-side tracing record, to IMPL.

(10) IMPL delivery the return value of the wrapper method to SKEL.

(11) SKEL marshalls the return value and sends it over the wire to STUB.

(12) STUB demarshalls the return value and returns it to the client-side AspectJ class.

(13) The result and the control are returned back to the join point, the client-side remote method call.

5.1.4 Collecting Traces

One of the main challenges while tracing distributed programs is the need not only to collect local trace information from clients and servers, but also to merge the data for further analyses. In our approach, the traces are first buffered locally and then propagated to the sever-side database by leveraging the communication channel provided by Java RMI. During the execution of a test case, first both client-side and server-side traces are stored in a local memory buffer. After the execution of the test case is completed, the server-side traces are stored in a database hosted on the server. The client-side tracer first ensures that not only the local execution is complete but also the server-side application is not busy. Next the local client runtime data can be transmitted to the server by invoking a server-side remote method. This remote method is implemented by the server-side tracer and uses the RMI communication channel to transfer the client-side traces to the database on the server. The template for the remote method is shown in Figure 5-7:

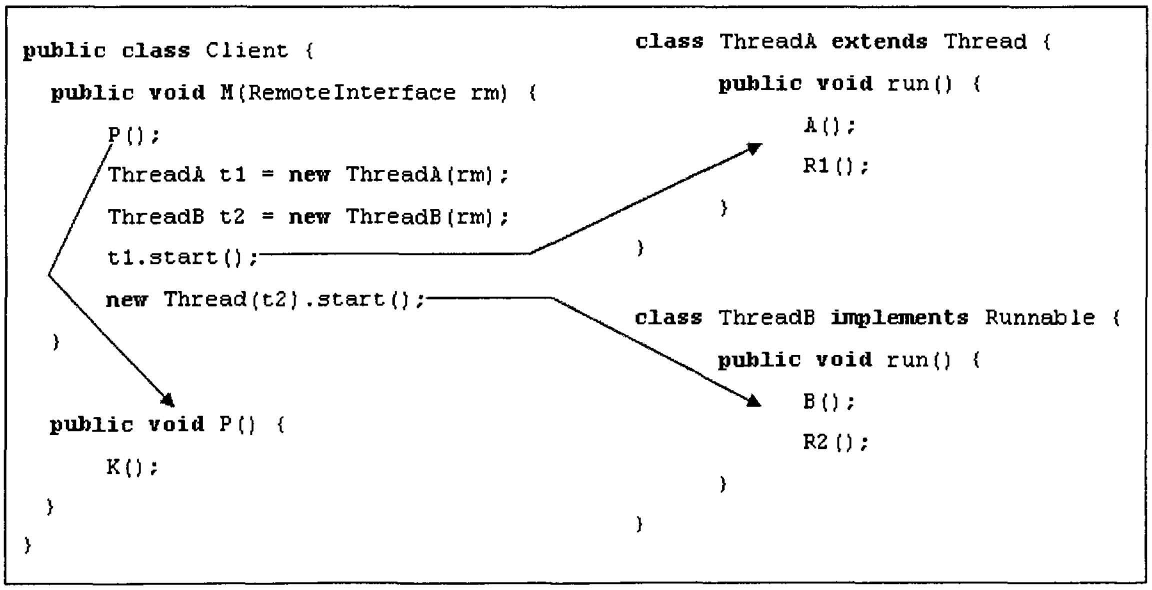
```
public boolean PACKAGE_NAME.INTERFACE_NAME.TransmitMethod
      (ArrayList recvBuffer, int dataType) throws RemoteException
      {
      ◆  Makes sure that the server application is not busy;
      ◆  Processes the tracing records received, which are
         execution data or test coverage information;
      ◆  Saves the records into the central database on the server.
      }
```

Figure 5-7: The template for collecting client-side traces

5.1.5 Tracing the Method Invocation Sequence

In AspectJ, method executions can be identified through join points and intercepted by pointcuts. When entering, executing or leaving a join point, additional advice operations can be performed, like `before`, `around`, and `after`. These advices are used to implement a stack-based algorithm that we use to (1) identify the parent-child relationship and (2) establish the invocation sequence among method calls at runtime.

Within our environment, an algorithm is implemented to maintain stacks of executed methods. A method is pushed on a stack each time when its `before` advice is reached. The execution related information of this method is stored in the output buffer when its associated `around` advice is executed. The method is popped off the stack through its `after` advice. If a method calls child methods, its execution is only completed after all the nested child method executions are completed.



```
public class Client {
  public void M(RemoteInterface rm) {
      P();
      ThreadA t1 = new ThreadA(rm);
      ThreadB t2 = new ThreadB(rm);
      t1.start();
      new Thread(t2).start();
  }

  public void P() {
      K();
  }
}
```

```
class ThreadA extends Thread {
      public void run() {
            A();
            R1();
      }
}

class ThreadB implements Runnable {
      public void run() {
            B();
            R2();
      }
}
```

Figure 5-8: A distributed, multithreaded client-side source code example

For a multithreaded application, the algorithm of tracking method calls has to be further extended, since method executions in different threads can be interleaved. In this case, tracking the sequence of method invocations requires to create multiple stacks - one stack per thread.

The example in Figure 5-8 shows a distributed and multithreaded client-side program, and the result of applying the stack algorithm on this example is shown in Figure 5-9.

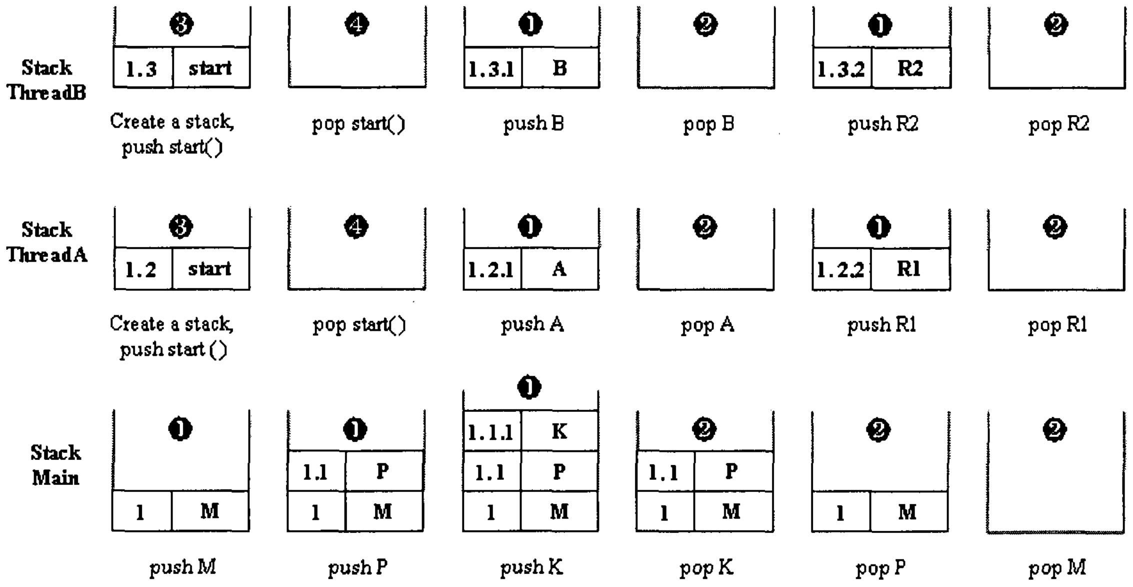


Figure 5-9: Application of the stack-based algorithm for creating the invocation sequence for the program in Figure 5-8

(1) In the first execution step, method `M()` calls method `P()`, and `P()` calls its child method `K()`. These invocations are captured by the Aspect template shown below:

```
before() :
call(public * PACKAGE_NAME..*(..)) &&
!call(* java.lang.Runnable+.start(..))
{
... pushes the method captured into the stack it belongs to
}
```

The pointcut specifies that any invocation of non-static or static method on any class in a given package (defined by `PACKAGE_NAME`) except the `start()` method calls in the interface `Runnable` or its subclasses are captured. The `before` advice then pushes the methods calls traced on their associated stack (❶).

(2) In the next step (❷), the `after` advice pops the completed method calls `K(),P()` off their associated stack. The pointcuts below capture any invocation of non-static or static methods on any class in a given package (defined by `PACKAGE_NAME`), and the `start()` method calls on the interface `Runnable` or its subclasses.

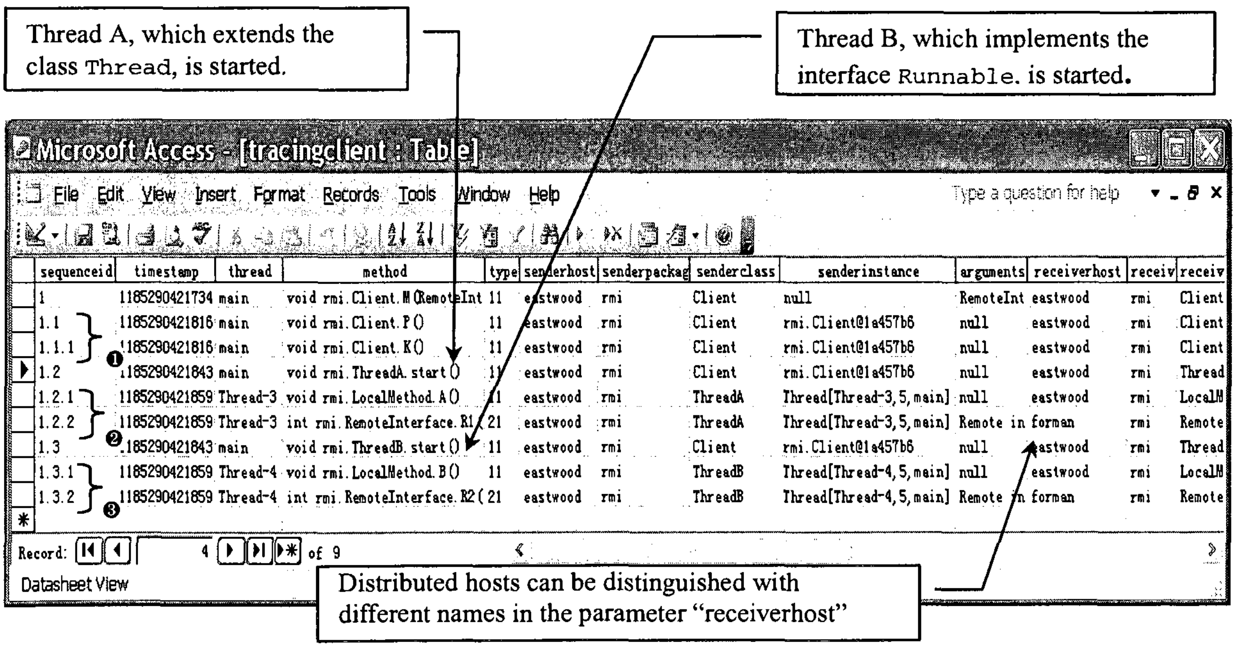
```
after() :
call(public * PACKAGE_NAME..*(..)) ||
call(* java.lang.Runnable+.start (..))
{
... pops the method captured off its associated stack
}
```

(3) In the next step (❸), method `M()` calls methods `new t1.start()` and `Thread(t2).start()` to start two new threads. The calls to the `start()` method are intercepted by the following Aspect template. In the template, the pointcut matches the `start()` method call in the interface `Runnable` or its subclasses, and intercepts the thread starts. The `before` advice then generates a new stack for the newly created thread, and pushes the `start()` method on the stack prior to the execution of the actual `start()` method.

```
before():
call(* java.lang.Runnable+.start(..))
{
... creates a new stack for the thread,
    And pushes the start() method into the stack
}
```

(4) After the completion of the `start()` method (❹), the `after` advice pops the `start()` method off the stack. The invocations of the methods involved in different threads, e.g. `A()`, `R1()`, `B()`, `R2()`, are also intercepted and advised through the previously introduced Aspect templates.

Figure 5-10 shows the traces generated for the above example program. The execution records in Stack Main, Stack ThreadA and Stack ThreadB are marked as ❶, ❷ and ❸.

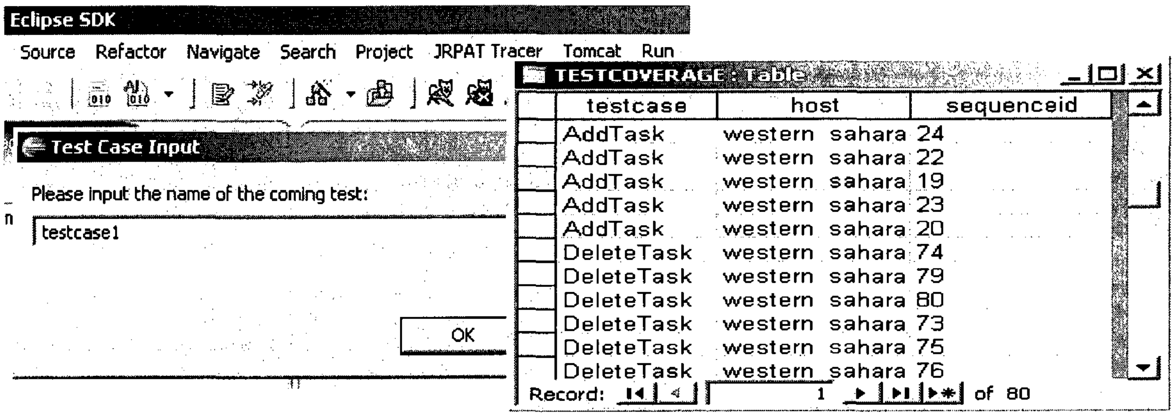

Thread A, which extends the class Thread, is started.
Thread B, which implements the interface Runnable. is started.
Microsoft Access - [tracingclient : Table]
File Edit View Insert Format Records Tools Window Help
Type a question for help
Distributed hosts can be distinguished with different names in the parameter "receiverhost"
❶
❷
❸
Record: 4 of 9
Datasheet View

sequenceid	timestamp	thread	method	type	senderhost	senderpackag	senderclass	senderinstance	arguments	receiverhost	receiv	receiv
1	1185290421734	main	void rmi.Client.M(RemoteInt	11	eastwood	rmi	Client	null	RemoteInt	eastwood	rmi	Client
1.1	1185290421816	main	void rmi.Client.P()	11	eastwood	rmi	Client	rmi.Client@1a457b6	null	eastwood	rmi	Client
1.1.1	1185290421816	main	void rmi.Client.K()	11	eastwood	rmi	Client	rmi.Client@1a457b6	null	eastwood	rmi	Client
1.2	1185290421843	main	void rmi.ThreadA.start()	11	eastwood	rmi	Client	rmi.Client@1a457b6	null	eastwood	rmi	Thread
1.2.1	1185290421859	Thread-3	void rmi.LocalMethod.A()	11	eastwood	rmi	ThreadA	Thread[Thread-3,5,main]	null	eastwood	rmi	LocalM
1.2.2	1185290421859	Thread-3	int rmi.RemoteInterface.R1(	21	eastwood	rmi	ThreadA	Thread[Thread-3,5,main]	Remote in	forman	rmi	Remote
1.3	1185290421843	main	void rmi.ThreadB.start()	11	eastwood	rmi	Client	rmi.Client@1a457b6	null	eastwood	rmi	Thread
1.3.1	1185290421859	Thread-4	void rmi.LocalMethod.B()	11	eastwood	rmi	ThreadB	Thread[Thread-4,5,main]	null	eastwood	rmi	LocalM
1.3.2	1185290421859	Thread-4	int rmi.RemoteInterface.R2(	21	eastwood	rmi	ThreadB	Thread[Thread-4,5,main]	Remote in	forman	rmi	Remote

Figure 5-10: The client-side traces for the program in Figure 5-8

5.1.6 Executing Test Cases

Users of our system are provided with an interface to specify and associate a meaningful alias test name prior to executing a test case (Figure 5-11). A test case database table is created, containing the input conditions for the test cases and information about the host on which the test case is executed. The resulting test case table is used as a look up table to associate later on test cases with different unique sequence ids.


Eclipse SDK
Source Refactor Navigate Search Project JRPAT Tracer Tomcat Run
Test Case Input
OK
TESTCOVERAGE : Table

Please input the name of the coming test:

testcase1

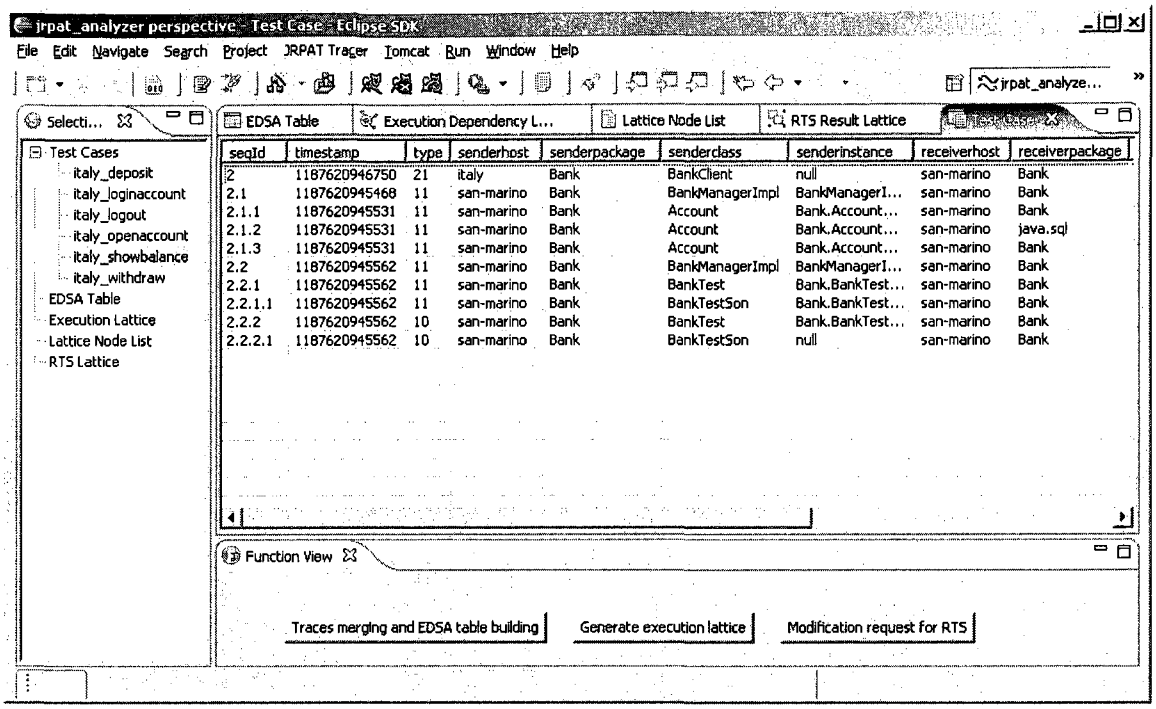
testcase	host	sequenceid
AddTask	western sahara	24
AddTask	western sahara	22
AddTask	western sahara	19
AddTask	western sahara	23
AddTask	western sahara	20
DeleteTask	western sahara	74
DeleteTask	western sahara	79
DeleteTask	western sahara	80
DeleteTask	western sahara	73
DeleteTask	western sahara	75
DeleteTask	western sahara	76

Record: 1 of 80

Figure 5-11: The test case name setting interface and a sample test coverage matrix

5.2 The JRPAT-Analyzer Plug-in

The JRPAT-Analyzer plug-in provides services for merging the client/server traces, by constructing an external data sharing table among system test cases, visualizing the execution dependency concept lattice, and supporting the proposed RTS method. An overview of the plug-in's GUI is shown in Figure 5-12.



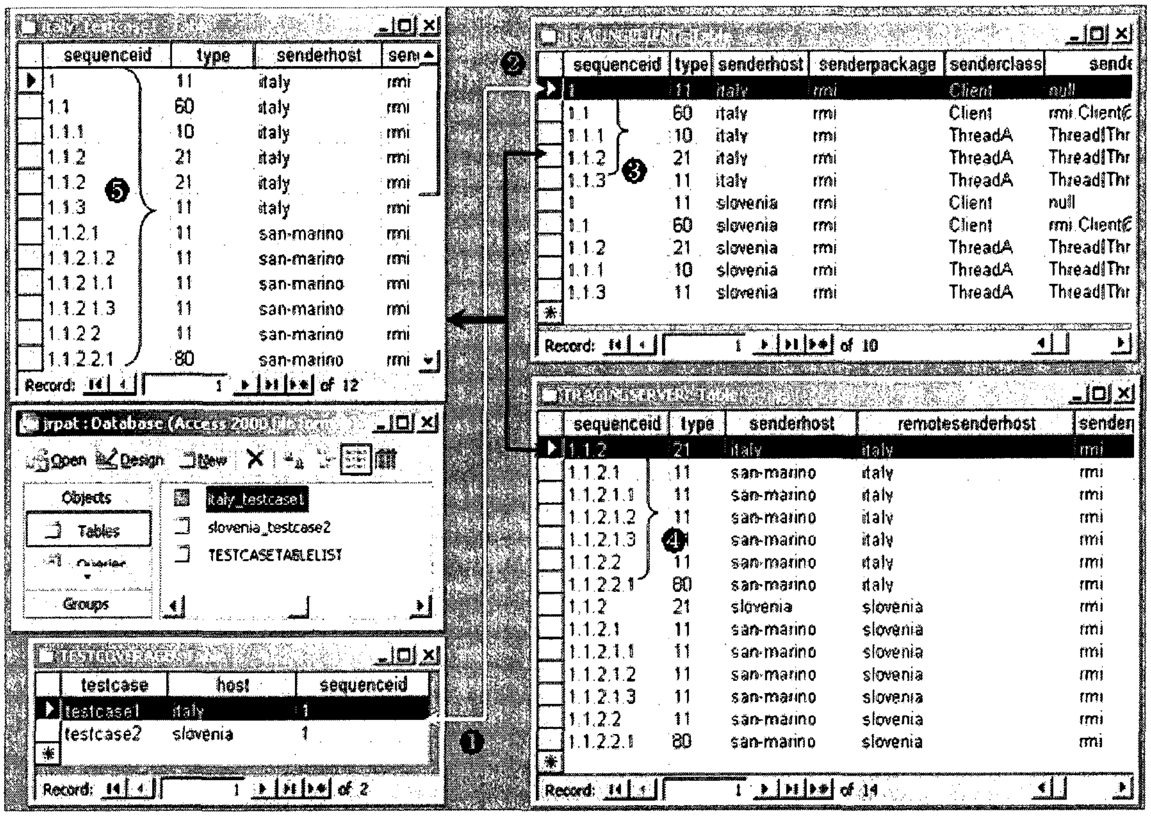
seqId	timestamp	type	senderhost	senderpackage	senderclass	senderinstance	receiverhost	receiverpackage
2	1187620946750	21	italy	Bank	BankClient	null	san-marino	Bank
2.1	1187620945468	11	san-marino	Bank	BankManagerImpl	BankManagerI...	san-marino	Bank
2.1.1	1187620945531	11	san-marino	Bank	Account	Bank.Account...	san-marino	Bank
2.1.2	1187620945531	11	san-marino	Bank	Account	Bank.Account...	san-marino	java.sql
2.1.3	1187620945531	11	san-marino	Bank	Account	Bank.Account...	san-marino	Bank
2.2	1187620945562	11	san-marino	Bank	BankManagerImpl	BankManagerI...	san-marino	Bank
2.2.1	1187620945562	11	san-marino	Bank	BankTest	Bank.BankTest...	san-marino	Bank
2.2.1.1	1187620945562	11	san-marino	Bank	BankTestSon	Bank.BankTest...	san-marino	Bank
2.2.2	1187620945562	10	san-marino	Bank	BankTest	Bank.BankTest...	san-marino	Bank
2.2.2.1	1187620945562	10	san-marino	Bank	BankTestSon	null	san-marino	Bank

Figure 5-12: The overview GUI of the JRPAT-Analyzer

In what follows we discuss in more detail the main functionality of the plug-in and its implementation.

5.2.1 Merging Client/Server Traces

The client/server traces are merged by analyzing the invocation sequence (Section 5.1.5) and the test coverage matrix (Section 5.1.6). As a result of this analysis a merged database table is created for every test case executed, with each table being identified by a unique name (a combination of the test case and its host name). For the example in Figure 5-13, we first get the information of *testcase1* (❶), including its host name (*italy*) and its base sequence id (*1*) from the test coverage matrix. This information is used to find and match the related records in the client-side traces (❷). In the next step client-side records (❸) and server-side records (❹) are linked through the sequence id (e.g. *1.1.2)*, and the remote sender host name (e.g. *italy*). Finally all records are selected and stored in a table named *italy_testcase1* (❺).



jrpat : Database (Access 2000 file format)

- italy_testcase1
- slovenia_testcase2
- TESTCASETABLELIST

sequenceid	type	senderhost	sen
1	11	italy	rmi
1.1	60	italy	rmi
1.1.1	10	italy	rmi
1.1.2	21	italy	rmi
1.1.2	21	italy	rmi
1.1.3	11	italy	rmi
1.1.2.1	11	san-marino	rmi
1.1.2.1.2	11	san-marino	rmi
1.1.2 1.1	11	san-marino	rmi
1.1.2 1.3	11	san-marino	rmi
1.1.2 2	11	san-marino	rmi
1.1.2 2.1	80	san-marino	rmi

Record: 1 of 12

testcase	host	sequenceid
testcase1	italy	1
testcase2	slovenia	1

Record: 1 of 2

sequenceid	type	senderhost	senderpackage	senderclass	sende
1	11	italy	rmi	Client	null
1.1	60	italy	rmi	Client	rmi.Client€
1.1.1	10	italy	rmi	ThreadA	Thread|Thr
1.1.2	21	italy	rmi	ThreadA	Thread|Thr
1.1.3	11	italy	rmi	ThreadA	Thread|Thr
1	11	slovenia	rmi	Client	null
1.1	60	slovenia	rmi	Client	rmi.Client€
1.1.2	21	slovenia	rmi	ThreadA	Thread|Thr
1.1.1	10	slovenia	rmi	ThreadA	Thread|Thr
1.1.3	11	slovenia	rmi	ThreadA	Thread|Thr

Record: 1 of 10

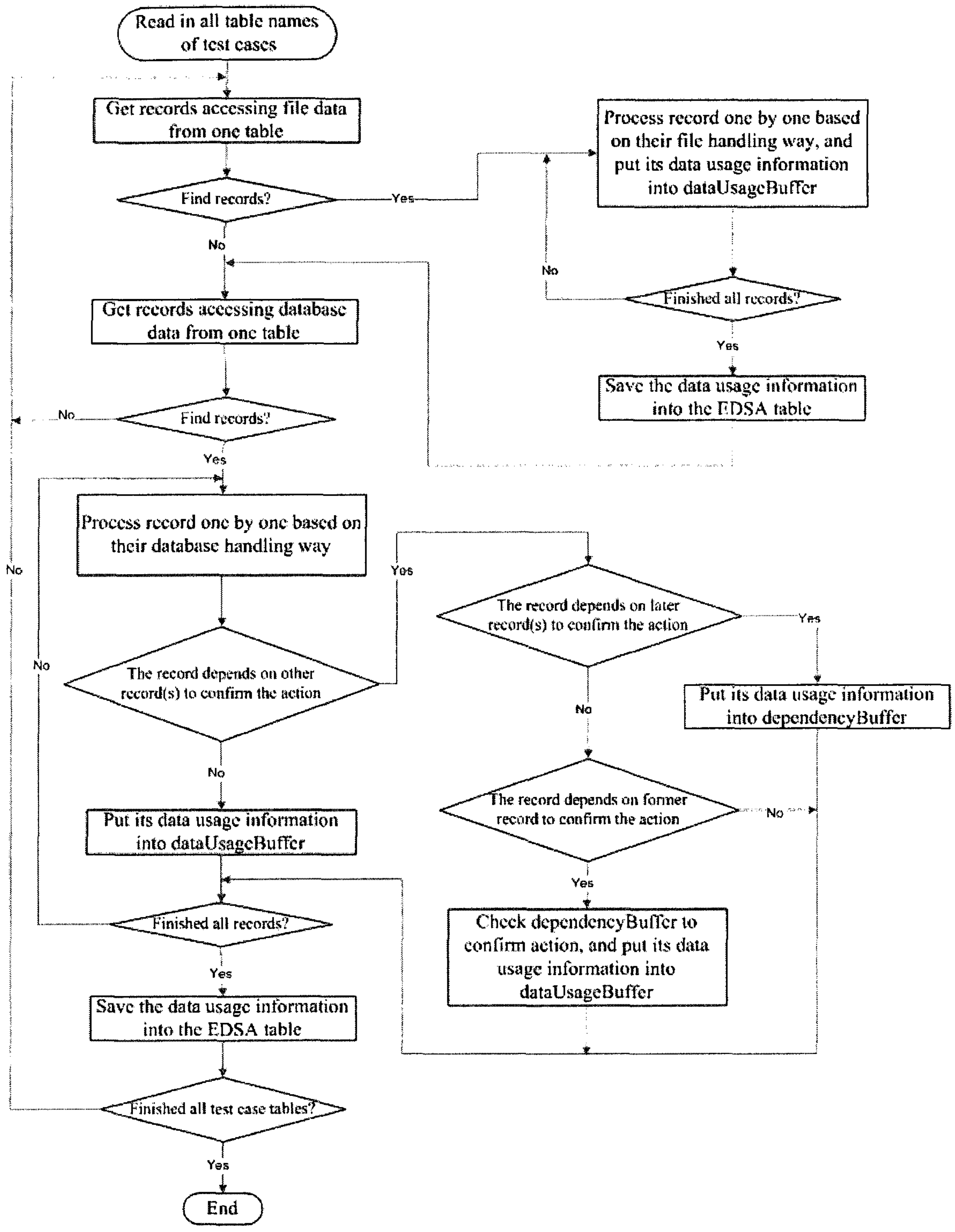
sequenceid	type	senderhost	remotesenderhost	sender
1.1.2	21	italy	italy	rmi
1.1.2.1	11	san-marino	italy	rmi
1.1.2.1.1	11	san-marino	italy	rmi
1.1.2.1.2	11	san-marino	italy	rmi
1.1.2.1.3	11	san-marino	italy	rmi
1.1.2.2	11	san-marino	italy	rmi
1.1.2.2.1	80	san-marino	italy	rmi
1.1.2	21	slovenia	slovenia	rmi
1.1.2.1	11	san-marino	slovenia	rmi
1.1.2.1.1	11	san-marino	slovenia	rmi
1.1.2.1.2	11	san-marino	slovenia	rmi
1.1.2.1.3	11	san-marino	slovenia	rmi
1.1.2.2	11	san-marino	slovenia	rmi
1.1.2.2.1	80	san-marino	slovenia	rmi

Record: 1 of 14

Figure 5-13: Client/Server traces for a test case are merged into a database table

5.2.2 Constructing External Data Sharing Table

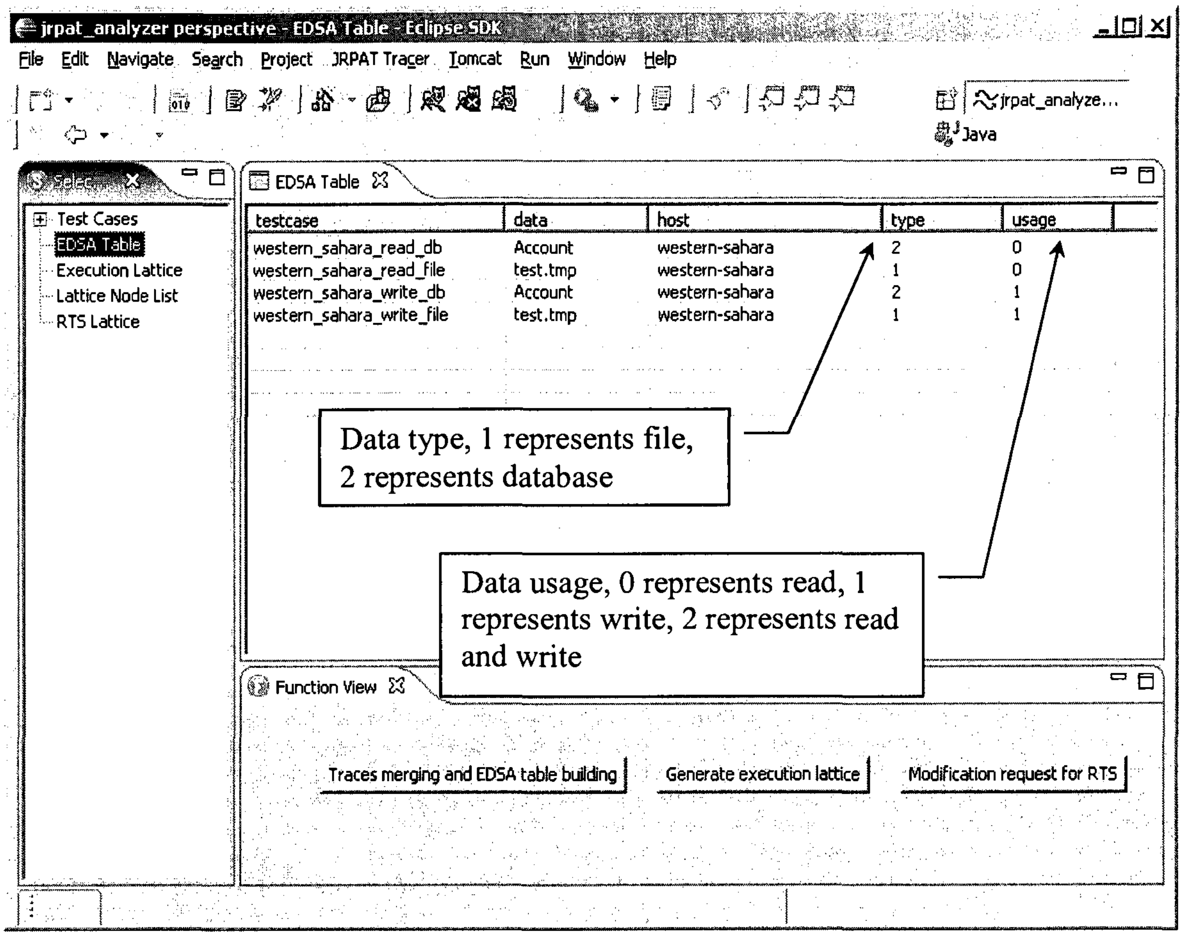
The external data sharing analysis (EDSA) table is created by the JRPAT-Analyzer plug-in by querying the database for all execution traces collected from the executed test cases. The flowchart in Figure 5-14 describes the algorithm used to create this external data sharing table in more detail.



Read in all table names of test cases
Get records accessing file data from one table
Find records?
Yes
No
Process record one by one based on their file handling way, and put its data usage information into dataUsageBuffer
Finished all records?
No
Yes
Save the data usage information into the EDSA table
Get records accessing database data from one table
Find records?
No
Yes
Process record one by one based on their database handling way
The record depends on other record(s) to confirm the action
Yes
No
The record depends on later record(s) to confirm the action
Yes
No
Put its data usage information into dependencyBuffer
The record depends on former record to confirm the action
No
Yes
Check dependencyBuffer to confirm action, and put its data usage information into dataUsageBuffer
Put its data usage information into dataUsageBuffer
Finished all records?
No
Yes
Save the data usage information into the EDSA table
Finished all test case tables?
No
Yes
End

Figure 5-14: The External Data Sharing table creation flowchart

Figure 5-15 shows a screen capture of the table created by the JRPAT-Analyzer plug-in based on the external data sharing analysis. As discussed earlier, the external data sharing table allows us to further enrich traditional data dependency analysis, which focuses typically only on internal program states, to include also external data states (e.g. files, databases).


jrpat_analyzer perspective - EDSA Table - Eclipse SDK
File Edit Navigate Search Project JRPAT Tracer Tomcat Run Window Help
jrpat_analyze...
Java
Selec...
Test Cases
EDSA Table
Execution Lattice
Lattice Node List
RTS Lattice
EDSA Table
Function View
Traces merging and EDSA table building
Generate execution lattice
Modification request for RTS

testcase	data	host	type	usage
western_sahara_read_db	Account	western-sahara	2	0
western_sahara_read_file	test.tmp	western-sahara	1	0
western_sahara_write_db	Account	western-sahara	2	1
western_sahara_write_file	test.tmp	western-sahara	1	1

Data type, 1 represents file, 2 represents database

Data usage, 0 represents read, 1 represents write, 2 represents read and write

Figure 5-15: A screenshot of an EDSA table example

5.2.3 Visualizing the FCA Execution Lattice

As discussed in Section 4, we use FCA to analyze traces to identify test cases that have to be re-executed as part of a modification request. Within the JRPAT-Analyzer plug-in, the recorded execution traces are pre-analyzed to convert them into a FCA context compatible format. In the next step we invoke our existing FCA algorithm to perform the formal concept analysis. The formal concept analysis algorithm creates an output as a *dot* format file, which is with the standard Graphviz file format. Graphviz is an open source graph visualization application which provides different options for representing structural information as abstract graphs or networks [ATT00]. Using the *dot* format file, the JRPAT-Analyzer invokes Graphviz to create the corresponding graph. The graph is then displayed within the JRPAT-Analyzer (shown in Figure 5-16).

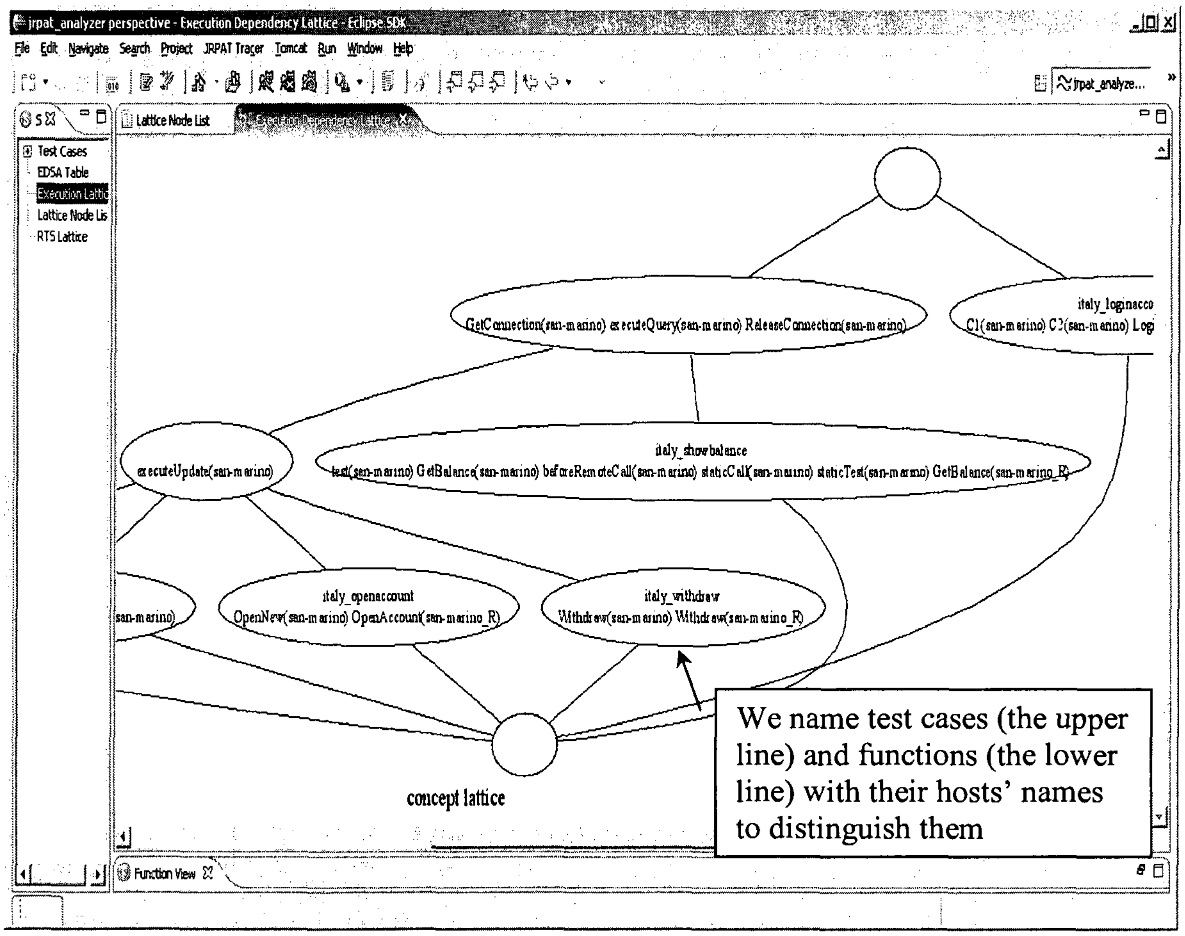
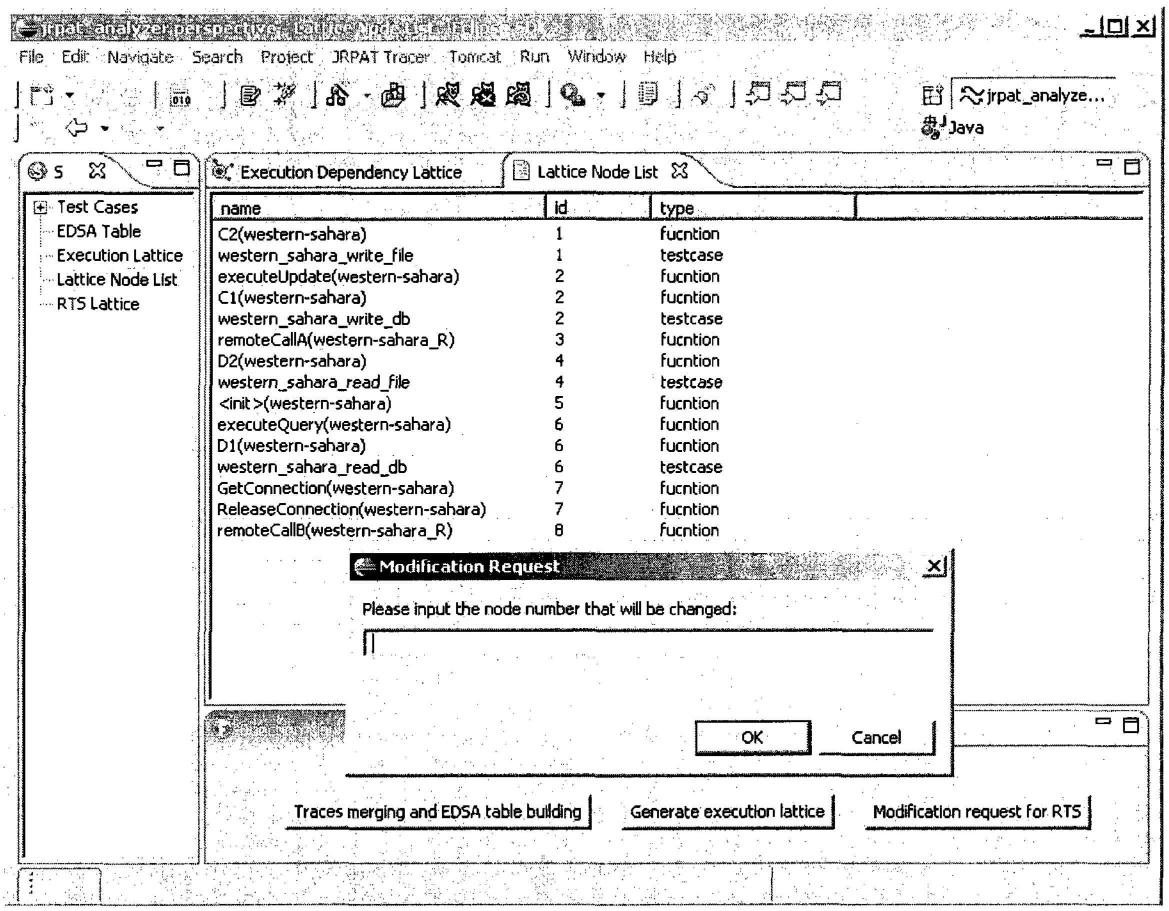

jrpat_analyzer perspective - Execution Dependency Lattice - Eclipse SDK
File Edit Navigate Search Project JRPAT Tracer Tomcat Run Window Help
jrpat_analyze...
Lattice Node List
Execution Dependency Lattice
Test Cases
EDSA Table
Execution Lattic
Lattice Node Lis
RTS Lattice
GetConnection(san-marino) executeQuery(san-marino) ReleaseConnection(san-marino)
italy_loginacco
C1(san-marino) C2(san-marino) Logi
executeUpdate(san-marino)
italy_showbalance
test(san-marino) GetBalance(san-marino) beforeRemoteCall(san-marino) staticCall(san-marino) staticTest(san-marino) GetBalance(san-marino_R)
(san-marino)
italy_openaccount
OpenNew(san-marino) OpenAccount(san-marino_R)
italy_withdraw
Withdraw(san-marino) Withdraw(san-marino_R)
concept lattice
We name test cases (the upper line) and functions (the lower line) with their hosts' names to distinguish them
Function View

Figure 5-16: A sample execution dependency lattice

5.2.4 Specifying the Modification Request

In order to specify a modification request, the user will have to determine which node will be modified as part of the change request. Every node in the lattice is identified by a node id, which can be found in the node list view of the JRPAT-Analyzer (Figure 5-17).


File Edit Navigate Search Project JRPAT Tracer Tomcat Run Window Help
jrpat_analyze...
Java
S
Test Cases
EDSA Table
Execution Lattice
Lattice Node List
RTS Lattice
Execution Dependency Lattice
Lattice Node List
Traces merging and EDSA table building
Generate execution lattice
Modification request for RTS

name	id	type
C2(western-sahara)	1	fucntion
western_sahara_write_file	1	testcase
executeUpdate(western-sahara)	2	fucntion
C1(western-sahara)	2	fucntion
western_sahara_write_db	2	testcase
remoteCallA(western-sahara_R)	3	fucntion
D2(western-sahara)	4	fucntion
western_sahara_read_file	4	testcase
<init>(western-sahara)	5	fucntion
executeQuery(western-sahara)	6	fucntion
D1(western-sahara)	6	fucntion
western_sahara_read_db	6	testcase
GetConnection(western-sahara)	7	fucntion
ReleaseConnection(western-sahara)	7	fucntion
remoteCallB(western-sahara_R)	8	fucntion

Modification Request

Please input the node number that will be changed:

OK Cancel

Figure 5-17: A sample lattice node list and modification request interface

5.2.5 Performing Regression Test Case Selection

After specifying the node to be modified, the JRPAT-Analyzer invokes the FCA algorithm to determine the list of test cases that need to be retested after the program modification. The set of test cases identified by the FCA analysis is used as input to our

External Data Sharing Analysis. The EDSA uses this initial set of regression test cases to further analyze the existence of data dependencies with external data. Including these external data members might result in additional test cases that have to be included as part of the regression test selection. The flowchart depicted in Figure 5-18, illustrates the regression test selection process that includes the external data sharing analysis using the EDSA table, which corresponds to the algorithm discussed in Section 4 (Figure 4-6).

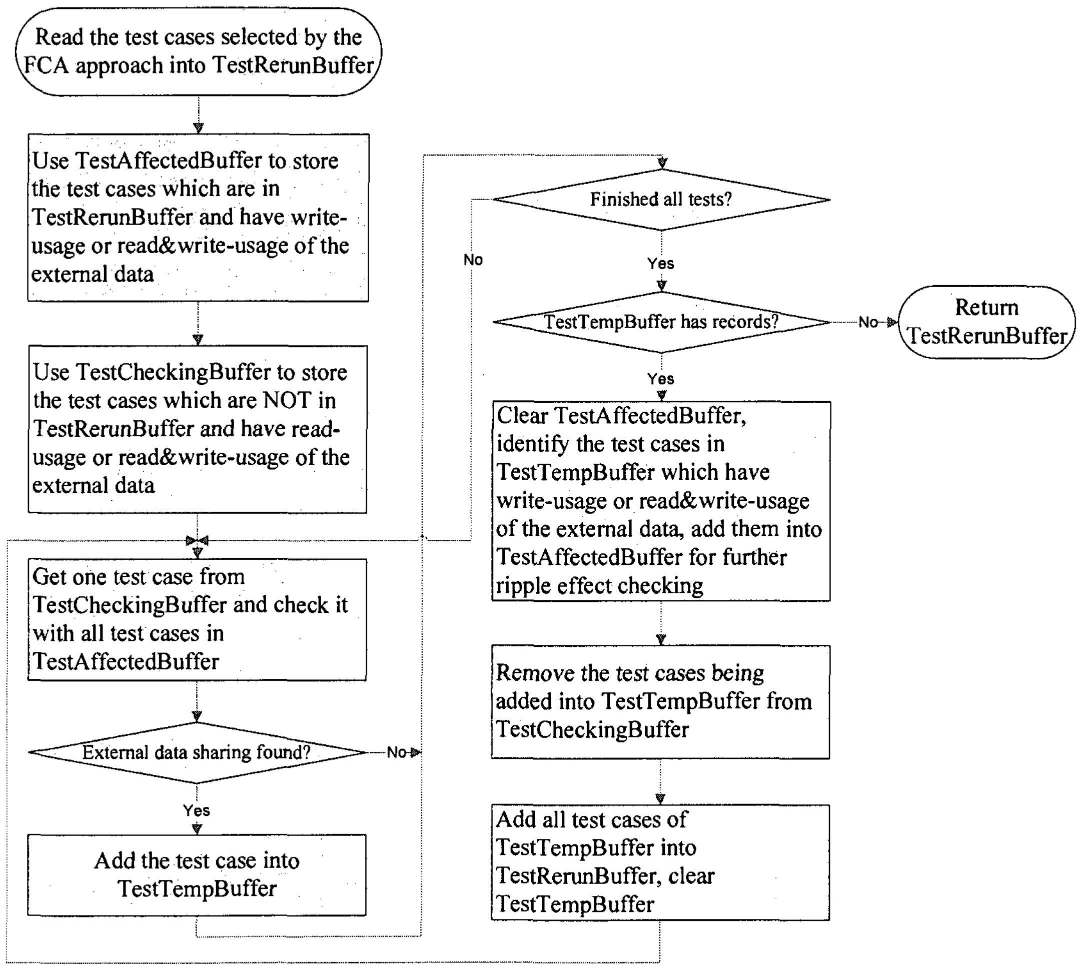

Read the test cases selected by the FCA approach into TestRerunBuffer
Use TestAffectedBuffer to store the test cases which are in TestRerunBuffer and have write-usage or read&write-usage of the external data
Use TestCheckingBuffer to store the test cases which are NOT in TestRerunBuffer and have read-usage or read&write-usage of the external data
Get one test case from TestCheckingBuffer and check it with all test cases in TestAffectedBuffer
External data sharing found?
No
Yes
Add the test case into TestTempBuffer
Finished all tests?
No
Yes
TestTempBuffer has records?
No
Return TestRerunBuffer
Yes
Clear TestAffectedBuffer, identify the test cases in TestTempBuffer which have write-usage or read&write-usage of the external data, add them into TestAffectedBuffer for further ripple effect checking
Remove the test cases being added into TestTempBuffer from TestCheckingBuffer
Add all test cases of TestTempBuffer into TestRerunBuffer, clear TestTempBuffer

Figure 5-18: The External Data Sharing Analysis flowchart

The resulting graphical representation of the RTS results is shown in Figure 5-19. In the lattice, the filled diamond corresponds to the program entity that will be modified, the two filled ellipses are the test cases selected by the FCA, and the two filled hexagons correspond to the additional test cases identified by the EDSA.

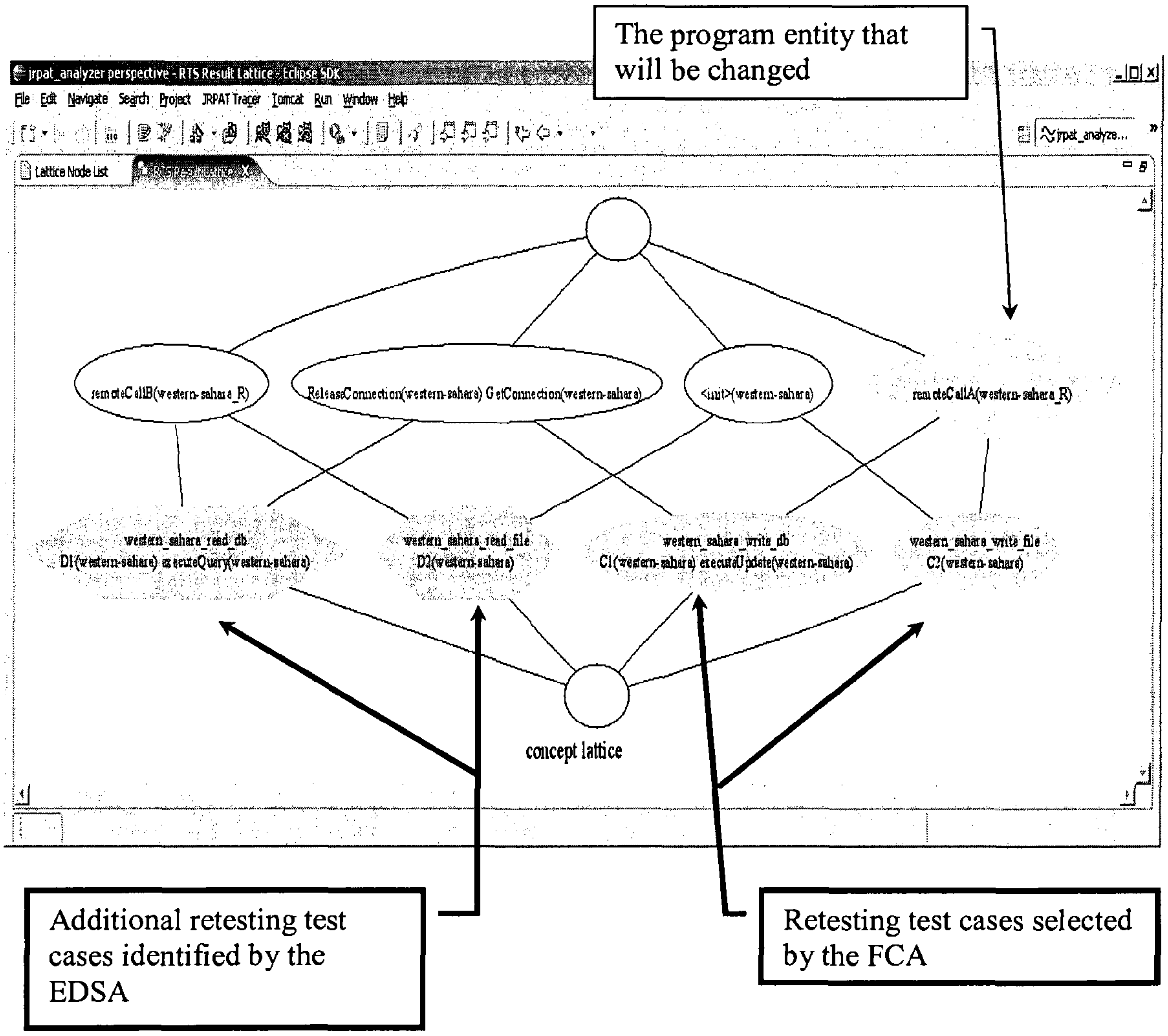

The program entity that will be changed
jrpat_analyzer perspective - RTS Result Lattice - Eclipse SDK
File Edit Navigate Search Project JRPAT Tracer Tomcat Run Window Help
jrpat_analyze...
Lattice Node List
remoteCallB(western-sahara_R)
ReleaseConnection(western-sahara) GetConnection(western-sahara)
<init>(western-sahara)
remoteCallA(western-sahara_R)
western_sahara_read_db D1(western-sahara) executeQuery(western-sahara)
western_sahara_read_file D2(western-sahara)
western_sahara_write_db C1(western-sahara) executeUpdate(western-sahara)
western_sahara_write_file C2(western-sahara)
concept lattice
Additional retesting test cases identified by the EDSA
Retesting test cases selected by the FCA

Figure 5-19: A sample graphical presentation of the RTS result

6. Initial Evaluation

In this chapter we present results of two initial case studies performed to evaluate the presented approach: in section 6.1 we present a case study performed on M-e-c Schedule. This case study was used to evaluate the applicability of our JRPAT plug-in for instrumenting Java RMI distributed program and to extract execution traces from it. The second case study (section 6.2) is performed on a sample RMI Java program called ExternalSharing that was used to evaluate the tools applicability in performing regression test selection.

6.1 Case Study 1: M-e-c Schedule

This case study is based on an open source Java RMI distributed program, M-e-c schedule[1]. The scheduling program was used to illustrate the applicability of the JRPAT plug-in for instrumenting and collecting execution traces, and representing them through both in either textual (table) or graphical (concept lattice) views. We also used this case study to analyze and evaluate the overhead associated with collecting the execution traces.

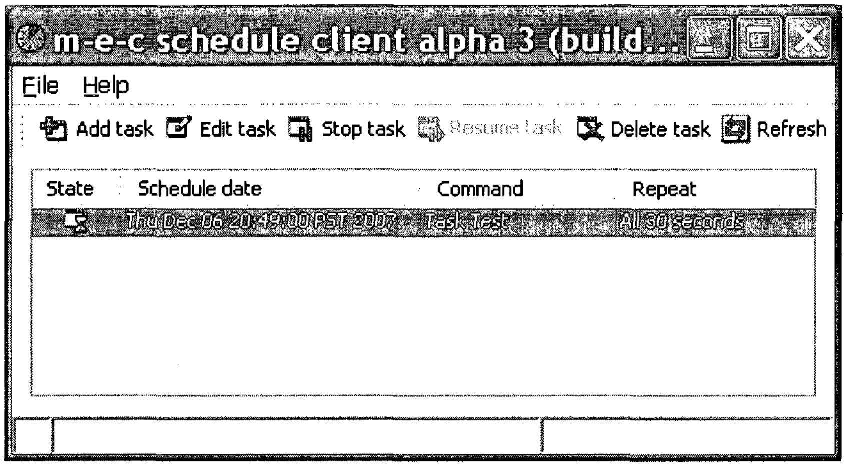
6.1.1 Case study setting

M-e-c schedule is built on client-server architecture consisting of a console based server application and SWING based client (Figure 6-1). The system allows users to schedule

[1] http://sourceforge.net/projects/mec-schedule.

tasks on the server through the client. After initiating a task with a start date and a periodic repeat, users are able to manage (edit, stop, resume, delete, etc.) the task.

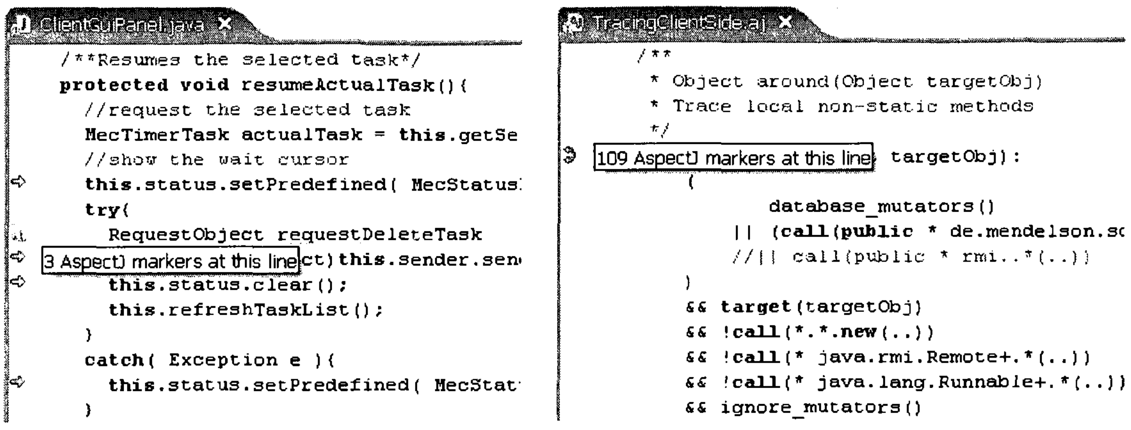
M-e-c schedule consists of 71 classes, which implement 6 different functionalities: Add task, Edit task, Stop task, Resume task, Delete task and Refresh (task). We treat each function as a separate test case.



State	Schedule date	Command	Repeat
	Thu Dec 06 20:49:00 PST 2007	Task Test	All 30 seconds

Figure 6-1: M-e-c schedule client

Both, the client and server programs of the M-e-c schedule program are instrumented through the JRPAT-Tracer running on both sides and the corresponding AspectJ tracing packages are generated and added to the project.


ClientGuiPanel.java
TracingClientSide.aj
3 AspectJ markers at this line
109 AspectJ markers at this line

```
/**Resumes the selected task*/
protected void resumeActualTask(){
  //request the selected task
  MecTimerTask actualTask = this.getSe
  //show the wait cursor
  this.status.setPredefined( MecStatus:
  try(
    RequestObject requestDeleteTask
                               ct)this.sender.sen
    this.status.clear();
    this.refreshTaskList();
  }
  catch( Exception e ){
    this.status.setPredefined( MecStat
  }
```

```
/**
 * Object around(Object targetObj)
 * Trace local non-static methods
 */
                                targetObj):
    (
          database_mutators()
        || (call(public * de.mendelson.sc
        //|| call(public * rmi..*(..))
    )
    && target(targetObj)
    && !call(*.*.new(..))
    && !call(* java.rmi.Remote+.*(..))
    && !call(* java.lang.Runnable+.*(..))
    && ignore_mutators()
```

Figure 6-2: The result of the instrumentation procedure

Figure 6-2 shows the source code after the instrumentation of the destination program. The left side shows a source code snippet of *ClientGuiPanel.java,* a client side based GUI class. The sample includes the entries to the test case functionalities listed above (i.e. resume task, add task). After instrumentation, this class file contains additional execution points (AspectJ join points) which are used to capture and generate the execution traces during runtime. The right side of Figure 6-2 shows parts of *TracingClientSide.aj*, a tracing aspect class for the M-e-c schedule client program. The part shown is the `around` advice for local non-static method calls. The `around` advice monitors 109 join points in the M-e-c schedule client program. Table 6-1 lists the join points in M-e-c schedule for both the client and server side programs, which are captured by the `around` advice in the tracing AspectJ class.

Application	**The `around` advice**	
	Join Point Type	**Join Point Total Number**
Client-Side	static method calls	37
	non-static method calls	109
	constructor calls	4
	thread start calls	4
Server-Side	static method calls	15
	non-static method calls	100
	constructor calls	4
	thread start calls	3

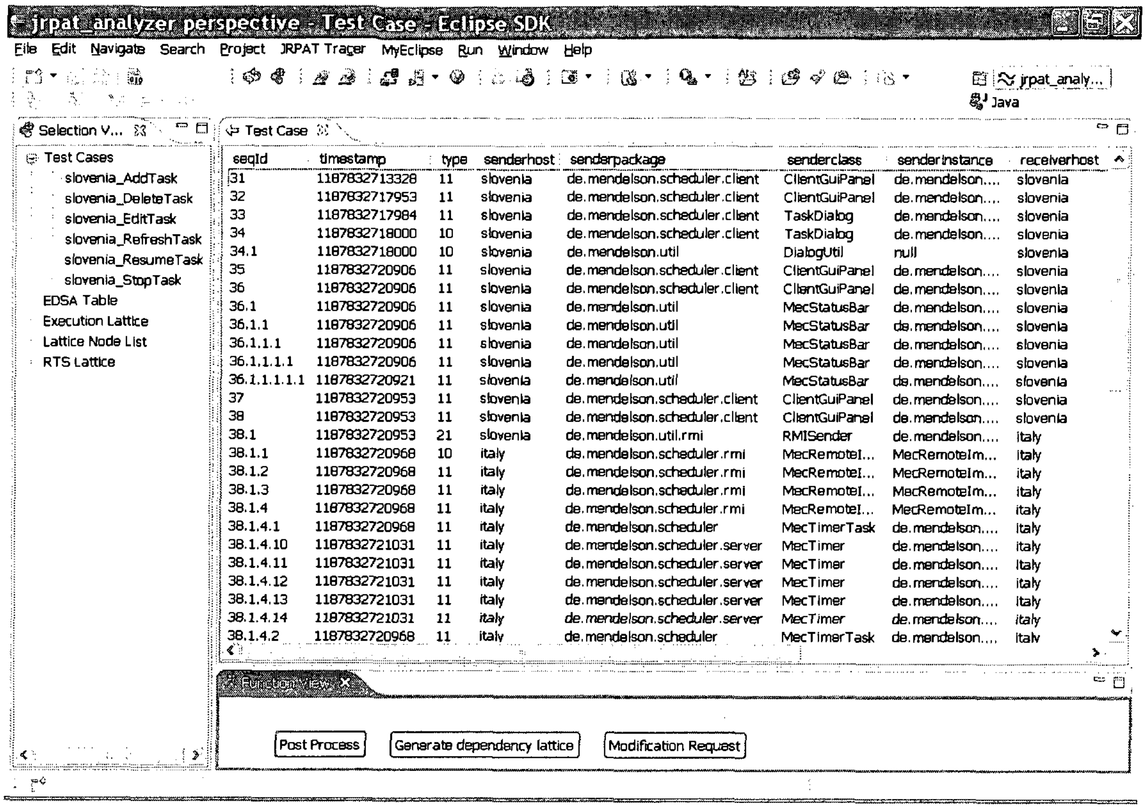
Table 6-1: The join points captured by the `around` advice on client and server sides

In the next step we execute both, the instrumented client and server programs, by executing selected test cases. The JRPAT-Tracer calculates the coverage achieved by the test case and generates execution traces for both local and remote calls. The traces are all stored as part of the system database on the server side. The server-side and client-side execution traces are initially stored within 2 separate tables, one for the client side and one for the server side. These tables are then analyzed and trace information related to a specific test case is merged from above two tables into a separate, test case specific table. For this case study, 6 tables based on available and executed test cases were created (start up of the client and server applications were not executed as part of a separate test case and therefore no specific tables were created for them). These 6 tables include 62 classes, 134 methods and 285 tracing records. The detailed information is shown in Table 6-2.

Test Cases	**Table Name**	**Coverage (sequence id range)**	**Classes**	**Methods**	**Records**
The client starts up	*No table*	*1-18*	*9*	*15*	*24*
The server starts up	*No table*	*1-13*	*8*	*14*	*40*
Add	slovenia_AddTask	19-30	14	32	83
Edit	slovenia_EditTask	31-45	11	27	51
Stop	slovenia_StopTask	46-55	9	18	30
Resume	slovenia_ResumeTask	56-65	9	18	31
Refresh	slovenia_RefreshTask	66-71	7	12	28
Delete	slovenia_DeleteTask	72-80	12	27	62

Table 6-2: The tables generated by merging Client/Server traces

Figure 6-3 uses a textual view to show the merged execution traces for an example test case: Edit task.



seqId	timestamp	type	senderhost	senderpackage	senderclass	senderInstance	receiverhost
31	1187832713328	11	slovenia	de.mendelson.scheduler.client	ClientGuiPanel	de.mendelson....	slovenia
32	1187832717953	11	slovenia	de.mendelson.scheduler.client	ClientGuiPanel	de.mendelson....	slovenia
33	1187832717984	11	slovenia	de.mendelson.scheduler.client	TaskDialog	de.mendelson....	slovenia
34	1187832718000	10	slovenia	de.mendelson.scheduler.client	TaskDialog	de.mendelson....	slovenia
34.1	1187832718000	10	slovenia	de.mendelson.util	DialogUtil	null	slovenia
35	1187832720906	11	slovenia	de.mendelson.scheduler.client	ClientGuiPanel	de.mendelson....	slovenia
36	1187832720906	11	slovenia	de.mendelson.scheduler.client	ClientGuiPanel	de.mendelson....	slovenia
36.1	1187832720906	11	slovenia	de.mendelson.util	MecStatusBar	de.mendelson....	slovenia
36.1.1	1187832720906	11	slovenia	de.mendelson.util	MecStatusBar	de.mendelson....	slovenia
36.1.1.1	1187832720906	11	slovenia	de.mendelson.util	MecStatusBar	de.mendelson....	slovenia
36.1.1.1.1	1187832720906	11	slovenia	de.mendelson.util	MecStatusBar	de.mendelson....	slovenia
36.1.1.1.1.1	1187832720921	11	slovenia	de.mendelson.util	MecStatusBar	de.mendelson....	slovenia
37	1187832720953	11	slovenia	de.mendelson.scheduler.client	ClientGuiPanel	de.mendelson....	slovenia
38	1187832720953	11	slovenia	de.mendelson.scheduler.client	ClientGuiPanel	de.mendelson....	slovenia
38.1	1187832720953	21	slovenia	de.mendelson.util.rmi	RMISender	de.mendelson....	italy
38.1.1	1187832720968	10	italy	de.mendelson.scheduler.rmi	MecRemoteI...	MecRemoteIm...	italy
38.1.2	1187832720968	11	italy	de.mendelson.scheduler.rmi	MecRemoteI...	MecRemoteIm...	italy
38.1.3	1187832720968	11	italy	de.mendelson.scheduler.rmi	MecRemoteI...	MecRemoteIm...	italy
38.1.4	1187832720968	11	italy	de.mendelson.scheduler.rmi	MecRemoteI...	MecRemoteIm...	italy
38.1.4.1	1187832720968	11	italy	de.mendelson.scheduler	MecTimerTask	de.mendelson....	italy
38.1.4.10	1187832721031	11	italy	de.mendelson.scheduler.server	MecTimer	de.mendelson....	italy
38.1.4.11	1187832721031	11	italy	de.mendelson.scheduler.server	MecTimer	de.mendelson....	italy
38.1.4.12	1187832721031	11	italy	de.mendelson.scheduler.server	MecTimer	de.mendelson....	italy
38.1.4.13	1187832721031	11	italy	de.mendelson.scheduler.server	MecTimer	de.mendelson....	italy
38.1.4.14	1187832721031	11	italy	de.mendelson.scheduler.server	MecTimer	de.mendelson....	italy
38.1.4.2	1187832720968	11	italy	de.mendelson.scheduler	MecTimerTask	de.mendelson....	italy

Figure 6-3: The merged traces for the test case Edit task

Figure 6-4 shows the concept lattice generated from all M-e-c schedule test cases. The interpretation of the lattice is as follows: test cases represent "the lattice objects" and the functions executed by the test cases correspond to the "attributes of objects". From the concept lattice, one can identify an test case generated execution traces by traversing upwards the lattice from the node containing the test case name until the root node is reached. For example, the executed functions of the test case *slovenia_StopTask* ❶ includes all the functions in the nodes which are located in the route (marked with arrows) passing from the node ❶ to the root node ❷, such as *getTask, getCommand* and so on. In above functions, function ❸ *getTask(italy)* and function ❹ *getTask(slovenia)* are the same functions running on different hosts (host italy and host slovenia).

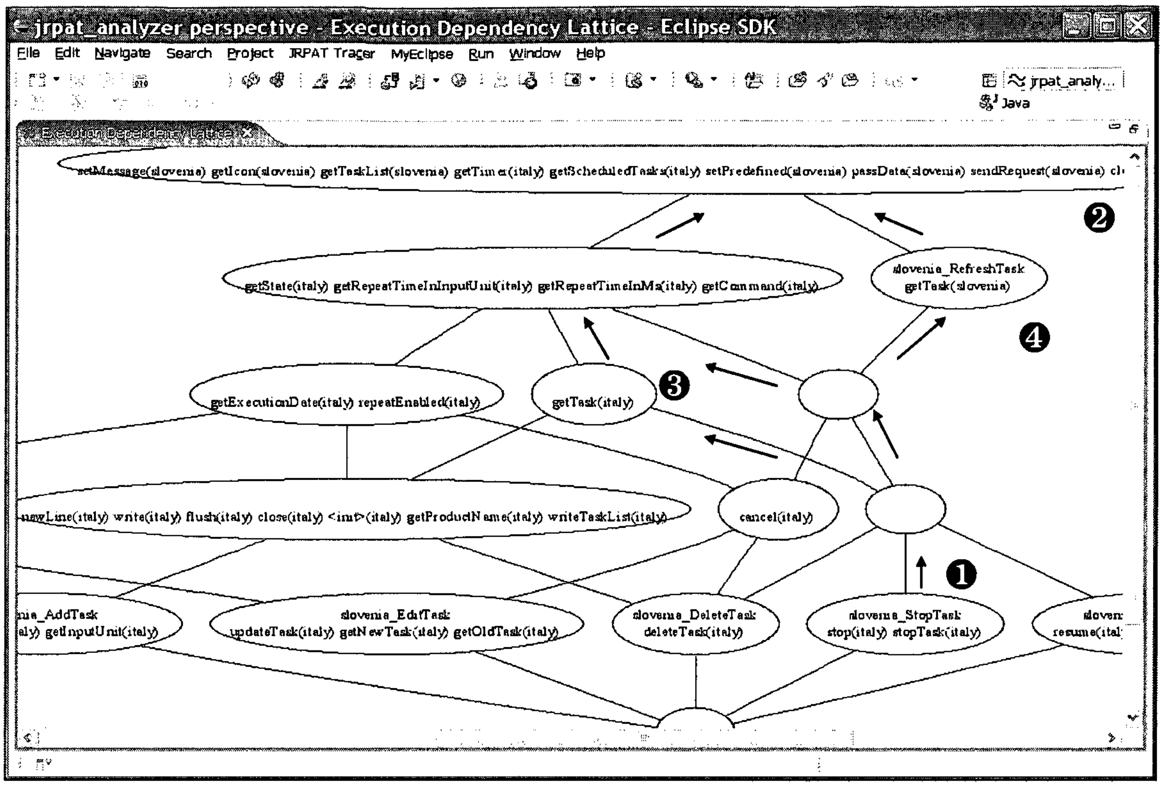

jrpat_analyzer perspective - Execution Dependency Lattice - Eclipse SDK
File Edit Navigate Search Project JRPAT Tracer MyEclipse Run Window Help
jrpat_analy...
Java
Execution Dependency Lattice
setMessage(slovenia) getIcon(slovenia) getTaskList(slovenia) getTimer(italy) getScheduledTasks(italy) setPredefined(slovenia) passData(slovenia) sendRequest(slovenia) cl
❷
getState(italy) getRepeatTimeInInputUnit(italy) getRepeatTimeInMs(italy) getCommand(italy)
slovenia_RefreshTask
getTask(slovenia)
❹
getExecutionDate(italy) repeatEnabled(italy)
getTask(italy)
❸
newLine(italy) write(italy) flush(italy) close(italy) <init>(italy) getProductName(italy) writeTaskList(italy)
cancel(italy)
❶
nia_AddTask
aly) getInputUnit(italy)
slovenia_EditTask
updateTask(italy) getNewTask(italy) getOldTask(italy)
slovenia_DeleteTask
deleteTask(italy)
slovenia_StopTask
stop(italy) stopTask(italy)
sloven
resume(ital

Figure 6-4: The execution dependency lattice for M-e-c schedule

In Figure 6-4, utility functions (used by more test cases) and specific functions (used by less test cases) are separated by their locations in the concept lattice, the former are at the top of the lattice while the later are at the bottom of the lattice. Through this concept lattice, we can also identify that several test cases that share the same functions are more close grouped together.

In Figure 6-5 the regression testing selection result for modifying the function *getTask* (the filled diamond node) are shown. All filled ellipse nodes represent test cases that need to be retested. These nodes are identified by passing the nodes down from the modified node to the bottom node.

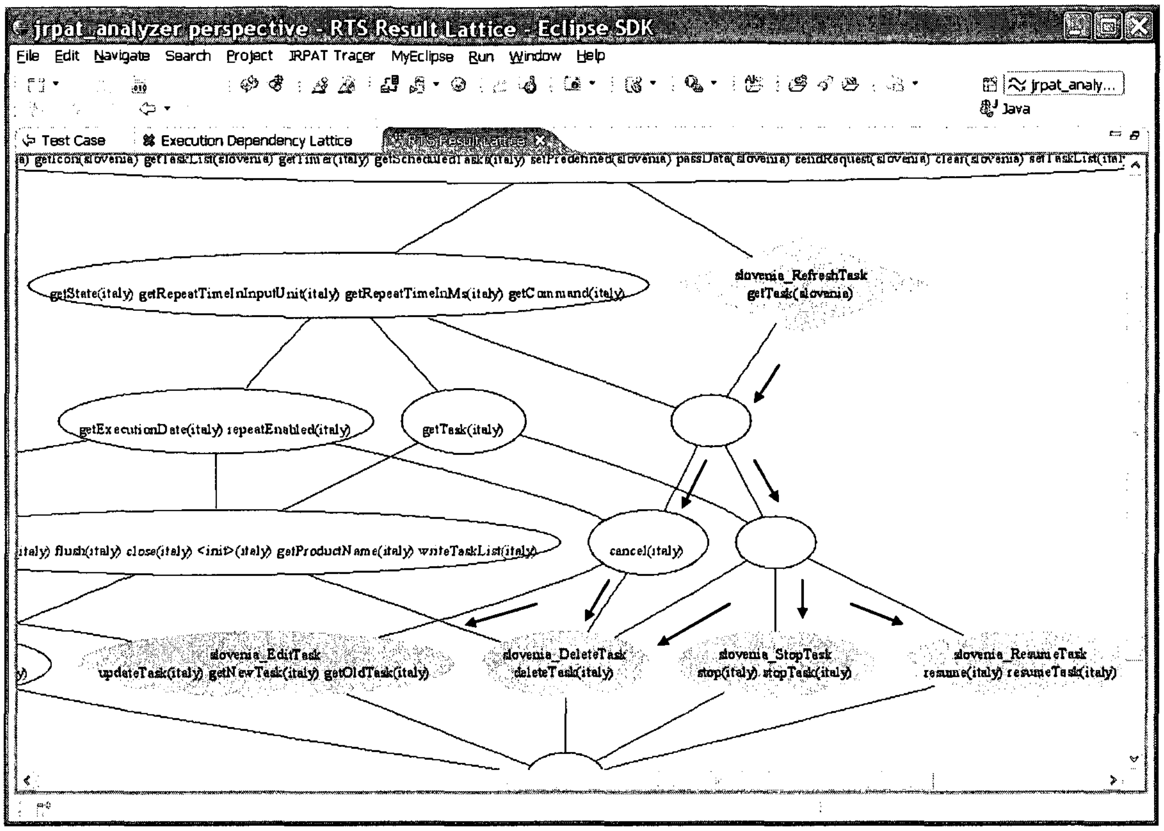

jrpat_analyzer perspective - RTS Result Lattice - Eclipse SDK
File Edit Navigate Search Project JRPAT Tracer MyEclipse Run Window Help
jrpat_analy...
Java
Test Case
Execution Dependency Lattice
RTS Result Lattice
a) getIcon(slovenia) getTaskList(slovenia) getTimer(italy) getScheduledTasks(italy) setPredefined(slovenia) passData(slovenia) sendRequest(slovenia) clear(slovenia) setTaskList(ital
getState(italy) getRepeatTimeInInputUnit(italy) getRepeatTimeInMs(italy) getCommand(italy)
slovenia_RefreshTask
getTask(slovenia)
getExecutionDate(italy) repeatEnabled(italy)
getTask(italy)
italy) flush(italy) close(italy) <init>(italy) getProductName(italy) writeTaskList(italy)
cancel(italy)
slovenia_EditTask
updateTask(italy) getNewTask(italy) getOldTask(italy)
slovenia_DeleteTask
deleteTask(italy)
slovenia_StopTask
stop(italy) stopTask(italy)
slovenia_ResumeTask
resume(italy) resumeTask(italy)

Figure 6-5: The RTS result lattice for M-e-c schedule

6.1.2 Tracing Overhead

It is a known fact that tracing program executions is not free and will cause an additional overhead in terms of execution time and resource requirements. In what follows we present some results with respect to the execution time overhead caused by our approach. We report execution times (in milliseconds) for three test cases, namely “Add”, “Edit” and “Refresh”. We compared for this evaluation the instrumented and non-instrumented versions of M-e-c schedule. The evaluation was performed on two computers running Windows XP with 3.0 GHz Pentium 4 CPU and 1 GB RAM. Each use case was executed several times in order to evaluate the affect of the execution length (memory requirements) on the overhead. In order to be able to create execution traces of various lengths, we introduced loop iterations which basically allowed us to repeatable execute the same use case. Table 6-3 shows observed tracing overhead for the three test cases. The table includes the loop iterations, the execution times of the original programs, the execution times for the instrumented (bytecode level) programs, and the percentage increase of the execution times.

Test Cases	**Loop Iterations**	**5**	**10**	**30**	**50**	**100**
Add	Number of Execution Statements	415	830	2490	4150	8300
	Original Program Execution Time	31ms	63ms	141ms	281ms	578ms
	Instrumented Program Execution Time	78ms	235ms	546ms	1133ms	2425ms
	Increasing Percentage	252%	373%	387%	403%	420%
Edit	Number of Execution Statements	255	510	1530	2550	5100
	Original Program Execution Time	25ms	47ms	63ms	78ms	172ms
	Instrumented Program Execution Time	43ms	76ms	141ms	219ms	517ms
	Increasing Percentage	143%	162%	224%	281%	301%
Refresh	Number of Execution Statements	140	280	840	1400	2800
	Original Program Execution Time	16ms	31ms	45ms	62ms	156ms
	Instrumented Program Execution Time	24ms	57ms	109ms	172ms	451ms
	Increasing Percentage	150%	184%	242%	277%	289%

Table 6-3: The tracing overhead for running three test cases on two computers

Using the test case "Edit Task" as an example, Figure 6-6 illustrates the difference of the execution times between the original program and the instrumented program.

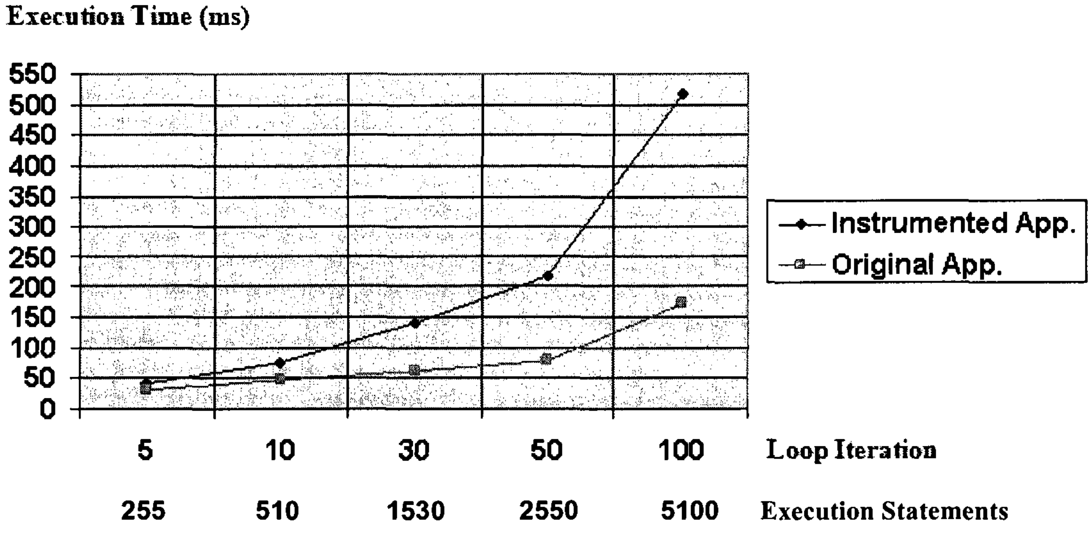


Figure 6-6: The execution overhead comparison of the original M-e-c schedule and the instrumented M-e-c schedule by running the test case "Edit Task"

From the above results, we can observe that bytecode-level instrumentation causes as expected an execution overhead compared to the non-instrumented version. Also as expected the overhead is directly related to the loop iteration and the length of the execution, with the increase in the number of loop iteration and execution statements, the time consumption grows. For instance, the execution time increases 143% when running the program 5 times (255 records in the execution trace), while the execution time increases 301% when running the program 100 times (5100 records in the execution trace). Though the overhead is significant, it is not overwhelming, because the tracing facility does not change the basic system's behavior. In addition, since the RMI-based application is not a real-time application, this kind of overhead could be ignored.

6.2 Case Study 2: ExternalSharing

The goal of the second case study was to evaluate the impact of the external data sharing analysis on the regression test case selection. We were in particular interested in identifying the impact of the external data states (e.g. files, databases) on the test case selection. For the case study we implemented a Java program, ExternalSharing. Figure 6-7 illustrates the test cases dependency of ExternalSharing based on five test cases (T1 to T5). These test cases access four external data variables, with D1 and D2 being two database tables, and D3 and D4 corresponding to two files. For example, Test case 1 (T1) reads data from the table D1 and writes data to the file D4.

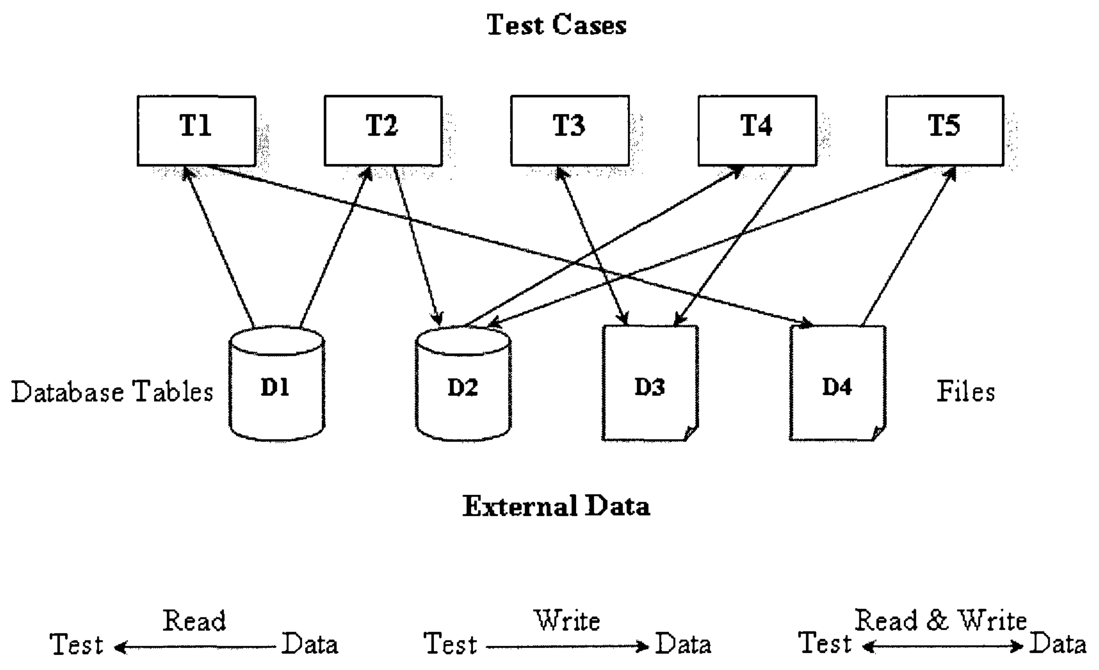
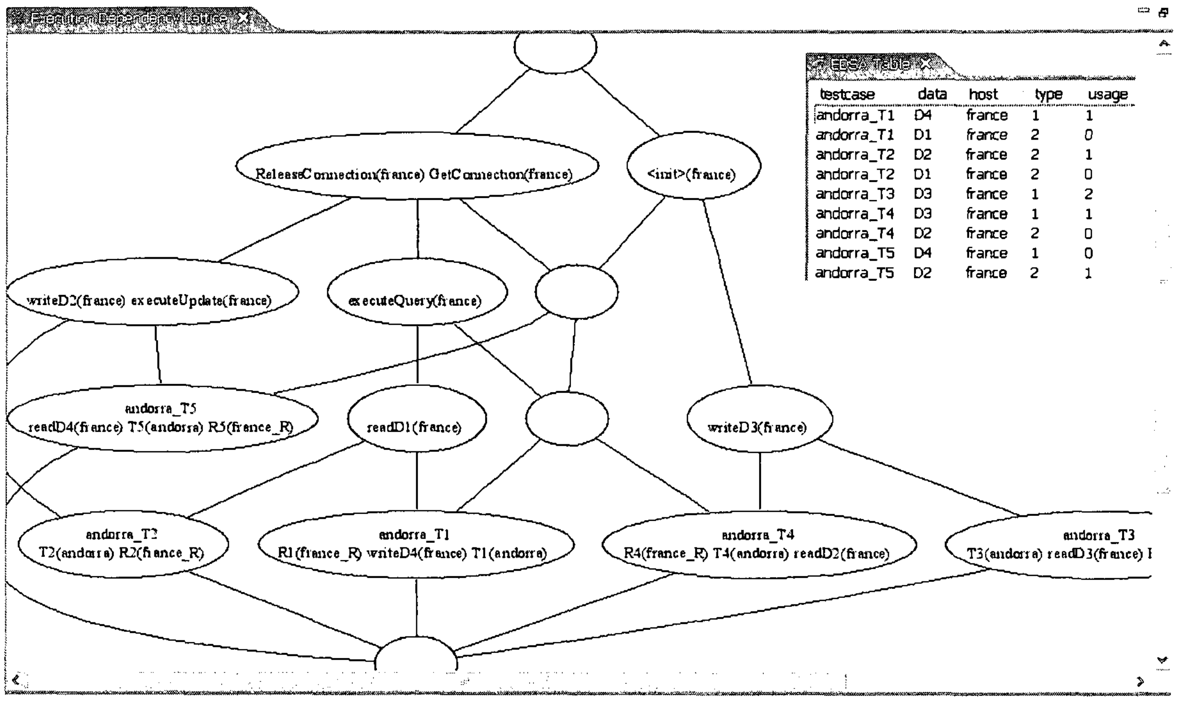


Figure 6-7: The test cases dependency for ExternalSharing

Figure 6-8 shows the execution dependency lattice and the external data sharing analysis (EDSA) table that are generated after executing the five test cases.


Execution Dependency Lattice
ReleaseConnection(france) GetConnection(france)
<init>(france)
writeD2(france) executeUpdate(france)
executeQuery(france)
andorra_T5
readD4(france) T5(andorra) R5(france_R)
readD1(france)
writeD3(france)
andorra_T2
T2(andorra) R2(france_R)
andorra_T1
R1(france_R) writeD4(france) T1(andorra)
andorra_T4
R4(france_R) T4(andorra) readD2(france)
andorra_T3
T3(andorra) readD3(france)
EDSA Table

testcase	data	host	type	usage
andorra_T1	D4	france	1	1
andorra_T1	D1	france	2	0
andorra_T2	D2	france	2	1
andorra_T2	D1	france	2	0
andorra_T3	D3	france	1	2
andorra_T4	D3	france	1	1
andorra_T4	D2	france	2	0
andorra_T5	D4	france	1	0
andorra_T5	D2	france	2	1

Figure 6-8: The execution dependency lattice and the EDSA table

Figure 6-9 shows an example for a regression test cases selection. In this example, the function *readD1* (filled diamond) would be modified as part of a modification request. Using the FCA-based regression testing selection method, test cases T1 and T2 (filled ellipses) were identified as the test cases that at the minimum to be retested. As part of our evaluation, we also performed our external data sharing analysis for the same modification request to see whether additional test cases need to be retested, due to the existence of external data sharing in the system. The external data sharing analysis uncovered the need for three additional test cases T3, T4 and T5 (the filled hexagons) that should be retested after changing the function *readD1*.

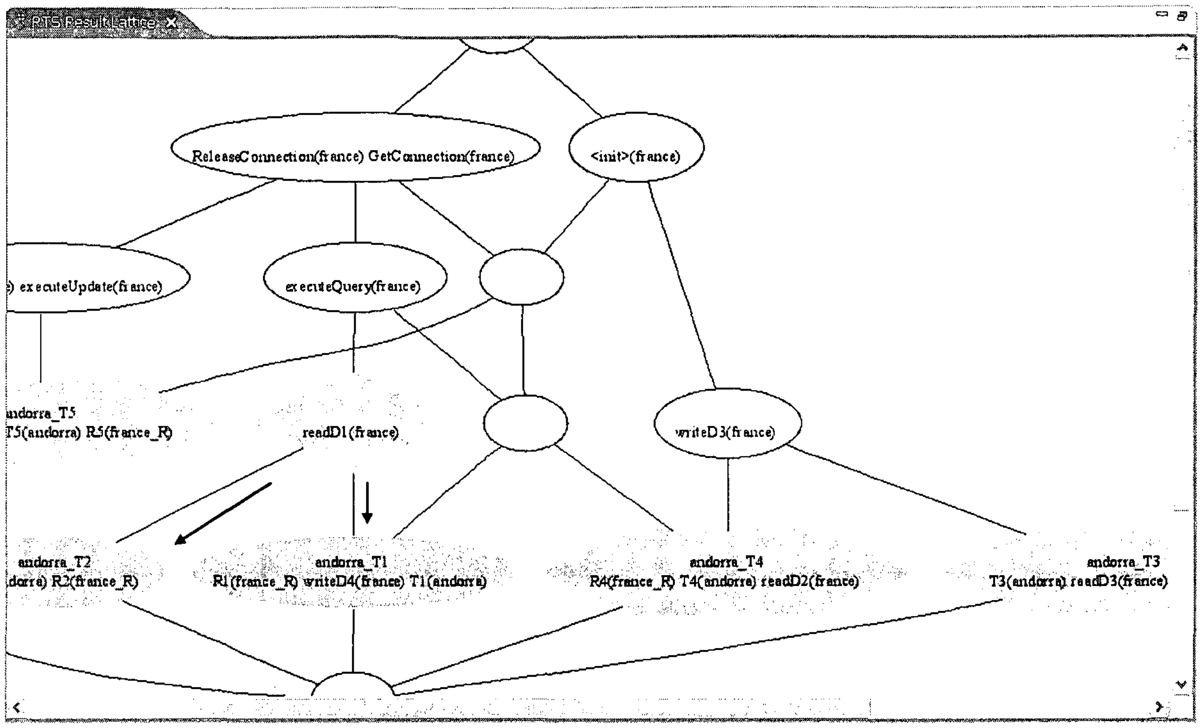


Figure 6-9: The RTS result lattice for ExternalSharing

In what follows we describe in more detail the EDSA-based RTS procedure performed on the two initially selected test cases, T1 and T2. Table 6-4 shows the results of the EDSA analysis for T1. From the table one can identify that T1 has write access to the external data member D4, while T5 has read access to it ❶. During retesting using T1, we also need to re-execute T5. Next, we check the ripple effect of T5. As part of the re-testing strategy, T4 is selected to be retested, because T5 writes data to D2 and T4 reads data from it ❷. The analysis continues with checking T4 and T3 is selected, due to T4 writes the data structure D3 and T3 has read access to the same data ❸. Since analyzing T3 does not select any new test cases, the EDSA-based RTS procedure for T1 is finished. T5, T4 and T3 need to be rerun after retesting T1.

Test Cases / External Data	*T1*	T2	T3	T4	T5
D1	R	R			
D2		W	❷	R	W
D3		❸	RW	W	
D4 ❶	W				R

Table 6-4: The EDSA-based RTS procedure for T1

Similar to the RTS procedure performed on T1, we also applied the analysis on T2, and identified that T4 and T3 need to be retested after rerunning T2. Therefore, after changing *readD1*, all five test cases (T1-T5) need to be rerun. The result lattice is depicted in Figure 6-9.

6.3 Threat to Validity

Based on the results from our initial experiments and observations made during the case studies, our system is able to trace distributed Java RMI applications. The regression test selection analysis was performed with reasonable overhead and only limited human intervention. The test case selection technique presented in this research was not only considering internal control flow but also external data sharing relationships among program entities and test cases. As expected, considering external data states in the change impact analysis affects the set of potentially affected parts in the program and therefore has also directly affects the change impact set.

Our experimental evaluation has shown that our dynamic approach to regression test selection can reduce the imprecision of static analysis techniques in examining causality relationship between local invocations and remote calls. As a result our EDSA approach was able to increase the accuracy of the analysis compared to our initial FCA-based RTS approach. The EDSA included some (i.e. T3, T4, T5 in Figure 6-9) test cases that were originally ignored by our FCA only based approach. The more precise handling of these relationships is due to the collect runtime communications among multiple threads on different nodes. As a result, our approach is capable of performing regression test selection on distributed and multi-threaded programs, while most of other methods are only suitable for analyzing single-threaded sequential programs [CHE94, HSI97, ROT00, WHI92, WHI97, XU07, ZHA06]. Furthermore, in comparison with many heavy weight approaches [ROT97, HAR01B] typically require fine grained traces at the statement level our approach is less expensive, since it is based on runtime data collected at the function level, which allows for smaller traces. Finally, our approach is quite intuitive and easy to use. A graphical representation, a concept lattice, is used to visualize the RTS results, and also simplifies the interpretation of these results. The presented methodology supports a selective regression test selection approach which is almost completely automatic, requiring only a minimum of user intervention. Like the instrumentation part which is supported by wizard dialogs, and the RTS analysis part takes advantage of an easy-to-use GUI.

It has to be noted that our approach also has some limitations. First of all, even though the approach is based on function-level execution traces; scalability might still remain a major limitation. Our approach currently lacks support for trace optimization and filtering techniques, e.g. pattern matching, sampling. As a result, when applying the approach on some large programs, the execution dependency lattice can become complex and unmanageable. Due to this potential scalability problem, both the concept lattice and the table representation in the system are limiting factors. Moreover, the analysis plug-in, the JRPAT-Analyzer, is theoretically able to perform the RTS method on the applications written in multiple programming languages, since the inputs of it are execution traces, which can be generated in most existing programming environments. However, the JRPAT-Tracer and the underlying tracing plug-in for collecting runtime data are implemented based on AspectJ. Therefore, our approach is presently limited to regression test selection problems encountered when developing or maintaining Java programs. Finally, in order to perform RTS, our approach requires test suites which are traceable to the user functions they cover. The accuracy level of the RTS result depends on the coverage achieved by the existing test suite. If the test cases achieve a poor coverage, our methodology will miss executions which are related to a specified modification, and therefore is not able to provide an accurate RTS result. For this reason our approach is neither minimum nor a safe selective regression testing approach.

Given the fact that our approach is based on the use of FCA for the analysis/clustering of the trace information, some of the existing FCA limitations also will affect our methodology. Firstly, consistency between the actual source code, test cases and the concept lattice becomes an issue. In our current implementation it would be necessary to re-run all the test cases after a completed modification, to update the concept lattice with the new test cases that might have been the result of the previously performed modifications. One way to address this problem would be to apply an incremental lattice update algorithm, which would not require the recreation of the complete concept lattice after each modification to the system. Secondly, scalability of the concept lattice might become an issue for larger software systems. The scalability problem can be addressed by including various visualization techniques, which would allow for different levels of abstractions (e.g. zoom, collapsing, contextual views).

6.4 Related Work

In this section we will discuss and compare our work with existing research that is closely related to ours.

6.4.1 Program Tracing

There exist a number of program tracing approaches, which mainly focus on profiling single-threaded sequential programs [LEE97, GOL03, SEE05, SYS01]. Since these approaches only collect runtime communications within components, they lack the capability of tracing client/server activities across multiple hosts. Our approach differs from these approaches by being able to capture separated trace records from various processes, and examine causality relationship between local invocations and remote calls. As a result, our approach is suitable for tracing multi-threaded distributed systems, especially Java RMI programs. Other approaches to trace Java RMI programs can be found in [KAZ00, LEE00, BRI05]. JaViz [KAZ00] focuses on detailed method-level execution data. It is able to trace distributed Java RMI applications and show the point where the distributed application behavior is worse in a single trace. The drawback of JaViz is its dependency on a modified JVM. VisOK [LEE00] is a visualization tool to debug distributed Java programs. The limitations of this tool are: it modifies the implementation of RMI to trace interactions among remote objects; it cannot be used to find method sequences, since the granularity of its traces is at the class level but not at method level. Differing from these works, our approach leverages AspectJ to reduce the implementation effort. The most closely related work to ours is [BRI05], in which the authors present a method that uses AspectJ as instrumentation strategy to produce execution traces, and then perform reverse engineering sequence diagrams for distributed Java RMI systems. The main disadvantage of this work is that users have to manually analyze the source code of the destination system (i.e. identify the RMI interface classes). Furthermore they also have to perform the instrumentation manually. In comparison to this work [BRI05], the instrumentation procedure within our approach is based on wizard dialogs and only involves limited human interaction.

6.4.2 Distributed System Comprehension

In the literature, several approaches for comprehending distributed applications have been proposed [BRU93, MEN01, MOE01, MOE02, GHO02, BRI05]. In them, BEE++ [BRU93] and X-Ray [MEN01] all aim to comprehend distributed systems written in C/C++. BEE++ uses dynamic method while X-Ray employs static techniques for their underlying analysis. BEE++ performs source code instrumentation to monitor the execution of distributed systems. The execution is considered as a stream of events, and the run-time events are dispatched to various distributed software comprehension tools. X-Ray recovers the architecture of distributed systems relying on the static analysis of C/C++ source code. The client-server relationships are identified using clustering techniques and clues from the source code. Johan Moe et al. proposed a three step method in [MOE01, MOE02], which uses execution trace data to help developers understanding and improving CORBA-based distributed system. First, remote procedure calls are traced using CORBA interceptors. Next, the trace data is parsed to construct RPC call-return sequences, and summary statistics are generated. Finally, a visualization tool is used to study the statistics and look for anomalous behavior. According to the researcher, this method is able to provide a fast overview of the run-time behavior and performance of the system.

Similar to our approach, [GHO02, BRI05] also analyze distributed Java RMI applications. A comprehensive runtime interaction validation strategy for distributed Java RMI applications is studied in [GHO02]. This approach proposes techniques for visualizing interactions, specifying and verifying assertions, and checking design conformance based on system execution traces. Local method sequences are collected after source code instrumentation is performed by using a custom security manager or the *Throwable* class in the Java API. For tracking remote method sequences, the approach leverages RMI logging facility, portable interceptors over RMI-IIOP, or customized RMI classes. [BRI04] addresses a methodology that reverse engineers UML sequence diagrams for distributed Java systems based on RMI. This approach defines two separate metamodels for traces and scenario diagrams, and it also defines the mapping rules between them. By means of the metamodels and the rules, it leverages AspectJ to produce execution traces, and then transforms the traces into scenario diagrams.

However, to the best of our knowledge, no previous work exists on performing regression test selection for distributed Java RMI applications. In this research, we perform the FCA-based regression test selection analysis on Java RMI programs. Our approach combines the benefits of dependency analysis and clustering capabilities of FCA. It collects runtime data of the distributed Java application from multiple hosts, and merges the execution traces for each test case properly to generate the visual representations of the test coverage matrixes. In our approach, different view can be easily generated, and maintainers and managers are able to better understand the impact of a requirement change before actually committing to or implementing the change.

6.4.3 Regression Test Selection

Similar to the program tracing approaches, most of the work on regression test selection has been focused on the sequential programs [CHE94, ROT97, WHI92, HSI97, ROT00, WHI97, HAR01B]. Among these researches, [CHE94, HAR01B, ROT00] explore selective regression testing for C/C++ and Java application by combing static programming analysis and dynamic system tracing. Other approaches utilizes control flow information [ROT97], data flow information [HAR89] or the firewall concept [WHI92] to identity which test cases are associated with modifications. However, to the best of our knowledge, there exists no previous work providing support for regression test selection for distributed programs (such as Java RMI applications).

Furthermore, most traditional RTS approaches have focused only on the change propagation through the internal program state (i.e. variables) manipulation, and do not consider change impacts involving persistent states (i.e. databases, files). Several papers [HAR04, WIL05] have addressed RTS for database-driven applications that take into account the interactions of the program with database states. [HAR04] proposed a regression testing approach for stored procedures in databases. [WIL05] presented a safe regression selection algorithm for database-driven applications. However, compare with our approach, these approaches are typically heavy weight, requiring fine grained traces at the statement level, making them very precise but also computational expensive.

7. Conclusions and Future Work

In this research, we introduced a methodology to support a lightweight FCA-based regression test selection analysis for distributed Java RMI programs. Our approach combines execution trace collection, external data sharing analysis and selective regression test selection. As part of this research we developed a toolkit, the Java RMI-based Programs Analysis Toolkit (JRPAT), to support our methodology and its automation. The JRPAT consists of two Eclipse plug-ins, which are capable to collect distributed execution traces, implement an External Data Sharing Analysis (EDSA) algorithm to establish test cases dependency information, perform regression test case selection, and visualize the result in both textual and graphical (with the help of a external graph drawing software integrated) representations. Using two initial case studies, we finally demonstrated and discussed the applicability of the proposed methodology and its tool support. The major contributions of this thesis can be summarized as follows:

(1) We introduce a novel RTS methodology by means of combining run-time information with Formal Concept Analysis for distributed Java RMI applications.

(2) Introduced an external data sharing analysis to explore the define-use relationship among program components of different test cases due to external data elements. We also performed a RTS analysis to estimate the potential testing effort required prior to implementing an actual modification request.

(3) Designed and developed a proof of concept toolkit, the JRPAT, which implements the proposed methodologies and automates the analysis process. We

showed that this tool can be used to trace distributed Java RMI applications and perform successful regression test selection.

As part of future investigation, we plan to address scalability issues related to tracing and analyzing large-scale Java applications. Since the size of the trace can become very large, also the corresponding execution dependency lattice might become too complex and unmanageable. Potential solutions might include selective tracing, viewing the trace in different level (e.g. object-level, class-level, component-level etc.), or filtering the trace through pattern matching, sampling etc. This would also allow omitting unrelated parts in the concept lattice representation.

Moreover, there is also a need to improve the granularity level of the external data sharing analysis. Each column in the table has to be considered as a separate variable, and the data flow relations existing from the usage of each column need to be traced separately.

Finally, for regression test selection, it would be interesting to develop and apply some prioritizing techniques to allow for a further reduction of the number of test cases, or compare our approach with other selective regression techniques in terms of performance, accuracy and effectiveness.

8. References

[BAL99] T. Ball. *The Concept of Dynamic Analysis.* In Proc. Seventh European Software Eng. Conf. Held Jointly with the Seventh ACM SIGSOFT Symp. Foundations of Software Eng., pages 216-234, Sept. 1999.

[BIR67] G. Birkhoff. *Lattice Theory.* American Mathematical Society, Providence, 2nd Edition, 1967.

[BJO06] Dines Bjorner, *The Role of Domain Engineering in Software Development.* Invited keynote paper for the Special Interest Group of Software Engineering (SIGSE), Information Processing Society of Japan (IPSJ) IPSJ/SIGSE Software Engineering Symposium Tokyo, Japan, Oct. 2006.

[BRI03] L. C. Briand, Y. Labiche, and Y. Miao. *Towards the reverse engineering of UML sequence diagrams.* In Proceedings of the 10th Working Conference on Reverse Engineering (WCRE'03), pages 57–66, 2003.

[BRI05] L.C. Briand, Y. Labiche, and J. Leduc. *Tracing distributed systems executions using AspectJ.* In Proc. of the 21st IEEE International Conference on Software Maintenance (ICSM 2005), pages 81-90, Sep. 2005.

[BRU93] B. Bruegge, T. Gottschalk, and B. Luo. *A framework for dynamic program analysis.* In Conference on Object-Oriented Programming Systems, Languages, and Applications (OOSLA93), Washington, USA, Sep. 1993.

[CHE94] Y.-F. Chen, D. S. Rosenblum, and K.-P. Vo. *TestTube: A system for selective regression testing.* In Int. Conf. Software Engineering, pages 211–220, 1994.

[CHE04] J. Chen, K. Wang, *Experiment on embedding interception service into Java RMI.* In Proc. of International Workshop on Scientific Engineering on Distributed Java Applications. Lecture Notes in Computer Science 2952, pages 48–61, Springer-Verlag, 2004.

[CLO07] Clover, *Frequently Asked Questions.* http://www.cenqua.com/clover/doc/faq.html, retrieved September 2007.

[COR89] T.A. Corbi, *Program Understanding: Challenge for the 1990s*, IBM Systems J., vol. 28, no. 2, pages 294-306, 1989.

[ECL07] The Eclipse Foundation, *The AspectJ Project*, http://www.eclipse.org/aspectj/, retrieved September 2007.

[EIS01] E. Eisenbarth, R. Koschke, and D. Simon. *Aiding Program Comprehension by Static and Dynamic Feature Analysis*. In Proc. Int'l Conf. Software Maintenance (ICSM '01), pages 602-611, Nov. 2001.

[EIS03] T. Eisenbarth, R. Koschke, and D. Simon. *Locating features in source code*. In IEEE Transactions on Software Engineering, 29(3):210–224, March 2003.

[ELR01] T. Elrad, M. Aksits, G. Kiczales, K. Lieberherr, and H. Ossher. *Discussing aspects of AOP*. Communications of the ACM, 44(10):33.38, 2001.

[GHO02] S. Ghosh, N. Bawa, S. Goel, and Y. R. Reddy. *Validating run-time interactions in distributed Java applications*. In IEEE International Conference on Engineering of ComplexComputer Systems, pages 7-16, 2002.

[GOL03] A. Goldberg, K. Havelund. *Instrumentation of Java bytecode for runtime analysis*. In Proc. FormalTechniques for Java-like Programs, Technical Reports from ETH Zurich, Vol. 408, Switzerland, ETH Zurich, 2003.

[GRA01] T. L. Graves, M. J. Harrold, Y. M. Kim, A. Porter, and G. Rothermel, *An Empirical Study of Regression Test Selection Techniques*. ACM Trans. on Software Engineering and Methodology, 10(2), pages184-208, 2001.

[GRE05] O. Greevy and S. Ducasse. *Correlating features and code using a compact two-sided trace analysis approach*. In Proceedings IEEE European Conference on Software Maintenance and Reengineering (CSMR 2005), pages 314–323, Los Alamitos CA, 2005.

[GSC03] T. Gschwind and J. Oberleitner. *Improving dynamic data analysis with aspect-oriented programming*. In Proccedings of the 7th European Conference on Software Maintenance and Reengineering (CSMR2003), Benevento, Italy, March 2003.

[GUE05] Y.-G. Gueheneuc and T. Ziadi. *Automated reverse-engineering of UML v2.0 dynamic models*. In proceedings of the 6th ECOOP Workshop on Object-Oriented Reengineering. Springer-Verlag, July 2005.

[GUP96] R. Gupta, M. J. Harrold, and M.L. Sofa. *Program slicing-based regression testing techniques*. In Software Testing, Verification and Reliability 6 (2), pages 83–111, 1996.

[HAM03] A. Hamou-Lhadj and T. C. Lethbridge, *Techniques for Reducing the Complexity of Object-Oriented Execution Traces*, In Proc. of the 2nd

"DESIGNFEST" on Visualizing Software for Understanding and Analysis (VISSOFT'03), Amsterdam, The Netherlands, pages 35-40, 2003.

[HAM05] A. Hamou-Lhadj, E. Braun, D. Amyot, and T. Lethbridge. *Recovering behavioral design models from execution traces*. In Proceedings of CSMR 2005 (9th European Conference on Software Maintenance and reengineering. IEEE Computer Society Press, 2005.

[HAM06] A. Hamou-Lhadj and T. Lethbridge. *Summarizing the content of large traces to facilitate the understanding of the behaviour of a software system*. In Proceedings of International Conference on Program Comprehension (ICPC 2006), pages 181–190, IEEE Computer Society, 2006.

[HAR89] MJ. Harrold and M.L. Soffa. *An incremental data flow testing tool*. In 6th Int. Conf. Testing Computer Software, Washington, D.C., 1989.

[HAR00] M. J. Harrold. *Testing: a roadmap*. In A. Finkelstein, editor, The Future of Software Engineering, Special Volume pubiished in conjunction with ICSE 2000, 2000.

[HAR01A] R. A. Haraty, N. Mansour, and B. Daou. *Regression testing of database applications*. In Proceedings of the 2001ACM Symposium on Applied Computing (SAC), March 11-14, 2001, Las Vegas, NV, USA, pages 285–289. ACM, 2001.

[HAR01B] M. J. Harrold, J. Jones, T. Li, D. Liang, A. Orso, M. Pennings, S. Sinha, S. A. Spoon, and A. Gujarathi. *Regression test selection for Java software*. In Conf. Object-Oriented Programming Systems, Languages, and Applications, pages 312–326, 2001.

[HAR04] R. A. Haraty, N. Mansour, and B. Daou. *Regression test selection for database applications*. In K. Siau, editor, Advanced Topics in Database Research, volume 3, pages 141- 65. Idea Group, 2004.

[HEU02] D. Heuzeroth, T. Holl and W. Lowe. *Combining Static and Dynamic Analyses to Detect Interaction Patterns*. In Proceedings of the Sixth International Conference on Integrated Design and Process Technology (IDPT), June, 2002.

[HSI97] P. Hsia, X. Li, D. C. Kung, C.-T. Hsu, L. Li, Y. Toyoshima, and C. Chen. *A technique for the selective revalidation of OO software*. J. Software Maintenance, 9(4):217–233, 1997.

[HUA78] J. C. Huang. *Program instrumentation and software testing*. COMPUTER, 10(4), 1978.

[JMO07] *org.jmonde.debug.Trace*, http://www.geocities.com/mcphailmj/Trace/, retrieved September 2007.

[JPR07] *jProf profiler*, http://perfinsp.sourceforge.net/jprof.html, retrieved September 2007.

[KAZ00] I. H. Kazi, D. P. Jose, B. Ben-Hamida, C. J. Hescott, C. Kwok, J. A. Konstan, D. J. Lilja, P. C. Yew. *JaViz: A client/server Java profiling tool.* IBM Systems Journal, Volume 39, Number 1, pp.96-117. 2000.

[KIC97] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J. Loingtier, and J. Irwin. *Aspect-oriented programming*. In Proc. of European Conference on Object-Oriented Programming (ECOOP 97). Lecture Notes in Computer Science,Vol. 1241, pages 220–242. Springer-Verlag, 1997.

[KIC01] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jerey Palm, and William G. Griswold. *Getting Started with AspectJ.* In Communications of the ACM, 44(10): pages 59-65, Oct. 2001.

[LEE97] H. Lee and B. Zorn, *BIT: A Tool for Instrumenting Java Bytecodes*, USENIX Symposium on Internet Technologies and Systems, pages 73-83, December 1997.

[LEE00] D. W. Lee, R. S. Ramakrishna. *VisOk: A Flexible Visualization System for Distributed Java Object Application.* In Proceedings of 14th International Parallel and Distributed Processing Symposium. Cancun, Mexico. 2000.

[LI99] Yuejian Li, Nancy J. Wahl. *An Overview of Regression Testing*, ACM SIGSOFT SoftwareEngineering Notes, vol 24 no 1, pages 69-73, January 1999.

[LIN99] T. Lindholm and F. Yellin, *The Java Virtual Machine Specification*, Second Edition, Sun Microsystems Inc., 1999.

[LIN00] C. Lindig. *Introduction to Formal Concept Analysis.* Harvard University, 2000.

[MAO05] C. Mao, Y. Lu. *Regression Testing for Component-based Software Systems* by Enhancing Change Information. In Proc. of APSEC'05, IEEE Press, pages 611-618, 2005.

[MCC07] McCluskey, Glen, *Java Test Coverage and Instrumentation Toolkits*, Glen McCluskey & Associates LLC, http://www.glenmccl.com/instr/index.htm, retrieved September 2007.

[MEN01] N.C. Mendonca, J. Kramer. *An Approach for Recovering Distributed System Architectures.* In Automated Software Engineering, 2001 (8), pages 311-354, 2001.

[MOE01] Johan Moe, David A. Carr. *Understanding Distributed Systems via Execution Trace Data.* In Ninth International Workshop on Program Comprehension (IWPC'01), pages 60-67, 2001.

[MOE02] Johan Moe, David A. Carr. *Using Execution Trace Data to Improve Distributed Systems.* In Software- Practice and Experience, 32, pages 889-906, 2002.

[OPT07] *OptimizeIt*! The ultimate Java performance profiler, http://www.optimizeit.com/, retrieved September 2007.

[PAB06] Leelahapant Pabhanin, *Predictive Regression Test Selection Technique by means of Formal Concept Analysis.* Thesis of Master, Concordia University, Montreal, Canada, 2006.

[PAU00] W.D. Pauw, G. Sevitsky, E. Jensen. Jinsight: *A tool for visualizing the execution of Java programs*, 2000. http://www-106.ibm.com/developerworks/library/jinsight/, http://www.alphaworks.ibm.com/formula/jinsight.

[ROT96] G. Rothermel and M. Harrold. *Analyzing regression test selection techniques*, IEEE Trans. on Software Engineering, 22(8), pages 529-551, Aug. 1996.

[RIC02] Tamar Richner and Stephane Ducasse. *Using dynamic information for the iterative recovery of collaborations and roles.* In Proceedings IEEE International Conference on Software Maintenance (ICSM 2002), page 34-43, Los Alamitos CA, October 2002.

[ROT97] G. Rothermel and M. J. Harrold. *A safe, efficient regression test selection technique.* ACM Trans. Software Engineering and Methodology, 6(2), pages 173–210, 1997.

[ROT00] G. Rothermel, M. J. Harrold, and J. Dedhia. *Regression test selection for C++ software.* J. Software Testing, Verification and Reliability, 10(2):77–109, 2000.

[SAL06] M. Salah, S. Mancoridis, G. Antoniol, and M. Di Penta. *Scenario-driven dynamic analysis for comprehending large software systems.* In Proc. of 10 th IEEE European Conference on Software Maintenance and Reengineering, pages 71–80, 2006.

[SEE05] A. Seesing and A. Orso. *InsECTJ: A Generic Instrumentation Framework for Collecting Dynamic Information within Eclipse.* In Proc. of the eclipse Technology eXchange (eTX) Workshop at OOPSLA 2005, pages 49-53, Oct. 2005.

[SIL03] G. Silva, L. Schnorr, B. Stein. JRastro: *A Trace Agent for Debugging Multithreaded and Distributed Java Programs,* Proceedings of the 15th Symposium on Computer Architecture and High Performance Computing (SBAC-PAD'03), 2003.

[SNE04] H. Sneed. Reverse *engineering of test cases for selective regression testing.* In Proceedings of the 8th European Conference on Software Maintenance and Reengineering (CSMR'04), pages 69–74, 2004.

[STO97] M.-A. D. Storey, F. D. Fracchia, H. A. Müller. *Cognitive Design Elements to Support the Construction of a Mental Model During Software Visualization.* Proceedings of the IEEE 5th International Workshop on Program Comprehension, Dearburn, Michigan, pages 17-28, May 1997.

[STV05] Bjarte M. Stvold and Thor Kristoffersen, *Analysis of object-oriented programs: a survey,* In Norsk Regnesentral, August 19, 2005.

[SUN99] Sun Microsystems Inc. *Java Virtual Machine Debug Interface (JVMDI),* 1999. http://java.sun.com/products/jdk/1.2/docs/guide/jvmdi/jvmdi.html

[SUN02] Sun Microsystems, Inc. *Java Virtual Machine Profiler Interface (JVMPI),* 2002. http://java.sun.com/j2se/1.4.2/docs/guide/jvmpi/

[SUN04] Sun Microsystems, Inc. *Java Virtual Machine Tool Interface (JVMTI),* 2004. http://java.sun.com/j2se/1.5.0/docs/guide/jvmti/jvmti.html

[SYS01] T. Systä, K. Koskimies, H. A. Müller. *Shimba – An Environment for Reverse Engineering Java Software Systems.* In Software–Practice Practice and Experience, 31(4), pages 371-394, 2001.

[WAN05] Y. Wang, Q. Li, P. Chen, C. Ren. *Dynamic Fan-in and Fan-out Metrics for Program Comprehension.* In Proceedings of the 1st International Workshop on Program Comprehension through Dynamic Analysis (PCODA'05), pages 38-42, 2005.

[WEN03] L. Wendehals. *Improving Design Pattern Instance Recognition by dynamic Analysis.* In Proc. of the ICSE 2003 Workshop on Dynamic Analysis (WODA), Portland, USA, pages 29–32, May 2003.

[WEN04] L. Wendehals. *Specifying Patterns for Dynamic Pattern Instance Recognition with UML 2.0 Sequence Diagrams*. In Proc. of the 6th Workshop Software Reengineering (WSR), Bad Honnef, Germany, Softwaretechnik-Trends, volume 24/2, pages 63–64, May 2004.

[WHI92] L. J. White and H. K. N. Leung. *A firewall concept for both control-flow and data-flow in regression integration testing*. In Int. Conf. Software Maintenance, pages 262–270, 1992.

[WHI97] L. J. White and K. Abdullah. *A firewall approach for regression testing of object-oriented software*. In 10th Annual Software Quality Week, May 1997.

[WIL05] D.Willmor and S. M. Embury. *A safe regression test selection technique for database-driven applications*. In Proc. of the 21st IEEE International Conference on Software Maintenance (ICSM 2005), pages 421-430, Sep. 2005.

[WIL81] R.Wille. *Restructuring Lattice Theory: An Approach Based on Hierarchies of Concepts*. Ordered Sets, I. Rival, ed., NATO Advanced Study Inst., pages 445-470, Sept. 1981.

[XU07] G. Xu, A. Rountev. *Regression Test Selection for AspectJ Software*, In Proceedings of the 29th International Conference on Software Engineering (ICSE07), pages 65–74, May 2007.

[ZAI04] A. Zaidman and S. Demeyer. *Managing trace data volume through a heuristical clustering process based on event execution frequency*. In CSMR'04, pages 329–338, 2004.

[ZAI05] A. Zaidman, T. Calders, S. Demeyer, and J. Paredaens. *Applying webmining techniques to execution traces to support the program comprehension process*. In CSMR'05, pages 134–142, 2005.

[ZHA06] J. Zhao, T. Xie, and N. Li. *Towards regression test selection for AspectJ programs*. In Proc. 2nd workshop on Testing aspect-oriented programs, pages 21–26, July 2006.

[ZHE06] J. Zheng, B. Robinson, L. Williams and K. Smiley. *Applying Regression Test Selection for COTS-based Applications*. In 28th IEEE International Conference on Software Engineering (ICSE'06), pages 512-521, May 2006.

[ZOL04] Zoltan Adam Mann and Karoly Kondorosi. *Tracing system-level communication in distributed systems*. Software: Practice and Experience, Volume 34, Issue 8, pages 727-755, Apr. 2004.