Predictive Regression Test Selection Technique by Means of Formal Concept Analysis

Pabhanin Leelahapant

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

March 2006

Your file Votre référence
ISBN: 0-494-14327-4
Our file Notre référence
ISBN: 0-494-14327-4

ABSTRACT

Predictive Regression Test Selection Technique by Means of Formal Concept Analysis

Pabhanin Leelahapant

Regression testing is an important software maintenance process that is applied to validate the correctness of software modifications. Since regression testing is usually costly, selective regression testing can be an alternative to traditional regression testing, allowing for a reduction of the overall cost associated with testing. Selective regression testing identifies only the test cases that execute parts of a program that can potentially be affected by the modification.

In this thesis, we propose a novel technique to perform selective regression testing by means of a data analysis technique, Formal Concept Analysis. We use the capability of Formal Concept Analysis to structure the commonalities of execution traces derived from existing test cases. Formal Concept Analysis provides information related to program execution dependency among different parts of a program, which can then be used to determine the relationships between a modified program portion and existing test cases. In this research, a novel approach analyzes the program execution dependency to allow for the selection of test cases that should be rerun after the program modification is complete. We have implemented a tool that automates regression test case selection and demonstrates a proof of our concept.

ACKNOWLEDGEMENT

I would like to thank all of those who have supported me throughout my master program. I owe a debt of gratitude to my thesis supervisor, Dr. Juergen Rilling who provided me with invaluable guidance in conducting research and has patiently reviewed my thesis. I have learnt from him as well how to retain a positive attitude which, to me, is an important resource to accomplish the research goals.

For their kind offers and great efforts in thesis proofreading, special thanks to my dear friends, Jacqueline Hewitt, Cornelia Kupfer, and Sutida Marie Turcot. Their generosity will always be remembered.

DEDICATION

To my parents who offer me unconditional love always

Table of Contents

List of Figures

List of Tables

1. Introduction

One necessary yet costly software maintenance activity is regression testing [ROT96]. Regression testing is the process of validating modified parts of the software to detect whether new errors have been introduced into previously tested code and to provide confidence that modifications are correct [GRA01]. One existing approach to reusing test suites is called *retest-all* technique that reruns all the test cases in the existing test suite. However, in most cases, except for the rare event of a major rewrite, changes to a system in the maintenance phase are usually small and are made to correct problems or incrementally enhance functionality [CHE94]. For such scenarios, the retest-all technique might not be required. An alternative approach to re-test all technique is called 'selective regression testing'. Selective regression testing allows for reducing the number of test cases that need to be rerun to test the modified program by reusing some test cases in the existing test suite. This will not only reduce the number of test cases but also the associated testing cost.

The general goal of regression test selection techniques is to identify a reusable subset of an existing test suite that can be applied to test a modified version of the same program [ROT96]. In this thesis, we apply Formal Concept Analysis (FCA) [WIL82], to support the selection of test cases. FCA is a mathematical approach to data analysis that is gaining wide-spread acceptance for various applications both in software engineering and business [STU03, EMA03, COL012, TOC01, SNE00, LIN97, REP97, TON01, WAT99]. During our research, we surveyed existing FCA-based applications and techniques to investigate the potentials of FCA to benefit program comprehension, impact analysis, and

more specifically selective regression testing. To our knowledge, there currently exists no selective regression testing by means of FCA.

Our main research goal is to apply FCA to selective regression testing. We propose an algorithm that is based on the execution dependency information derived from the FCA to predict the set of test cases that should be re-executed after a software modification is implemented. A tool that utilizes the capability of FCA to generate the execution dependency lattice has been implemented to automate regression test case selection.

The remainder of the thesis is organized as follows: Chapter 2 provides background knowledge related to Formal Concept Analysis, software testing, impact analysis, regression testing, and related researches. Chapter 3 states research goals, research hypotheses, and our problem solving approach. In addition, it reveals the implementation of the system in detail. Chapter 4 elaborates on each step of the problem solving approach through prepared case studies. The validity of the proposed algorithm and hypotheses is also discussed with the result of the analysis. This is followed by the discussion of the advantages as well as limitations of the proposed approach. Finally, Chapter 5 concludes the thesis and discusses potential future work.

2. Background Knowledge

In the following sections, we will introduce background information relevant to Formal Concept Analysis, software testing, impact analysis, and regression testing. These techniques are the theoretical foundation for the research presented in this thesis.

2.1 Formal Concept Analysis

The mathematical theory of Formal Concept Analysis was founded by Birkoff in 1940 [BIR67] and the related data analysis method was introduced by Ganter and Rudolf Wille in 1982 [WIL82]. In what follows, we will present major definitions and theorems related to Formal Concept Analysis, how to understand concept lattices, applications of Formal Concept Analysis, a survey of Formal Concept Analysis tools, and a discussion of Formal Concept Analysis tools.

2.1.1 Definitions and Theorems

Formal Concept Analysis (FCA) [WIL82] is a data analysis method that uses mathematical theory to perform data grouping. It provides a way to find, to structure, and to display relationships between concepts, which consist of objects with common attributes [EIS011, LIN00]. The fundamental FCA definitions and theorems laid by Wille and Ganter [WIL82] are as following [LIN95].

Definition 1: *A context is a triple* $C = (O, \mathcal{A}, \mathcal{R})$ *where* O *and* $\mathcal{A}$ *are sets of objects and attributes respectively and* $\mathcal{R} \subseteq O \times \mathcal{A}$ *is a relation among them.*

A context is normally represented in a relation table or a context table where the rows represent the objects and the columns represent the attributes as shown in Figure 2.1.1.1. The mark "x" held in the cell of a relation table indicates that the object of that particular row contains the attribute of that particular column.

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

Figure 2.1.1.1: An example of context table [LIN00]

Definition 2: For $O \subseteq O$ and $A \subseteq \mathcal{A}$ from a formal context $(O, \mathcal{A}, \mathcal{R})$ define common attributes $\omega(O) \equiv \{a \in \mathcal{A} \mid \forall_0 \in O : (o,a) \in \mathcal{R}\}$ and common objects $\alpha(A) \equiv \{o \in O \mid \forall a \in \mathcal{A} : (o,a) \in \mathcal{R}\}$

Definition 3: A concept $c = (O, A)$ of a context $(O, \mathcal{A}, \mathcal{R})$ is a pair where $O \subseteq O$, $A \subseteq \mathcal{A}$, $\alpha(A) = O$ and $\omega(O) = A$. The extent of c is $\pi_o(c) \equiv O$ while the intent is $\pi_a(c) \equiv A$. The set of all concepts of $(O, \mathcal{A}, \mathcal{R})$ is denoted by $B = (O, \mathcal{A}, \mathcal{R})$

Also, from the relation $\mathcal{R}$, a concept (O, A) could be informally seen as a maximal rectangle as shown in Figure 2.1.1.2. A maximal rectangle refers to the largest rectangle that indicates a grouping of a particular set of objects with common attributes.

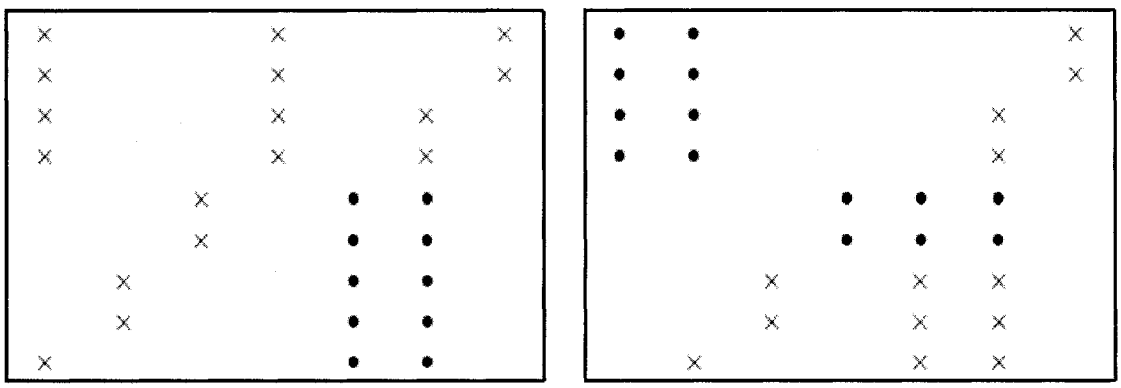


Figure 2.1.1.2: A concept seen as a maximal rectangle in the context table [LIN00]

Definition 4: Concepts $c_1 = (O_1, A_1), c_2 = (O_2, A_2) \in \mathcal{B} = (O, \mathcal{A}, \mathcal{R})$ are ordered by the subconcept-relation $\leq: c_1 \leq c_2$ *iff* $O_1 \subseteq O_2$. The structure of $\mathcal{B}$ and $\leq$ is denoted by $\mathcal{B} = (O, \mathcal{A}, \mathcal{R})$.

Theorem 1 *Let* $(O, \mathcal{A}, \mathcal{R})$ *be a context. Then* $\mathcal{B} = (O, \mathcal{A}, \mathcal{R})$ *is a complete lattice, the concept lattice of* $(O, \mathcal{A}, \mathcal{R})$. *Infimum and Supremum are given as follows:*

$$\prod_{i \in I} (O_i, A_i) = (\bigcap_{i \in I} O_i, \omega(\alpha(\bigcup_{i \in I} A_i))) \text{ and}$$

$$\coprod_{i \in I} (O_i, A_i) = (\alpha(\omega(\bigcup_{i \in I} O_i)), \bigcap_{i \in I} A_i)$$

Theorem 2 *Let* $\mathcal{B} = (O, \mathcal{A}, \mathcal{R})$ *be a concept lattice and let* $\sigma(o)$ *for* $o \in O$ *denote the smallest concept c for which* $o \in \pi_o(c)$ *holds. Let* $\mu(a)$ *for* $a \in \mathcal{A}$ *denote the greatest concept c for which* $a \in \pi_a(c)$ *holds. These concepts can be expressed as follows:*

$$\sigma(o) = (\alpha(\omega(\{o\})), \omega(\{o\})) \text{ and}$$

$$\mu(a) = (\alpha(\{a\}), \omega(\alpha(\{a\})))$$

The set of all concepts from a given context along with the FCA theorem [WIL82] will form a *concept lattice* that contains the complete information of the structure of all concepts. The visualization of concept lattices in hierarchical graphs permits the intuitive understanding of the information contained in the concept lattices.

2.1.2 Understanding Concept Lattices

Concept lattices are usually visualized as hierarchical graphs, often with non-redundant labeling to improve their readability. Each node represents a different concept. The node with an attribute $a \in A$ represents the most general concept that has a in its intent; on the other hand, the node with an object $o \in O$ represents the most specific concept that has o in its extent. Figure 2.1.2 shows an example of concept lattice graph derived from the previous context table in Figure 2.1.1.1.

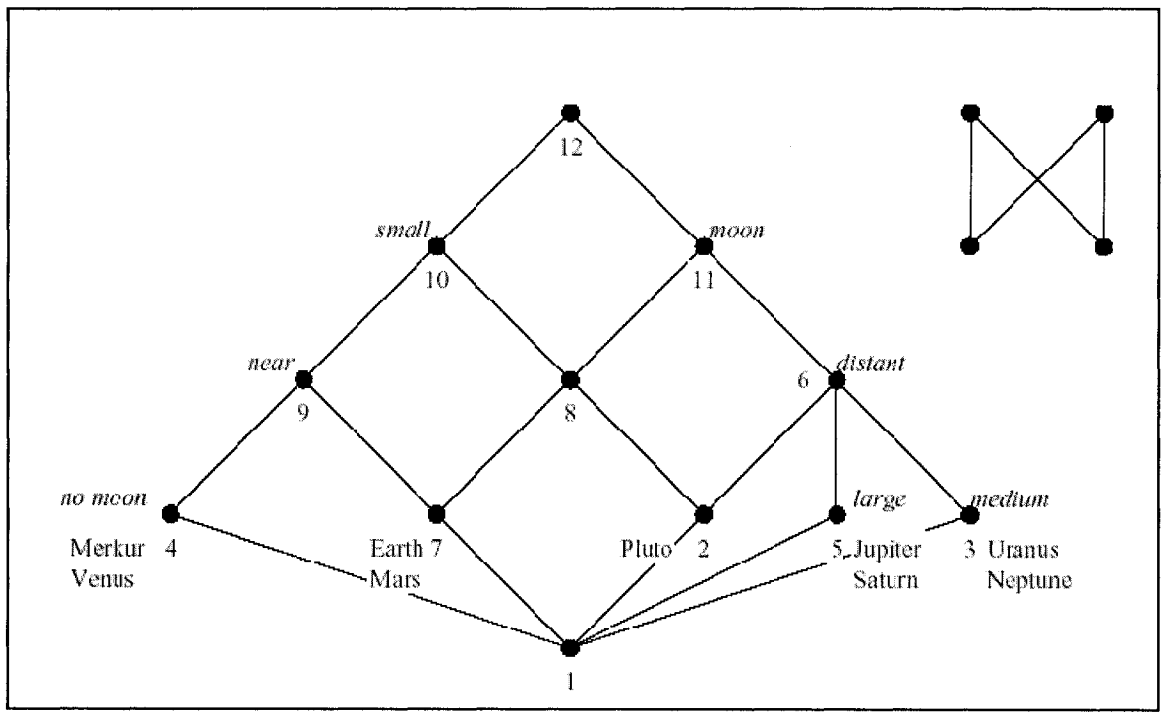


Figure 2.1.2: An example of concept lattice [LIN00]

Table 2.1.2 lists all the concepts in the sample lattice (Figure 2.1.2). Each concept is a set of objects with common attributes. For example, Concept 3 is a pair of the object set, {*Uranus, Neptune*}, and the attribute set containing {*moon, medium, distant*}. It can be interpreted that *moon*, *medium*, and *distant* are all members of the attribute sets of both *Uranus* and *Neptune*. Likewise, Concept 8 which is ({*Earth, Mars, Pluto*}, {*moon, small*}) indicates that in the attribute sets of *Earth, Mars,* and *Pluto* all contain *moon* and *small*.

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}, {*moon, small*})
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 2.1.2: Table of calculated concepts corresponding to the lattice in Figure 2.1.2

According to the FCA definitions and theorems introduced in the previous section, any number of concepts has one greatest subconcept and one greatest superconcept in common. In Figure 2.1.2, the bottom element, Concept 1, ({}, {*moon, medium, distant, small, large, near, nomoon*}) denotes the greatest subconcept which 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}, {}) represents the greatest superconcept which contains an empty set of attributes coupled with all the objects.

Generally, concept lattices are presented with either redundant labeling or non-redundant labeling. The concept lattices with redundant labeling display the complete list of objects and attributes for each concept. Although redundant labeling of lattices ensures that all objects and attributes for each concept are shown, it may cause an information-overloading problem. This problem can be addressed through the use of non-redundant labeling which presents each object and each attribute only once. Figure 2.1.2 is an example of a lattice with non-redundant labeling. In Figure 2.1.2, all names in italic represent attributes, while the others represent objects in the context. The concepts in non-redundant labeling lattice can be interpreted as follows, one must "pass attributes down and pass the objects up" [LIN00]. 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 2.1.2.

2.1.3 Applications of Formal Concept Analysis

In essence, FCA can provide support in identifying sensible groupings of objects with common attributes. There are several applications of Formal Concept Analysis. FCA has proven to be a cost-effective alternative to many applications in business and software engineering. FCA has been applied to various areas of Computer Science such as conceptual clustering, data analysis, information retrieval, knowledge discovery, ontology engineering, and software engineering. This section illustrates the applicability of FCA in different application domains.

2.1.3.1 General FCA-based Applications

Many applications of FCA have been developed for use in business and organizations. Figure 2.1.3.1.1 provides examples of real-life applications of FCA. Figure 2.1.3.1.1-A is the concept lattice resulting from the analysis of flight movement. Figure 2.1.3.1.1-B displays the concept lattice from the analysis of children suffering from diabetes, Figure 2.1.3.1.1-C shows the results of data analysis on a marketing database, and Figure 2.1.3.1.1-D portrays the concept lattice of an IT Security Management application.

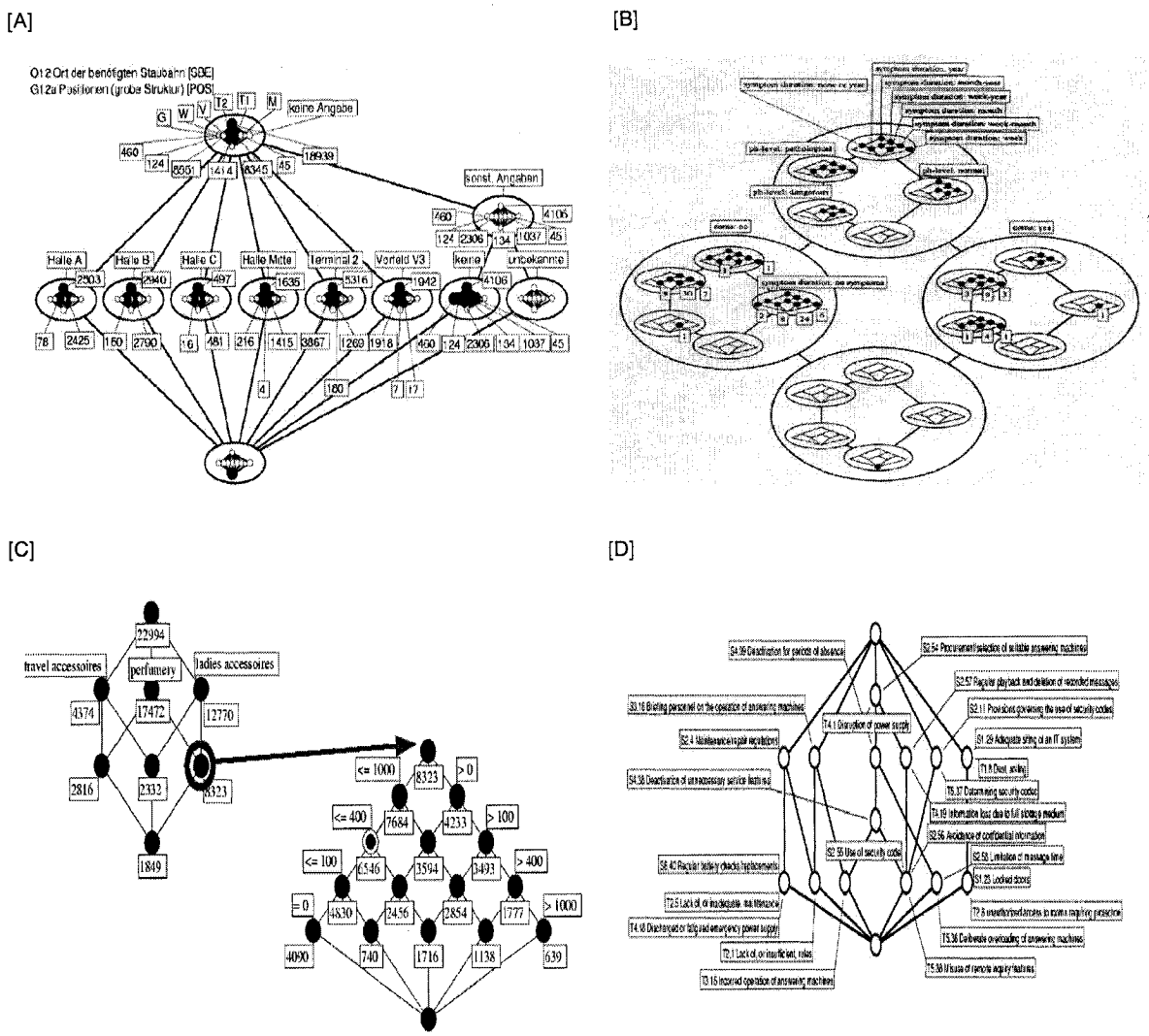


Figure 2.1.3.1.1: Sample applications of FCA [STU03]

Other FCA-based applications include:

- *E-mail management*

Mail-sleuth [EMA03] is a plug-in to MS Outlook [MIC] that is used to organize emails and documents. The program allows content-based virtual file management. Essentially, Mail-sleuth provides a complete context-based knowledge management system that conceptually organizes electronic documents. It includes a knowledge discovery

component with a visualization front-end that can generate specialized views over collections of documents. The program combines FCA with knowledge-based information retrieval technologies. This technique allows interpretations of lattice diagrams permitting inferences to be drawn from data.

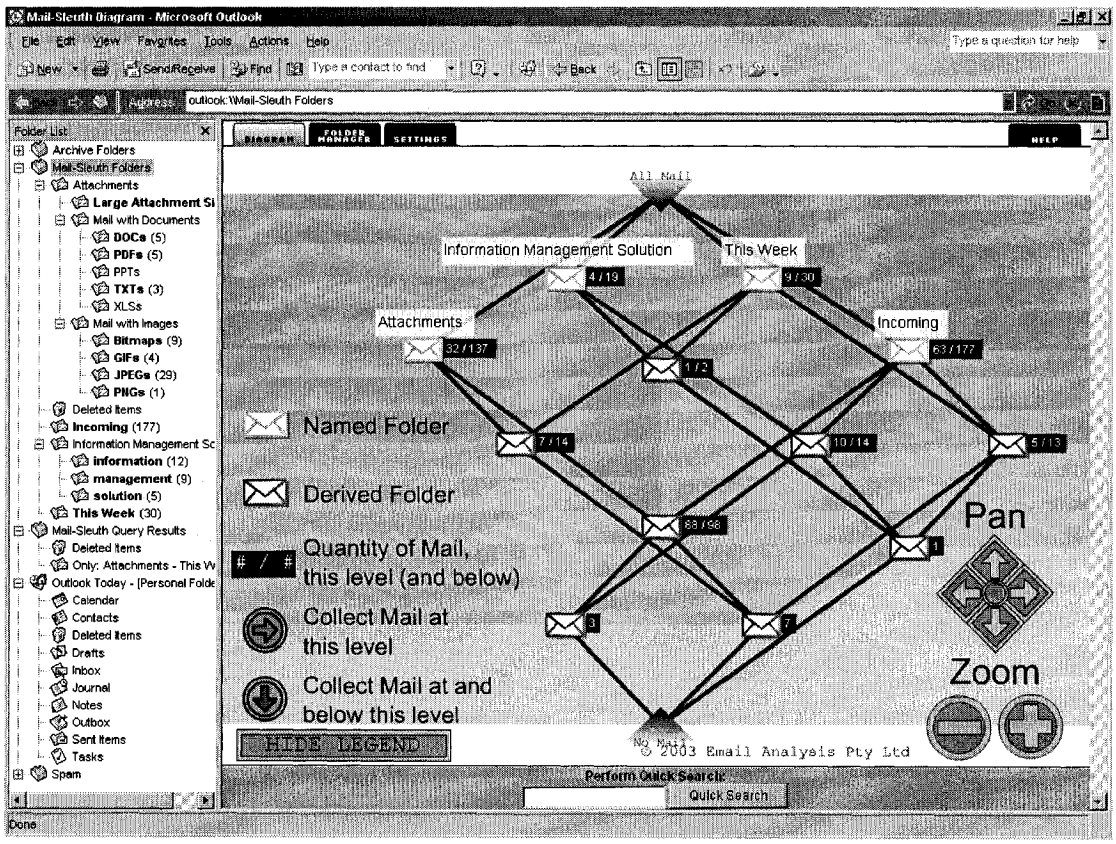


Figure 2.1.3.1.2: The screenshot of Mail-sleuth, a tool for e-mail management [EMA03]

In [COL012], Cole and EkLund proposed an e-mail discovery and information retrieval tool based on FCA. The program, ECA, implements a virtual file structure over a group of e-mails. The concept lattice is applied to search the stored e-mails more flexibly than the available function in general e-mail clients.

- *Scaling relational data*

  Tupleware [TOC01] is a tool for scaling relational data. It implements a scaling approach based on relations. It works on a set of "tuples" which is treated as a crossproduct of the value sets of the columns. These tuples can be read from a text file, queried via SQL, created via command line calls or queried with RDQL from RDF/N3 files. For scaling purposes, the object and attribute sets are selected as columns. If multiple columns are selected, the cross-product is used. The binary relation used is the projection of the original tuples onto the cross-product of the object and attribute sets. The object set is chosen once; afterwards multiple attribute sets can be selected to create a complete CSX file.

- *Document retrieval system*

  1. Score [TOC01]

     Score is a web-based document retrieval system using FCA. It is composed of four basic parts.

     - Indexer: An indexer that scans a number of documents and finds attributes to describe them, typically keywords.
     - Database: A relational database that stores references to the documents, the attributes and their relation.
     - FCA Module: A FCA module that creates a concept lattice from the database and offers a query API.
     - Front-End: A Java Servlet that offers query features to the user

  2. Docco [TOC01]

Docco is a small personal document retrieval system building on top of Apache's indexing and search engine Lucene [TOC01]. It adds user interfaces for indexing and querying to Lucene, where the latter gets enhanced by using FCA's visualization techniques.

3. In [BEC01], Becker and Eklund proposed the technique of document retrieval by using FCA. The concept of this approach represents a query state, where the attribute represents the query and the object represents all the documents matching the query.

- *The Management and Visualization of Document Collections*

  In [COL011], Cole R.J. worked on applying FCA to the organization of documents. This technique used the results of text retrieval system and employed FCA as the mechanism to help organize document collections by calculating the concept lattice from views and their attributes. A new algorithm for visualizing the concept lattice was also proposed by Cole to provide a layout line diagram that fits the concept lattice better than a general-purpose graph drawing algorithm.

- *Browsing Semi-Structured Web Texts*

  In [COL013], Cole and Eklund applied Formal Concept Analysis to solve two problems concerning the nature of web-texts. Their purpose was to provide a mechanism to discover the structure of input sources from unstructured web-data and to automatically extract large and dynamic data when necessary. The case study used

for this technique was the construction of a web-based FCA system to browse classified advertisements for real-estate properties.

2.1.3.2 FCA-based Applications in Software Engineering

In software engineering, Formal Concept Analysis has been applied to program understanding and maintenance. Several concept-based applications use FCA as an approach to perform program module identification, which can be further developed into the application to aid in program restructuring. FCA has also been applied to reengineering class hierarchies, module structure and software configurations. Furthermore, program feature location can be conducted by FCA. The extracted information assists programmers in program understanding, debugging, and modification. Some examples of applications of Formal Concept Analysis in software engineering are listed below.

- *Reengineering Class Hierarchies*

  Snelting and Tip [SNE00] present an application of FCA to class hierarchy reengineering. The goal of their research was to find imperfections in the design of class hierarchies, and to build an automatic tool to aid in the process of reengineering. In their technique, they analysed the usage of the hierarchy by a set of applications that use it. Also, "a fine-grained analysis or the access and subtype relationships between objects, variables and class members is performed" [SNE00]. By means of FCA, the design problems, such as redundant class members or classes that should have been split, were extracted along the process. Snelting and Tip create the binary

relations between class members and variables and perform Formal Concept Analysis. The commonality between class members and variables was then factored out. The result of the whole process of analysis is the alternative hierarchy that is behaviorally equivalent to the initial hierarchy but with each object only containing required members. They also applied this analysis to "a space-optimizing source-to-source transformation" that removed redundant fields from objects. At that time, this analysis was considered one of the most powerful techniques to reengineer class hierarchies. However, it was expensive compared to other available techniques.

- *Reengineering of Configurations*

  Snelting [SNE96] applied Formal Concept Analysis to infer configuration structures from existing source code. He developed the restructuring tool, NORA/RECS. This tool accepts "source code where configuration-specific code pieces are controlled by the pre-processor" [SNE96]. The algorithm then computed a concept lattice which gives users the image of the structure and properties of possible configurations. The lattice produced is a structure of fine-grained dependencies between configuration threads, and the overall quality of configuration structures. The reengineering of configurations is done by analysing interferences which indicated high coupling and low cohesion between configuration threads. NORA/RECS then automated the decomposition of the source code into modules in order to resolve these interference problems. As a result, each module only dealt with a cohesive subset of the configuration space.

- *Software Component Retrieval*

  Lindig [LIN95] proposed a technique to efficiently retrieve reusable software components by using FCA. In his approach, assuming that the reusable software components from a library are indexed with a set of keywords, the user incrementally specified a set of keywords from the requirement. The selected components and the exact set of remaining significant keywords then needed to be further refined before being presented to the user. The FCA algorithm acted as a powerful mechanism to compute a lattice that describes the relation between retrieved components and significant keywords. The results from the experiment demonstrated that after the FCA algorithm was applied, users could select components quickly and precisely. This method of software component retrieval also guarantees that at least one component will be found while none of the conflicting keywords will be specified.

- *Identifying Modules*

  Siff and Reps explained in [REP97] how to use FCA to identify modules in a program that does not designate its modules explicitly. They applied FCA to identify potential modules using both 'positive information', which is the values or types that the module depends on, and 'negative information' which is the ones that the module does not depend on. Each concept obtained from this algorithmic framework represents a potential module of a program. From the calculated concept lattice, they provided the algorithm that identifies possible alternatives to partitioning the program into modules.

- *Identifying objects from legacy code*

  Lindig and Snelting [LIN97] demonstrated how module structures can be presented in the lattice, and how the lattice can be used to assess cohesion and coupling between module candidates. Certain algebraic decompositions of the lattice can lead to the automatic generation of modularization proposals. The method was applied to several legacy code written in Modula-2, FORTRAN, and COBOL. Deursen et al. [DEU99], on the other hand, presented a method to identify objects by semi-automatically restructuring the legacy data structures. The method involved the selection of record fields of interest, the identification of procedures dealing with such fields, and the construction of coherent groups of fields and procedures into candidate classes. They conducted object identification by using cluster and Formal Concept Analysis and then illustrated their effect on a 100 KLOC Cobol system. They compared this clustering with FCA techniques. The authors concluded that FCA solves a number of problems encountered when using cluster analysis. Furthermore, FCA is more suitable for the purpose of object identification because it finds all possible combinations since it is not solely restricted to partitioning.

- *Module restructuring*

  Tonella [TON01] applied Formal Concept Analysis to program module restructuring. This technique was based on the computation of extended concept sub-partition. It provided alternative modularizations characterized by cohesion around the internal structures that are being manipulated. FCA was used as a powerful tool to support module restructuring. Tonella evaluated the ability of Formal Concept Analysis to

determine meaningful modularizations in two ways. Firstly, he used programs without encapsulation violations to assume the absence of violations as an indicator of careful decomposition. Secondly, he implemented restructuring intervention in his case studies to evaluate the feasibility of restructuring and to examine the code structure before and after the intervention.

- *Types and Formal Concept Analysis for legacy systems*

  Kuiper et al. [KUI00] implemented ConceptRefinery which is a tool that allows a software engineer to perform system modification while maintaining a relation with both the original calculated concepts, and the legacy source code. Types for variables and program parameters which were used to conduct Formal Concept Analysis were obtained from type interference in COBOL. The motivation of this tool came from the assumption that the structure in legacy systems can be obtained by FCA; FCA would perform a data analysis using the facts derived from the legacy source code together with type inference from those facts. The results showed that this technique was more precise than their previous experiment which used FCA without type inference.

- *Architectural element matching*

  Water et al. [WAT99] introduced an automated technique for matching the architectural elements using FCA. Their motivation for developing this technique stemmed from their wanting to support forward-engineering and design activities. They wanted to provide a method to help analysts perform architectural syntheses.

They found that FCA helped the analyst in the combination process of architectural synthesis. FCA guided the analyst to match elements from a variety of sources with the same underlying parts of the software system. A concept lattice was derived from the perspectives and descriptive information about the system domain depicted the structure of matching relations.

2.1.4 Survey of Formal Concept Analysis Tools

Several existing tools of Formal Concept Analysis have been developed mainly for research purposes and also as a learning tool in computer science courses. Most tools serve as a context editor, a concept calculator and a concept lattice drawing tool. Current projects of Formal Concept Analysis are listed below.

- *Toscana* [TOS01]

  Toscana is one of the components in the TOCKIT project [TOC01]. Toscana was formerly written in C++ and OWL but suffers from typical legacy problems. As a result, the ToscanaJ project has been developed through the collaboration between DSTC, the University of Queensland and the Technical University of Darmstadt to recreate a Formal Concept Analysis tool called Toscana and to give the FCA community a platform to work with. The ToscanaJ suite comes with editors for different types of Conceptual Information System (with or without relational database in the backend), but editing can be a complex task and this complexity is not exposed to the user of the final system. The principal FCA tools in this project are the following:

1. ToscanaJ - a concept browsing front-end used as a concept viewer and browser. The main features of ToscanaJ are as follows.
    a. Display of simple and nested line diagrams
    b. Color encodes the object contingent sizes (can be changed to extent), in addition, node size can be used for this type of information
    c. The object set of interest can be filtered.
    d. Nodes in the diagram can be clicked to get highlighting as reading help
    e. Diagrams can be exported as SVG, PNG and JPEG. Additional information about the way the diagram was achieved is exported as separate text files, via clipboard or within the SVG file
    f. Different label contents, using data-specific SQL fragments
    g. Additional database views can be opened from the diagram, e.g. a viewer using HTML templates where query elements get resolved
    h. The database viewer interface is designed as a plug-in interface to make it very easy to extend ToscanaJ for other specific purposes
    i. HTML descriptions can be attached to the schema, to diagrams and to attributes
    j. The database viewers can be used for the attributes, e.g. to query an URL from the database which is then opened in an external browser

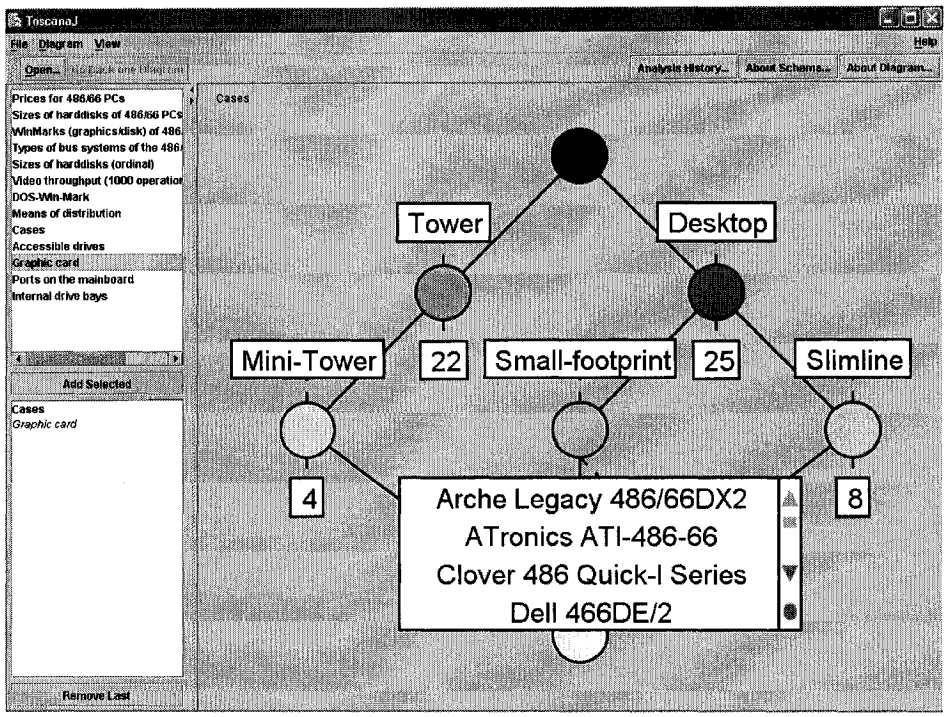


Figure 2.1.4.1: The screenshot of ToscanaJ [TOS01]

2. Elba - an editor for conceptual schemas on relational databases. Database-aware and offering extra tools like exporting SQL scripts. The following features of Elba allow users to create their own conceptual system.
    a. Editing of diagrams and contexts
    b. Database wizard helps users connect to different databases (internal, JDBC, ODBC, MS Access files)
    c. Scale generators help create different types of diagrams
    d. The diagram layout and manipulation is based on n-dimensional structures, making editing diagrams easy while ensuring additive drawings
    e. Different manipulators can be used to override the additivity and to create arbitrary Hasse diagrams

f. A grid can be used as a guide for neat layout. This is supported by all manipulators

g. A very simple XML editor allows adding the descriptions of the different elements

h. SQL scripts can be exported to port small databases from proprietary systems into using the embedded database engine

i. An XML summary of the diagrams can be exported to analyze the realized diagrams, i.e. the coverage of the lattices by the objects in the database

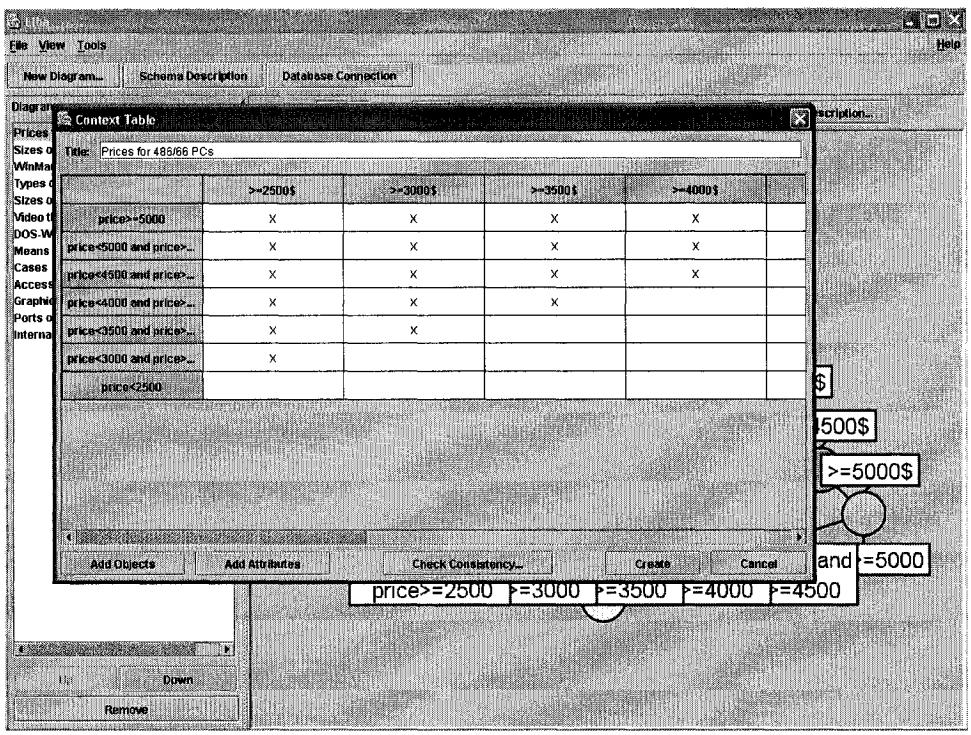


Figure 2.1.4.2: The screenshot of Elba [TOS01]

3. Siena - Siena offers similar features to Elba's without the need of a database. Users can edit conceptual schemas that store their data in memory.

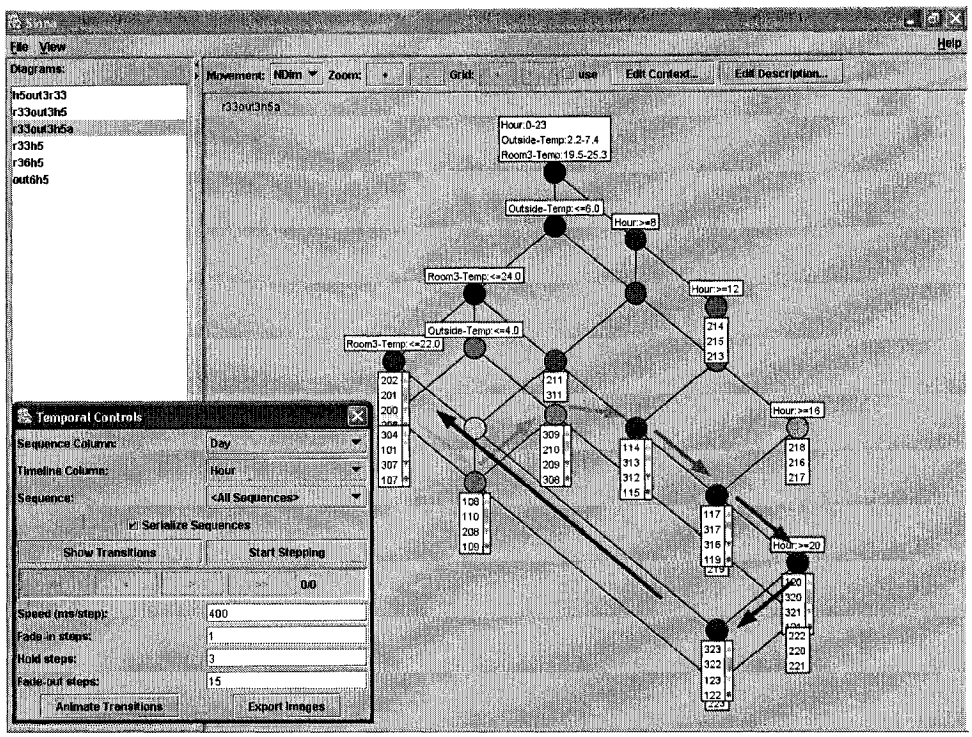


Figure 2.1.4.3: The screenshot of Siena [TOS01]

4. Lucca - An experimental editor that is designed to make use of implication analysis of SQL clauses to allow very explorative and intuitive creation of database-connected systems

- Anaconda [ANA99]

  Anaconda is an interactive editor for ConScript files which runs under Microsoft Windows. ConScript is the file format which is supported by *The Formal Concept Analysis Library* for storing the data structures of Formal Concept Analysis. Anaconda provides in a Multiple Document Interface (MDI) a series of different editors for the different structures. The output can be prepared as well as manipulated independently. For example, a line diagram of the concept lattice of a formal context

can be computed in Anaconda for a direct analysis on the screen or in a printout. However, the main purpose for the usage of Anaconda is to prepare input for Toscana.

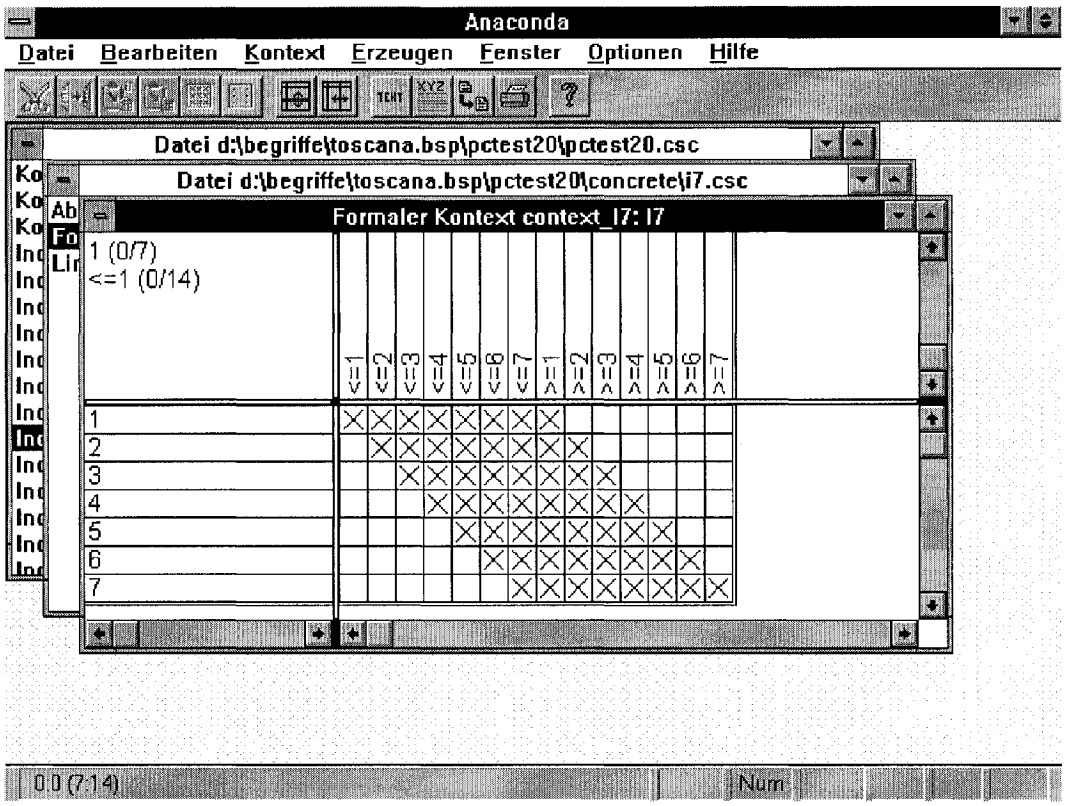


Figure 2.1.4.4: The screenshot of Anaconda [ANA99]

- DOS [DOS99]

  Darmstadt Research Group focuses on the researches in Formal Concept Analysis. It has developed the three following concept-based programs:

  1. ComImp - a DOS program for contexts, concepts, concept lattices and implications
  2. Diagram - a DOS program for drawing line diagrams of concept lattices
  3. MBA - a DOS program for many-valued contexts

- ConExp [CON03]

  ConExp or Concept Explorer implements basic functionality, needed for students and researchers in the field of Formal Concept Analysis. The main features of ConExp are the following.

  1. Context editing
  2. Automatic display of arrow relation
  3. Generation and display of concept lattice
  4. Editing of the concept lattice diagram
  5. Variable display of the diagram
  6. Highlighting of diagram nodes
  7. Calculation of Duquenne-Guigues base of implications
  8. Calculation of association rules

- Formal Concept Calculator [AUE04] – A web-based tool written by Sören Auer. This web-based tool provides a wizard for creating contexts and visualizing concept lattices.

- Online Java Lattice Building Application for Concept Lattices [JAL04] – JaLaBA provides an online tool to create contexts, and manipulate the resulting Concept Lattice. Lattices are graphically displayed as a Hasse-Diagram which is difficult to draw by hand. This application was created as part of the PhD-Thesis of Maarten Janssen, called SIMuLLDA. First, JaLaBA builds lists of the names of objects and attributes, and their relations; from this, it creates a list of FCA concepts. These are

then put into a lattice drawing application to finish up the lattice, and exported to a printable format.

- Concepts [LIN02]

  Lindig has written the Concept program in C running distributed under the GNU Public License [GNU] as portable C source code. Concepts is a command line application running under the UNIX environment. It was written by following Formal Concept Analysis theory to compute a lattice of concepts from a binary relation of objects and attributes. Concepts program's output could be presented in a plain text or in the Graphlet format [BRA99], a graph drawing tool written in C. This project has been extended by several researchers. It is mainly used as a mechanism to derive concept lattice graphs out of object and attribute lists.

Besides open source Projects, there also exist FCA commercial tools/services currently provided by NaviCon [NAV04], a German company, offering commercial services and concept-based software, Navicon Decision Suite, which is composed of Cernato and Toscana.

- Cernato supplies a complete overview of dependency, commonality and variability in the data. The user has full control of the analysis process at each moment and can individually examine and change at any time. Thus the process retains transparency. This program provides data input function and diagram viewer with clear structure. It also supports the unrestricted representation of complex data.

- TOSCANA is a front end Tool for graphical analysis and search data in SQL databases and serves for information system design. Toscana defines objects and characteristics in the data and computes their conceptual structure. These data are then converted into clear and readable diagrams and can also be graphically analyzed and examined by the tool.

2.1.5 Discussion of Formal Concept Analysis Tools

Current tools for Formal Concept Analysis provide the following main features:

- Context Editor: the function that allows user to input concept context conveniently and intuitively.
- Diagram Editor: the interactive tool for editing nodes and their properties. The edited data, in return, will be fed back to the context table.
- Diagram Layouter: the function that passes calculated concepts and their relationships to the graph drawing algorithm and visualize the concept lattice in hierarchical pattern.
- Concept Calculator: The function providing algorithm to calculate concepts and their relationships from the input context table.
- Output file format supported: the file format of the program's output.
- Input file format supported: the input file format that the program accepts.

On-line tools such as Formal Concept Calculator [AUE04] and JaLaBA [JAL04] are helpful for students and small research projects. They can serve as tools to create small examples to convey the idea of Formal Concept Analysis. However, the limitations of internet-based applications, such as poor response time, limited reliability and the

inability to integrate these tools with other tools, limits their application for larger projects and data analysis.

ToscanaJ is a well-known FCA tool [TOS01], in particular due to its support for various graphic outputs and input file formats. It allows for a data exchange among FCA tools. The ToscanaJ developer team emphasizes the development of a general Formal Concept Analysis framework and promotes the input file format .csx which is based on XML to be accepted as a standard exchange format in the Formal Concept Analysis research community. However, the input of ToscanaJ is still dependent upon its editor, Elba and Siena. The algorithm to layout the diagram is not an independent module to be easily extended by other researchers.

Concepts by Lindig, C. [LIN02] is another common FCA tool that is used widely in the research community as a data analysis mechanism in FCA-based applications. It provides the algorithm to calculate concepts and its lattice. However, it is written in C and it is only available for the UNIX/LINUX operating system. The output in graph script format is also tied to the program Graphlet [BRA99] that also requires the UNIX operating system. The limited portability is one of the major restrictions of this tool.

ConExp [CON03] supports most of the main features of Formal Concept Analysis. It is assumed to be the easiest tool to perform FCA analysis since users can actually input the context file through simple text file providing a set of objects and attributes. Therefore, this program is suitable for FCA studies as it is simple for the users who do not have a

strong FCA background. Table 2.1.5 is the summary of features supported by each FCA tool.

Feature / Tool	Context Editor	Diagram Editor	Diagram Layouter	Concept Calculator	Output format supported	Input file format supported
ToscanaJ		√	√		**.svg, .png, .jpeg, .pdf, .eps, .emf**	.csx (xml)
Elba	√	√	√		**Sql script, xml**	
Siena	√	√	√		**xml**	
Anaconda	√	√		√	**.csx**	
DOS Suite	√		√	√		
ConExp	√	√	√	√	**.cxt, .cex (xml)**	.cex(xml),cxt, .csv, .oal
Formal Concept Calculator	√	√	√	√	**.ps**	
JaLaBA	√		√	√		
Concepts				√	**script for Graphlet [BRA99]**	.con (text file)
Cernato	√		√	√		.csv, xml
Toscana			√			

Table 2.1.5: Summary table of Formal Concept Analysis Tools

In terms of research and development, most of the tools presented in Figure 2.1.5 are not suitable for use in FCA application development. One reason is that they are not designed to support extensions by other newly developed programs or features. Some are designed

to be learning tools so there are limitations with respect to their scalability. Some tools require input to be in a specific format which has to be created by a specific context editor. Some tools are not open-source program. Thus, they do not facilitate the connectivity to other add-on components. However, the Concepts program by Lindig is designed to be used in both FCA application and research. The fact that Concepts is also an open-source program enables it to be further developed. Researchers can understand FCA algorithm, extract their desired information at any point before the program terminates, and use Concepts as a main mechanism for FCA application. It is also possible to implement an automated FCA application by calling Concepts and transform the output of this program to other formats required by other graph layout programs.

Based on these observations, Concepts was selected as the most suitable basis for this thesis. However, it should be noted that there are some limitations in terms of programming language and portability as previously mentioned. Further discussion and proposed solutions for this issue will be addressed in the Chapter 3.

2.2 Software Testing

Software testing is a process used to uncover errors in a software system. It ensures that the system behaves in a way that meets its specification. Several definitions of software testing have been proposed. One definition is in [MYE04] where software testing is defined as "the process of executing a program (or system) with the intent of finding errors". Testing strategies, in general, are either structural or functional. A structural testing uses the knowledge of program structure and a functional testing uses the knowledge of the system requirements to derive the test data [HAR90].

In software testing process, a test case specification documents the actual values used for input along with the anticipated outputs. A test case also identifies constraints on the test procedures resulting from the use of that specific test case. Test cases are separated from test designs to allow for use in more than one design and to allow for reuse in other situations [IEEE]. Larger test cases may also contain prerequisite states and descriptions. A test suite is the most common term for a collection of test cases. Sometimes, a collection of test cases are called test scripts or test scenarios.

It is commonly a good practice that developers maintain a level of confidence in the software to ensure the software has an acceptable defect rate. An interesting aspect of software testing is test adequacy which decides whether the performed testing is sufficient. This issue was first raised by Goodenough and Gerhart [GOO75]. A test adequacy is a set of rules that imposes requirements on a test set [MCQ05]. It also provides a basis for deciding which test cases to use during testing to increase the

chances that the system faults will be found. Code coverage describes the degree to which the source code of a program has been tested. It can be viewed as a measure in software testing as stated in [MCQ05], "coverage can be used to measure the extent to which an adequacy criterion is satisfied". Several test adequacy criterions and code coverage have been proposed. Three well-known criteria defined by Rapps and Weyuke [RAP82, RAP85] include *All-Paths* (path coverage): a test coverage criteria which requires every possible logical path through the program to be executed at least once during program testing, *All-Edges* (branch coverage): a test coverage which requires that for each decision point each possible branch be executed at least once during program testing, and *All-Nodes* (statement coverage): a test coverage which requires every lines of source code to be executed at least once during program testing.

2.3 Impact Analysis

It has been observed that software modifications can cause ripple effects in existing systems [LEE98]. It is also common that software that has been well tested and has been running properly for long periods of time could yield undesirable results after a few changes have been made. Thus, one of the most challenging tasks in software maintenance is to avoid ripple effects that may occur when a program change is implemented. In order to gain control of maintaining the impact of changes on a software system, software maintainers have to put a great deal of effort into assessing all the potential effects of a change. This process has been termed change impact analysis. Norman Wilde et al defined change impact analysis as "The task of assessing the effects of making a set of changes to a software system." [WIL94].

Therefore, a set of proposed changes is the starting point to track all possible effects on the rest of a system. In [MOR90], the following model to describe the impact analysis process was introduced:

1. Convert proposed change into a system change specification.
2. Extract information from information source and convert into Internal Representation Repository.
3. Calculate change impact for these change proposals. Repeat 1-3 again for other competing change proposals.
4. Develop resource estimates, based on consideration such as size and software complexity.
5. Analyze the cost and benefits of the change request, in the same way as for a new application.
6. The maintenance project manager advises the users of the implementation of the change request, in business rather than in technical terms, for them to decide whether to authorize proceeding with the change.

Figure 2.3.1 illustrates the typical model of impact analysis process.

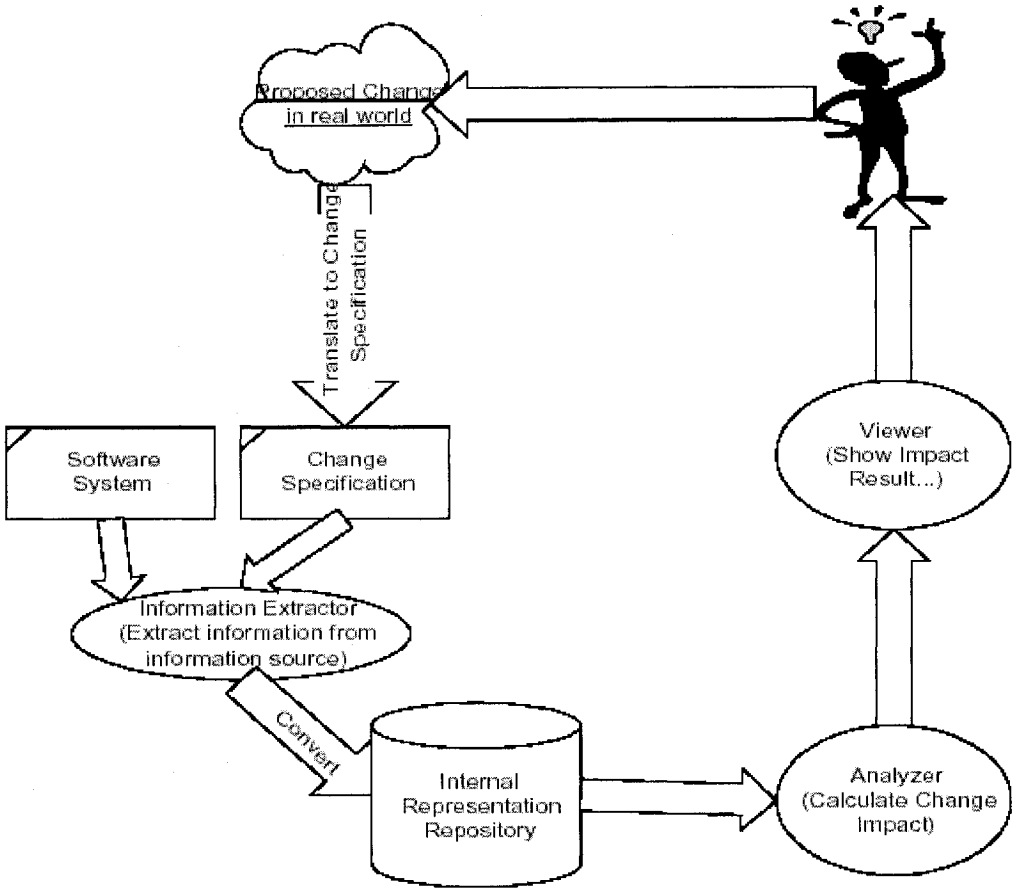


Figure 2.3.1: Typical model of impact analysis process [MOR90]

Impact analysis benefits the software maintenance process in many ways. Information from the analysis allows maintainers to determine which change to select and how to build test suits. Impact analysis accelerates the regression testing process by specifying the parts of the system that are guaranteed not to be affected by the applied changes. Furthermore, impact analysis reduces time and effort spent on program debugging by determining a safe approximation of the changes responsible for a given test's failure [REN04].

Impact analysis is a task that helps indicate the possible effects and drives regression testing efforts. Impact analysis can be performed manually by the maintainers who have the full understanding of the program. To deal with large scale software, conducting impact analysis manually may not be efficient and likely not possible. Different impact analysis techniques have therefore been proposed to ensure software quality and also to automate the impact analysis tasks. Some of these techniques employ dependence analysis [LOY93], data flow analysis [KEA88], program dependency graph [KIM99], execution behavior analysis [REN04], data dependency graphs [LEE00], execution traces analysis/program slicing [ORS04], [LAW03], Island grammars [MON02].

2.4 Regression Testing

2.4.1 Overview and Definition

One necessary, yet costly maintenance activity is regression testing [ROT96]. Regression testing is the process of validating modified parts of the software to ensure that no new errors are introduced into previously tested code and to provide confidence that modifications are correct [GRA01].

Regression testing generally makes use of the test suite from the initial deployment to ensure quality of modified system. One existing approach to reusing tests is called *Retest-all* technique. In this approach, all existing test cases for the system are rerun. The major disadvantage of this strategy is that executing an entire test suite consumes a considerable amount of time and resources due to program size and testing frequency. In most cases, the changes made to a software system during the maintenance phase are usually minor

as they are made to correct the problems or enhance the software functionality. Therefore, the alternative approach to the retest-all technique is selective regression testing technique. It is useful as it potentially reduces the number of test cases and consequently the associated testing cost. Regression test selection is the major process in selective regression testing since it identifies a reusable subset of an existing test suite to test a modified program [ROT96]. They also identify parts of modified programs or specifications that should be retested. A typical selective retest solution is defined in the following steps: [ROT96]

Given program P, its modified version P', and test set T used previously to test P. The goal is to find a way to reuse T that provides 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''

2.4.2 Regression Test Selection Techniques

Many regression test selection techniques have been proposed in the literature. The following classification of regression test selection techniques can be found in [GRA01].

2.4.2.1 Ad Hoc / Random approach

This technique randomly selects a predetermined number of tests from the existing test suite. This technique is often used when test selection tools are not available and when there is time constraint.

2.4.2.2 Retest-All approach

The technique reuses all tests from the test suite from the original deployment.

2.4.2.3 Minimization approaches

These approaches seek to identify a minimal set of tests from the set of test cases, *T,* based on some structural coverage criterion. When the selected set of tests is rerun, it must exercise every added or modified statement for P'. The research in this category includes [HAR90] and [FIS81].

2.4.2.4 Coverage approaches

Like the minimization approaches, they are based on coverage criteria. However, they do not require minimization, instead, they attempt to select every test that exercises modified or affected program parts. Examples for these approaches can be found in [WHI92], [HAR88], and [OST88].

2.4.2.5 Safe approaches

These techniques place an emphasis on program results. They attempt to select every test that will cause the modified program to produce one or more different outputs from *P*. Examples for this approach include [CHE94] and [ROT93].

In what follows, we provide more detail of some regression test selection techniques mentioned in each of the above mentioned categories.

- Hartmann and Robson in [HAR90] focused on the problem of how to identify and select the set of test cases that should be rerun after a program modification. They stress that the test case selection process must be performed systematically to increase the reliability of the regression testing. The authors developed a revalidation technique by extending the Fischer algorithm [FIS81]. Fischer developed a revalidation strategy and test criteria based on a zero-one integer programming model. The authors applied his strategy to several modern, high-level languages. They also implemented a prototype environment based on his methodology to automate the test case selection.

- White and Leung in [WHI92] presented a methodology to perform regression testing at both integration testing and functional testing level. The significance of their research is to present a systematic approach for testing global variables, and to develop a concept of firewall for data-flow module dependencies. A firewall concept was defined to include all affected modules which must be retested. An

approach for testing and regression testing global variables is based upon the firewall concept definition in this research.

- Rothermel et al. in [ROT93] presented a conservative technique for selective regression testing that makes use of control dependency graphs for program versions. The algorithm, called *SelectTests*, uses these graphs to select the tests from the existing test suite that may uncover regression errors in the new version of the program. The algorithm automates the detection of code portions in the new version that differ from the original version and then selects all test cases that traverse the detected area. Therefore, the algorithm is able to perform the selection without the prior knowledge of program modifications. That is, the selection process is based on changed code instead of whether the original code is affected. The authors stated that the method is general since it can handle all language constructs and program modifications, arbitrary programs, language constructs. The approach also facilitates regression testing at the integration and system levels. This technique is considered safe and precise since it does not select tests that cannot traverse changed statements. The algorithm is efficient because it does not need a complete mapping of corresponding parts between original and modified version.

2.5 Related Work

2.5.1 Formal Concept Analysis & Feature Analysis

Eisenbarth et al. work on conducting feature analysis by means of Formal Concept Analysis. In [EIS011], they introduce a technique to support program understanding by deriving feature-component correspondence utilizing dynamic information for the part of the program to be understood. The motivation is to create a program understanding method that is "simple to apply, cost-effective, largely language-independent, and can yield results quickly". They define a usage scenario as the sequences of user inputs that will trigger and execute a feature of a program. From different usage scenarios, Formal Concept Analysis derives a feature-component correspondence graph which reveals the binary relation between features and the executed subprograms. The feature-component correspondence is the basic foundation for this approach. It describes which components are required to implement a set of related features and what the commonalities and variability among sets of related features and components are. In other words, feature-component correspondence is applied for identifying all these components that contribute either to a certain feature or all features to which a component contributes, and the components that are jointly needed to implement only a subset of all features.

In [EIS012] and [EIS013], Eisenbarth et al. present extensions from their previous work on feature-driven program understanding using concept analysis of execution traces found in [EIS011]. The dynamic feature analysis was conducted based on the same method as in the previous research. The difference is that in this approach a combination

of static and dynamic analyses is applied. The static analysis uses the result of this dynamic analysis to specify additional feature-component along the dependency graph. Their current implementation is developed for the UNIX operating system. They wrote short Perl scripts to feed the output from Concepts program [LIN02] into Graphlet [BRA99] which is a graph viewer program to visualize the concept lattice. Then, the static dependency graph is derived from Bauhaus Toolkit [BAU04].

Some of the limitations of these two approaches are now discussed. Since both approaches focus on functional features, they are only applicable to externally visible and executable features. The choice of usage scenarios highly affects the shape of feature-component correspondence and the success of this approach. For example, scenarios that cover too many program functions can cause an unreadable lattice. Furthermore, the application of Eisenbarth et al. is tied to the UNIX platform as its underlying algorithm is provided by Concepts program.

2.5.2 Formal Concept Analysis & Impact Analysis

Tonella in [TON03] compares two impact analysis techniques, a decomposition slice graph with concept lattice of decomposition slices. The decomposition slice graph partitions the program into computations performed on different variables and displays the dependence relation among the computations.

The concept lattice is created by considering a program variable as an object and the attributes are the program slice of the variable. The technique begins with localizing the basic concept lattice nodes which are directly affected by the change. Then, the lattice is traversed upward to seek further impacted nodes affected by the primary ones.

One significant difference between these two techniques is that, for the decomposition slice graph, infimum (least upper bound) and supremum (largest lower bound) are not assured to be unique for any pair of nodes while the concept lattice is. Consequently, a concept lattice of decomposition slices provides more information which is useful for program understanding, in particular, for change impact analysis. The concept lattice of decomposition slices can be applied to assess the impact of change made at given program points by determining whether the change of the computation on one variable will affect the computation on another variable. The idea of this approach is based on the assumption that any change of a particular line of code will affect its upward reachable nodes on the concept lattice.

The main advantage of the concept lattice representation over the decomposition slice graph is that the lattice presents the weak interferences among elements explicitly. The lattice representation of a program is more compact than the program itself. [TON03] However, scalability is still a challenge. Presenting a decomposition slice graph of a large program often results in information overloading which makes the graph unreadable.

2.5.3 Selective Regression Testing

In this section two techniques in selective regression testing are presented that are relevant to this research.

1) *TestTube: A System for Selective Regression Testing* [CHE94]

Chen Rosenblum and Vo. in [CHE94] present a regression test selection technique that detects modified code entities. A system called *TestTube* was implemented to perform the static and dynamic analysis for regression test selection. The technique partitions a software system into basic program entities. Code entities are defined as executable components (e.g. functions) and non-executable components (e.g. storage locations). *TestTube* first monitors the execution of each test unit and analyzes its relationships with the system under test. Next, the approach defines the coverage of each test unit in a test suite so that it can determine which subset of the code entities it covers. Program entities are kept in a database to facilitate the entity comparison. The technique then uses information of program entities, coverage and software modification to select all tests associated with changed entities. When the software is changed, *TestTube* identifies the software entities that were modified. The approach checks if any of previously computed

set of covered entities has changed. If at least one entity has changed, the test unit has to be recomputed to create a new version. The fundamental idea and the benefit of this approach are illustrated in the Figure 2.5.3.

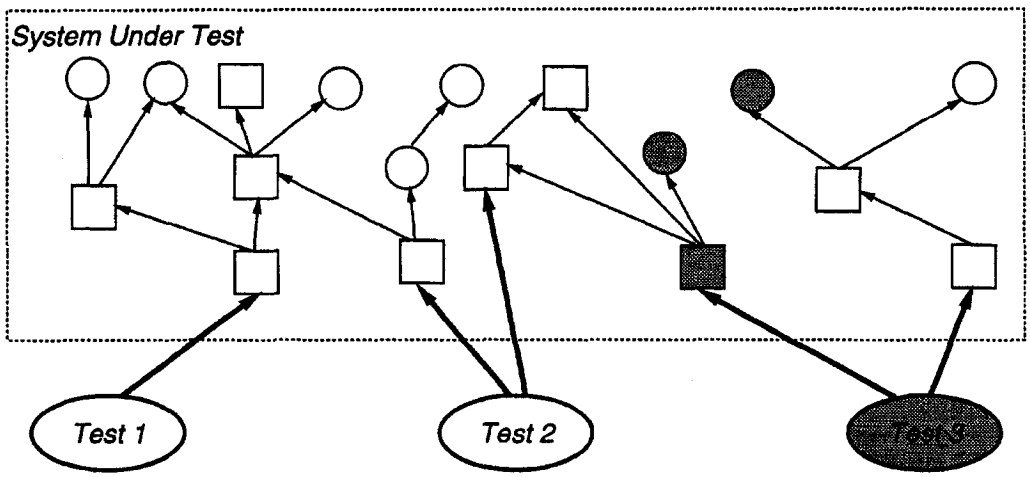


Figure 2.5.3: Selective retesting of a new version [CHE94]

The boxes represent subprograms and the circles represent variables. The arrows represent static and dynamic dependency relationships with the entity at the arrow tail being dependent on the entity at the head. If the shaded entities are modified, all three test units must be rerun under the retest-all strategy while, in *TestTube* approach, Test 1 and Test 2 can be ignored. According to the dependency analysis of the test units and the software entities, Test 3 covers all the modified entities. Consequently, Test 3 is adequate to test the modified system. This describes how *TestTube* reduces the number of test cases and identifies which subset of a test suite must be rerun.

The observed result from the experiments is claimed to reduce more than 50% of the number of test cases needed to test typical software change. *TestTube* has a few

advantages over other techniques. Firstly, it can be used with any chosen test generation and test suite maintenance strategy. Secondly, the system is suitable for both unit-level and system-level testing. Thirdly, the analysis algorithm employed in the technique is computationally inexpensive; however it still provides a good scalability. There are some limitations to this approach which include that the technique was designed for software systems written in C only. Additionally, the authors ignore the problems of generating adequate tests and maintaining the test suite as new versions are created.

2) *Incremental Program Testing Using Program Dependence Graphs* [BAT93]

Bates and Horwitz describe a test selection technique that uses test data adequacy criteria and program slicing. Test adequacy criteria help ensure the quality of a particular test suite. The paper addresses the problems of how to identify which components of the modified program can be tested using files from the old test suite, and which components have been affected by the modification. The proposed solution uses program slicing and program dependence graphs (PDG) which includes all PDG-nodes and PDG-flow-edges to provide safe approximation. The goal is to select the reusable test cases out of the original test suite to exercise the potentially affected statement of the modified program.

The technique first applies program slicing to group PDG from the original program and the modified program into execution classes. Then, the technique identifies the affected components by comparing slices of corresponding components in the original program against the modified program. The technique finally selects the set of tests that exercises components appearing in both execution class and affected component. The algorithms

for selecting the tests to be reused are proposed. The first algorithm is used to identify tests that can be reused to test statements of the modified program. The second one is used to determine which reused tests should be rerun.

There are some advantages of this technique over the incremental testing work reviewed before. One advantage is that the algorithm can handle multiple program modifications. Another advantage is that this approach puts the emphasis on the semantic definition of 'affected' rather than the syntactic definition which may fail to cover the transitive effects of a modification.

3. Contributions

In this chapter, we describe the problem statement and motivation for this research. Furthermore, the research goals, research hypotheses, and the problem solving approach will be presented in detail.

3.1 Problem Statement

Existing regression test case selection techniques and their resulting processes are facing a major challenge in providing automatic support for identifying test cases that have to retest the modified programs.

Several regression test case selection strategies exist; however, most of these approaches are typically very expensive with respect to their computation resource requirements. One approach includes rerunning all possible test cases but at the associated cost with respect to time and hardware resources. In [ROT96], Rothermel et al. evaluate the efficiency of several regression test selection techniques and present a table of comparison (Appendix A). Techniques, on average, require runtime in exponential of the program size and the number of existing test cases, rendering some approaches infeasible. In fact, often such a heavyweight test case selection approach might not even be required. In many cases, it is not worth using such efficient test case selection strategies due to the cost associated with the selection process. For example, it is often the case that project managers only wish to predict initial testing effort and therefore the cost associated with a particular change before determining the detailed testing strategy. In such cases, a predictive test case

selection approach is more appropriate to support the prediction and selection of test cases.

Due to FCA's capability to perform sensible grouping of objects that have common attributes, FCA can help extract dependency information that is required by selective regression testing. To our knowledge, there currently exists no approach that applies Formal Concept Analysis to regression testing in order to identify relevant test cases that can be reused after the system modification is made.

3.2 Research Goals and Research Hypotheses

3.2.1 Research Goals

The following is a list of the detailed research goals according to the presented problem statements. The following primary research goals have been established for this thesis.

- Provision for regression test case selection method. The method should guide software maintainers or any other software's stakeholders in predicting testing effort and indicate all the test cases that should be re-executed after a program change is implemented. The method should enable the reduction of the software testing cost. In addition, the provided method should be language independent.
- Development of a system that performs a predictive regression test case selection by utilizing a FCA algorithm. We consider such system should be able to :
  - determine the dependency between software components and test cases
  - visualize the dependency structure that depicts the coverage of each test case
  - perform a predictive regression test case selection at a reasonable level of accuracy
  - work in multi-platform environments

3.2.2 Research Hypotheses

The following research hypotheses and assumptions are used to set the overall framework for this research.

Hypothesis A:

Based on collected program execution traces, FCA can generate an execution dependency lattice that can identify the test cases that execute a particular software component.

Null Hypothesis A:

FCA cannot generate an execution dependency lattice that can identify the test cases that execute a particular software component.

The *hypothesis A* will hold for the following reasons. It has been shown that FCA can create execution dependency structures. In [EIS011], feature-driven program understanding techniques use the information extracted from the program execution traces (see detail in section 2.5.1). In this case, a usage scenario is defined as an FCA object and execution traces accounting for the usage scenario is defined as its attribute(s). The usage scenarios are first created and run to allow the collection of execution traces. The FCA algorithm then calculates concepts, and builds up the program structure lattice from program execution traces collected during each program run. It has been proven that the lattice of feature-component correspondence can identify all software components (e.g. functions, procedures, statements) that contribute either to a certain feature or all features to which a component contributes.

From FCA-based application presented in [EIS011], we see the potential of the concept lattice to identify all the test cases that execute a particular software component. In our approach, a test case replaces a usage scenario with the basic idea being similar to what is presented in [EIS011]. Thus, in the FCA context, test cases are objects and the execution traces accounting for each test case are their attributes. The difference between [EIS011] and this research is the interpretation of the concept lattice as well as the research goals. In [EIS011], each concept in the lattice represents the common source code portion executed by different usage scenarios; while in our research, the concept represents the common source code portion executed by different test cases. Our lattice can identify all the test cases that execute a particular software component for the same reason the lattice in [EIS011] can identify all features to which a component contributes.

To further elaborate on our justification regarding this hypothesis, we derive a sample concept lattice from FCA of program execution traces and test cases (applied to a program in Appendix B), as shown in Figure 3.2.2. Within a concept node the FCA object (test case) is shown on the first line and the attribute (execution trace) is displayed on the second line. Based on the interpretation of the concept lattice derived by using test cases as the objects and execution trace elements as the attributes, each node (concept) in the lattice represents a group of source code statements that are executed by a particular set of test cases. Therefore, we can identify the test cases that execute a particular software component by using FCA and the Null Hypothesis A can be rejected. We call the concept lattice derived from execution traces based on test cases as *execution dependency lattice*.

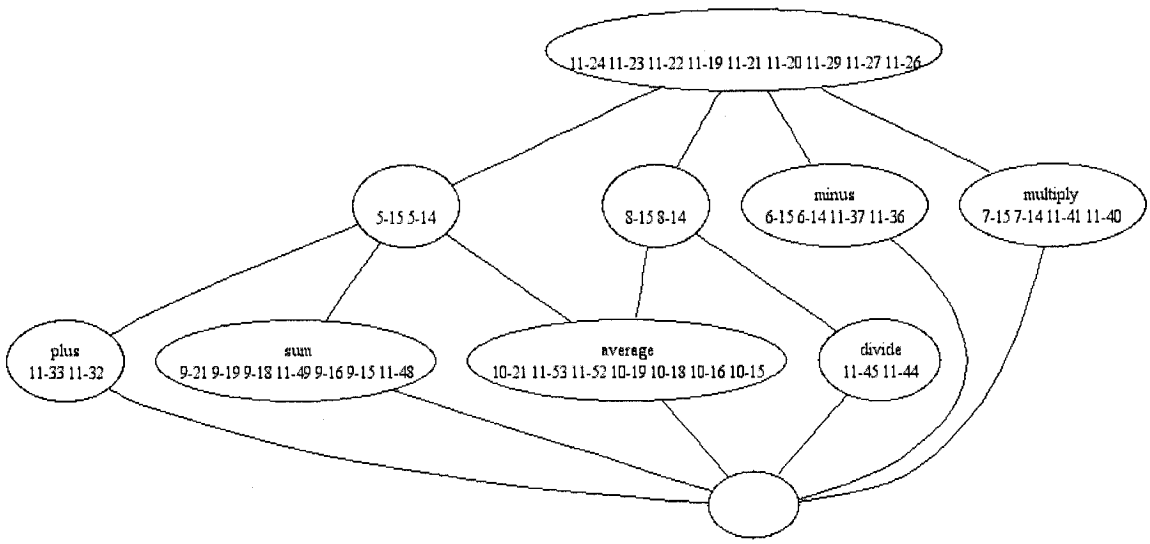


Figure 3.2.2: Sample concept lattice derived from execution traces

Hypothesis B:

The execution dependency lattice resulting from FCA can be utilized to predict and select the test cases that should be re-executed after a software change is implemented.

Null Hypothesis B:

The execution dependency lattice resulting from FCA cannot be utilized to predict and select the test cases that should be re-executed after a software change is implemented.

The hypothesis B will hold due to the following reasons: In software testing, when a change is made to a particular software component, the test cases from the initial test suite that should be re-executed are the ones that have executed that software component. We exclude all other test cases that never execute that software component since they are not directly related to it. It is true that there might be other software components that are

adversely affected by the change and the re-executions of indirectly related test cases are required. However, in this case, we only perform predictive test case selection for being a low-cost alternative of predicting the test cases that potentially have to be re-executed. From hypothesis A, an *execution dependency lattice* resulting from FCA can identify all the test cases that execute a particular software component; therefore, this execution dependency can be used to predict the test cases that should be rerun after the software change is made. The null hypothesis B can be rejected.

3.3 Overview of Problem Solving Approach

The presented research goals are achieved by first providing the FCA algorithm, which is the main mechanism to enable the establishment of execution dependency lattice. Then, we provide the method for regression test case selection based on execution dependency lattice. Finally, we implement the FCA Test Case Selection System or FCA-TCSS to automate predictive test case selection processes using provided FCA algorithm and test case selection method. The overall workflow of FCA-TCSS is shown in Figure 3.3.

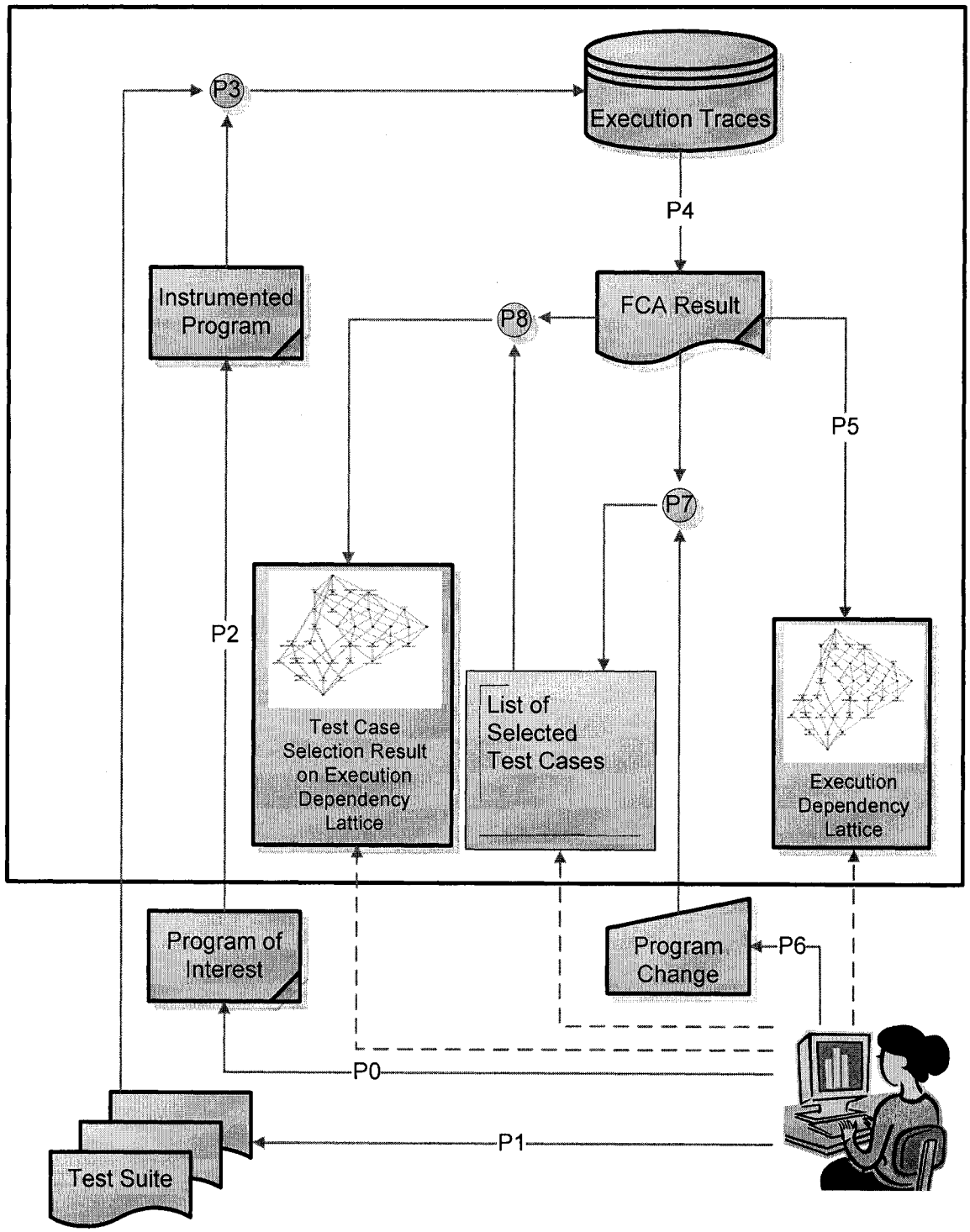


Figure 3.3: FCA Test Case Selection System workflow

The FCA-TCSS performs two main tasks: (1) Generating the program *execution dependency lattice* and (2) Performing predictive regression test case selection. Figure 3.3 illustrates the major steps/activities (P0 –P8) involved as well as how they relate to the different system parts. The steps P0-P8 are briefly described below in sequence of the occurrence.

- ***P0*** *– Select a program of interest*:

  Select a program that is to be analyzed by the FCA-TCSS.

- ***P1*** *- Input a test suite*:

  Enter an existing test suite that can be applied to the program of interest into the FCA-TCSS.

- ***P2*** *– Perform instrumention on the program of interest*:

  Perform instrumentation on the program of interest by the instrumentation tool developed in [CHA04]. The instrumentation will enable the system to keep track of program execution during the program run.

- ***P3*** *- Run the instrumented program with prepared test suite*:

  Run the test suite (P1) for the instrumented program (P2). The resulting program execution traces are collected and stored in a repository for further processing in step P4.

- ***P4*** *- Perform Formal Concept Analysis*:

  The FCA-TCSS retrieves the execution traces from the repository and uses these traces to compute the FCA concepts and the relations among these concepts. The result from the analysis is then further processed by formatting it to the specific format to be used in step P5 and step P7.

- ***P5*** *- Visualize FCA result*:

  The graph visualization software draws the execution dependency lattice from the FCA result from step P4.

- ***P6*** *– Locate program change*:

  Given the *execution dependency lattice* from step P5, user(s) identifies the program position where a particular program change will be performed.

- ***P7*** *- Conduct regression test case selection*:

  The FCA-TCSS performs a test case selection algorithm to select reusable test cases out of test suite from step P1. Such test cases will execute the program statements that are potentially affected by the program modification identified in step P6.

- ***P8*** *– Visualize test case selection results:*

  The FCA-TCSS visualizes the result from test case selection process (P7) by highlighting the modified node and its relevant reusable test cases on the execution dependency lattice.

3.4. Detailed Description of the Problem Solving Approach

In this section, the problem solving approach introduced in section 3.3 is discussed in more detail.

3.4.1 Provision for FCA Algorithm for Multi-platform Environments

As discussed in the survey of FCA tools (see section 2.1.4), the Concepts program by Lindig [LIN02] is our choice of FCA tool due to its extensibility and open-source implementation. However, the major limitation of Lindig's Concepts implementation is its portability. One possible solution to this problem is to re-implement Concepts program, which is originally written in C under the UNIX/LINUX operating system, in Java. Since Java is platform-independent, it increases the portability of FCA algorithm and therefore its applicability. A discussion on FCA algorithm re-development can be divided into three parts. First, the original FCA algorithm from Concepts program is introduced. Then, the implementation of FCA algorithm in Java is briefly described. The last part focuses on the comparison between the original version and the re-written version.

3.4.1.1 Original FCA algorithm source code in C

Christian Lindig developed a command line application named Concepts. Concepts program provides the algorithm that computes a concept lattice from a binary relation table. Lindig took the theory of Formal Concept Analysis from the textbook *Formale Begriffsanalyse - Mathematische Grundlagen* by Ganter and Wille, published 1996. Concepts is distributed under the GNU Public License [GNU] as portable C source code.

The application is promised to work on a popular UNIX operating system. The algorithm implemented in Concepts has a time complexity that is quadratic in the size of the input relation [LIN02].

3.4.1.2 Implementation of FCA algorithm in Java

The Java version of the FCA algorithm is directly derived from Lindig's FCA C programming language implementation. The process of re-developing the algorithm is based on a divide-and-conquer strategy. The original algorithm is first divided into parts by assigning some checkpoints to where the program yields pre-calculated results. The new program is developed toward each sub-goal to gain the same result as in the original algorithm. During the development, the program comprehension tasks were primarily involved and they were performed manually.

The overview of the Java version of the algorithm in Java is explained step by step below.

1. Create pairs of objects and their attribute list.
2. Create pairs of attributes and the list of objects that each attribute belongs to.
3. Create an ordered list of objects and one for the attributes.
4. Create a relation table which contains a bit set. Each number in a bitset represents the bit that is set corresponding to its position in object/attribute ordered-list.
5. From the relation table, list common attributes for each object.
6. From the relation table, list common objects for each attribute.
7. Calculate all concepts according to the FCA theory from the common attribute and object list by using union and intersection mathematical operation.
8. From the concept list, create a lattice by lining the concepts hierarchically.

9. Create a lattice in the specific format required by graph drawing program.

3.4.1.3 FCA - C versus. Java version

The fundamental idea of the algorithm written in Java is the same as the one written in C, but some implementation differences exist. Some common pre-defined Java classes are used to facilitate the implementation. Some data structures used in the C version have been replaced by Java classes. For example, vector and hash table replace some user-defined data structures. Bitset is used to calculate union/intersection. Another difference is that Concepts application finally emits the lattice in a format suitable for the graph drawing tool Graphplace [JOS94], while the FCA algorithm in Java yields a format for graph visualization software, Graphviz [ATT00]. The Java implementation also gives the option to create the output in Graphplace format to accommodate UNIX users.

3.4.2 Provision for Predictive Regression Test Selection Method

In this section, we present our FCA-based regression test selection method along with its fundamental ideas and assumptions.

The regression test case selection will be performed according to the proposed method: *Starting from the node to be modified, all possible test cases that should be retested will be identified by traversing the execution dependency lattice downward and identify all the reachable leaf nodes (and therefore all the statements that have to be re-executed)*

Figure 3.4.2 shows a sample overview picture of a program *execution dependency lattice* illustrating the proposed approach. Assuming a change to node labelled 13, the shaded leaf nodes are all the test cases expected to be rerun after the change is made.

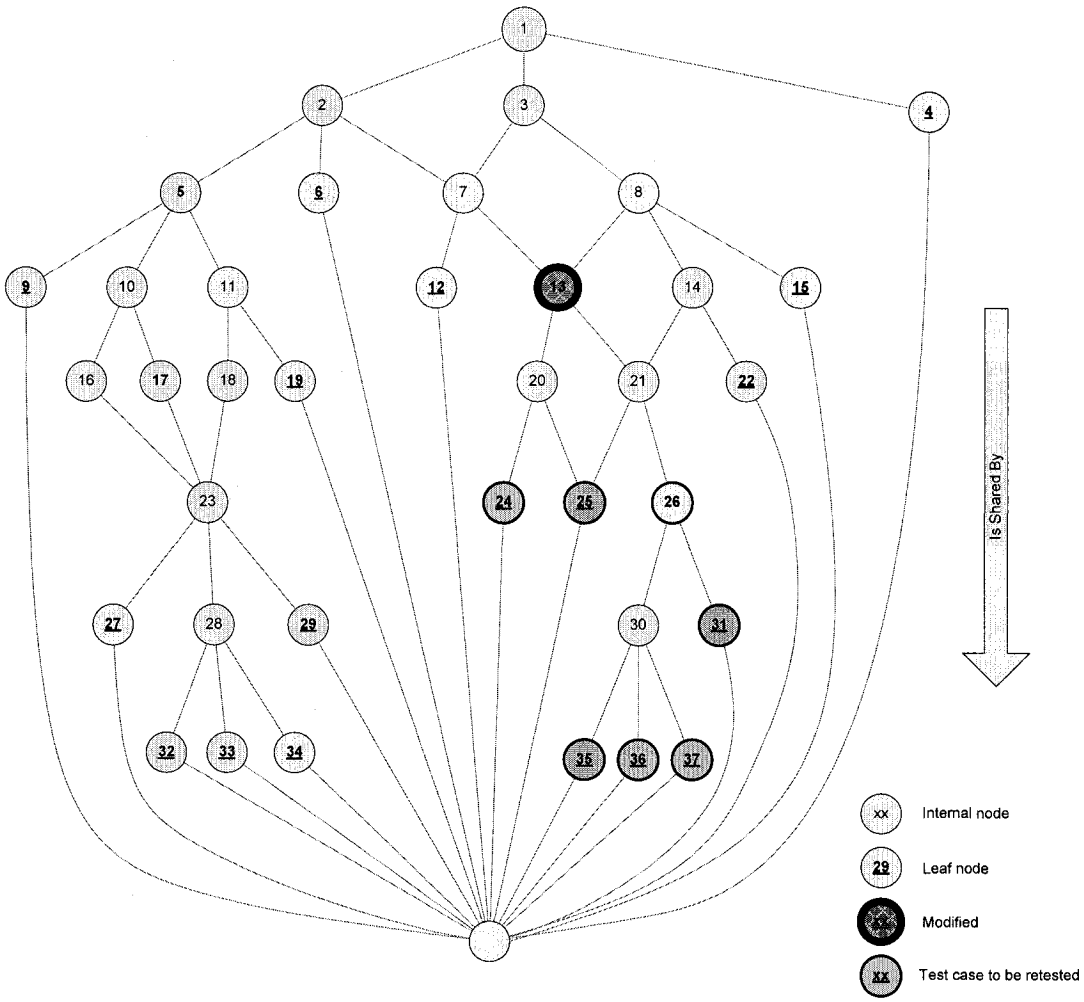


Figure 3.4.2: Overview of execution dependency lattice

In what follows, we justify how our method enables the regression test case selection. We will also discuss how the selection of the reusable test cases can be performed.

Our test case selection technique attempts to select every test case that exercises modified program parts and it can be classified as a coverage approach (see section 2.4.2). As stated in the hypothesis A, we can identify all the test cases that execute a particular software component because each node in the execution dependency lattice represents a group of program statements that are executed by a particular set of test cases. The execution dependency lattice is traversed downwards based on the method of reading a concept in a non-redundant-labelling lattice: passing up the objects and passing down the attributes to the concept of interest (see section 2.1.2). Therefore, when a change is made at a particular concept, all downward reachable objects (test cases) are the test cases that execute the changed software component(s).

However, there might exist some test cases, contained in non-leaf nodes that are included in the path between the modified node and its reachable leaves. Our proposed method excludes all such test cases because each of their attributes (execution trace elements) is already included in the attribute list of the related test cases at the leaf-level. In other words, if such leaf-level test cases are invoked, all the non-leaf-level test cases included on this path will automatically be invoked. Therefore, the non-leaf-level test cases can be excluded and the list of test cases to be retested after the software modification can be obtained by listing the test cases that are contained within all downward reachable leaf nodes.

3.4.3 The implementation of FCA-Test Case Selection System

In this section, we describe in detail all the activities involved in performing regression test selection within the FCA-TCSS, as well as the implementation at each step.

3.4.3.1 Input test suite(s)

After the program of interest is chosen as an input for the FCA-TCSS in step P0, a user specifies the existing test suite that was used to test a program of interest. Such test suite(s) is expected to cover as many program statements as possible or at least the program portion related to where the change is made.

3.4.3.2 Instrumentation of the program of interest

FCA Test Case Selection System integrates the instrumentation tool, developed by Philippe Charland [CHA04] that instruments and collects execution traces during program execution. Step P2 makes use of this tool. Sample input and output for step P2 are demonstrated in Figure 3.4.3.2.1 and Figure 3.4.3.2.2 respectively. The fundamental concept of this instrumentation technique is to insert a small code probe into each program source code line in order to track the execution of the program of interest. The role of this monitoring code is to assign a unique number to each program line and to output this execution information to a repository.

```
package calc;

import java.util.Vector;

public class average{

    public static double calculate(Vector x){
        double result = 0;

                for (int i=0; i<x.size(); i++){
                        result = plus.calculate(result, Double.parseDouble(x.elementAt(i).toString()));
                }
                return divide.calculate(result,x.size());
    }
}
```

Figure 3.4.3.2.1: Source code before being instrumented

```
package calc;

import java.util.Vector;

public class average{

    public static double calculate(Vector x){/*_I*/instr.InstrUtil.showLine(349, 15, "D:\\trace.txt");/*I_*/
        /*_I*/instr.InstrUtil.showLine(349, 16, "D:\\trace.txt");/*I_*/double result = 0;

                /*_I*/{instr.InstrUtil.showLine(349, 18, "D:\\trace.txt");/*I_*/for (int i=0; i<x.size(); i++){
                        /*_I*/{instr.InstrUtil.showLine(349, 19, "D:\\trace.txt");/*I_*/result =
                                plus.calculate(result, Double.parseDouble(x.elementAt(i).toString()));/*_I*/}/*I_*/
                }/*_I*/}/*I_*/
                /*_I*/{instr.InstrUtil.showLine(349, 21, "D:\\trace.txt");/*I_*/return
                                divide.calculate(result,x.size());/*_I*/}/*I_*/
    }
}
```

Figure 3.4.3.2.2: Source code after being instrumented

3.4.3.3 Re-execution of instrumented program with the test suite

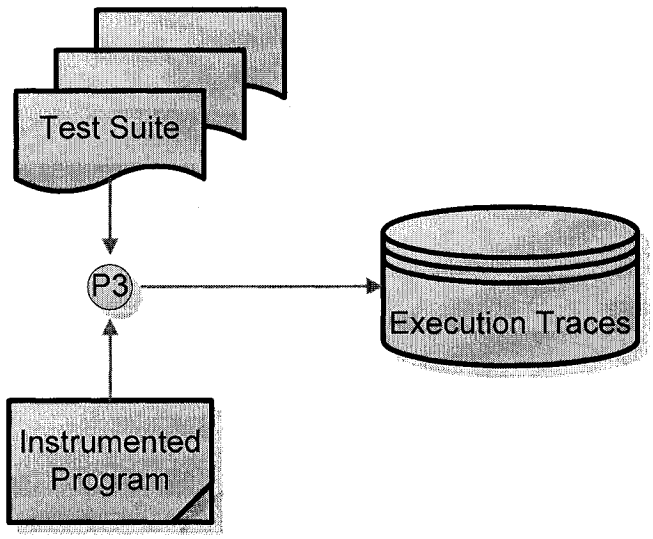


Figure 3.4.3.3.1: Step P3 – Run the instrumented program with the test suite

In step P3, the instrumented program is compiled and executed by the test suite. The resulting output or the execution traces are stored in a repository. At this step, the instrumentation tool first generates a temporary text file to store the execution trace information. The execution traces are then stored in a DBMS repository. The use of a temporary file allows for an overall reduction of the storage overhead required to store the execution traces in the repository. Using the temporary text files also has the following advantages. First, it allows for a use of optimized swapping. Second, it allows for a batch processing of the trace information when storing the information into the repository and therefore reduces the otherwise necessary overhead in connecting/disconnecting from the server based repository.

Figure 3.4.3.3.2 provides an example of an execution trace file stored in the text file. Since a program can contain more than one file (or classes in OO programs), each record is composed of a unique file and line number ID in order to uniquely identify each line in the source code.

```
350 19350 20350 21350 22350 23350 24350 26350 27350 29350
32354 14354 15350 33
```

Figure 3.4.3.3.2: A sample execution trace file in text format

PostgreSQL [POS], an open source software object relational database management system, is the database of choice used by the instrumentation tool. The DBMS manages the execution trace storage and retrieval. A screenshot of PostgreSQL window is displayed in Figure 3.4.3.3.3.

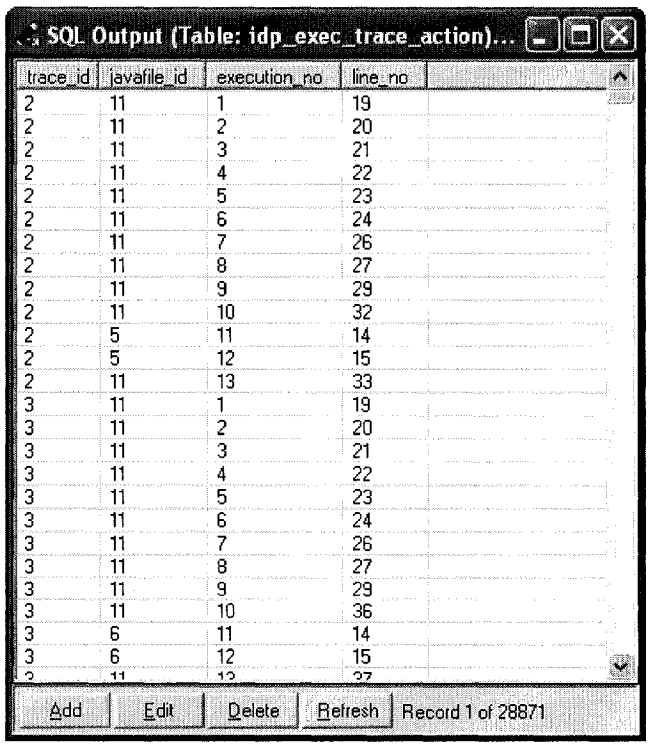


Figure 3.4.3.3.3: A screenshot of a PostgreSQL program.

There is a customized module in the instrumentation tool package which facilitates the transfer of the execution trace from a text file to the DBMS. At this stage, all the execution traces are assumed to be kept in DBMS and ready to be queried. Note that the instrumentation tool is originally designed for general purpose instrumentation and monitoring of Java programs. The main table as shown in the Figure 3.4.3.3.3 stores the data that can be applied to generate execution dependency lattice. However, some fields have to be re-interpreted. For example, 'trace_id' refers to test case number, javafile_id refers to file number, and line_no refers to line number in the source code.

3.4.3.4 Performing Formal Concept Analysis

In step P4, the FCA-TCSS queries the repository for the execution traces and performs the Formal Concept Analysis. As stated earlier in the research hypothesis A, the FCA objects refer to test cases and the attributes refer to program statements accounting for each test case execution. All the information needed for FCA algorithm is prepared and stored in the repository by step P3.

Before the FCA algorithm is executed, the context needs to be prepared in a format required by the algorithm. The context format is simple and stored in a text file. The text file is composed of lines of object name followed by their attributes. Such format looks similar to what follows.

object1 : attribute1.1 attribute1.2 attribute 1.3 .... attribute 1.n

object2 : attribute2.1 attribute2.2 attribute 2.3 .... attribute 2.n

....

objectN : attributeN.1 attributeN.2 attribute N.3 .... attribute N.n

Figure 3.4.3.4 demonstrates a format of FCA context required by FCA Test Case Selection System.

```
35:350-19 350-20 350-21 350-22 350-23 350-24 350-26 350-27 350-29 350-32 354-14 354-15 350-33
36:350-19 350-20 350-21 350-22 350-23 350-24 350-26 350-27 350-29 350-36 352-14 352-15 350-37
37:350-19 350-20 350-21 350-22 350-23 350-24 350-26 350-27 350-29 350-40 353-14 353-15 350-41
38:350-19 350-20 350-21 350-22 350-23 350-24 350-26 350-27 350-29 350-44 351-14 351-15 350-45
39:350-19 350-20 350-21 350-22 350-23 350-24 350-26 350-27 350-29 350-48 355-15 355-16 355-18
   355-19 354-14 354-15 355-19 354-14 354-15 355-19 354-14 354-15 355-19 354-14 354-15 355-21
   350-49
40:350-19 350-20 350-21 350-22 350-23 350-24 350-26 350-27 350-29 350-52 349-15 349-16 349-18
   349-19 354-14 354-15 349-19 354-14 354-15 349-19 354-14 354-15 349-19 354-14 354-15 349-21
   351-14 351-15 350-53
```

Figure 3.4.3.4: A sample FCA context

The FCA-TCSS queries the execution trace information from the repository and prepares the FCA context file in the required format. The system then reads the context file and performs the Formal Concept Analysis. At the end of this process, the calculated concepts as well as the relationships among them are listed and an output file is created in a particular format shown in Figure 3.4.3.5.2.

3.4.3.5 Visualizing the FCA results

The FCA results from step P4 contain structural information that is the input for creating the execution dependency lattice. However, the resulting output is in text-format, making it difficult to comprehend the relationships among the FCA concepts. Therefore, a visual

abstraction and representation of these results is needed to improve their comprehensibility.

A brief survey on graph visualization software was conducted and Graphviz [ATT00] was selected to visualize the lattice structure for many reasons. Graphviz is an open source graph visualization software which provides a way of representing structural information as diagrams of abstract graphs and networks [ATT00]. It is composed of several graph layout programs that "take descriptions of graphs in a simple text language, and make diagrams in several useful formats". It has various features which are useful for diagram drawing, such as options for colors, fonts, tabular node layouts, line styles, hyperlinks, and custom shapes. Output to various formats, such as images, SVG for web pages, Postscript for inclusion in PDF, or display in an interactive graph browser is supported. Figure 3.4.3.5.1 displays a screenshot of Graphviz program.

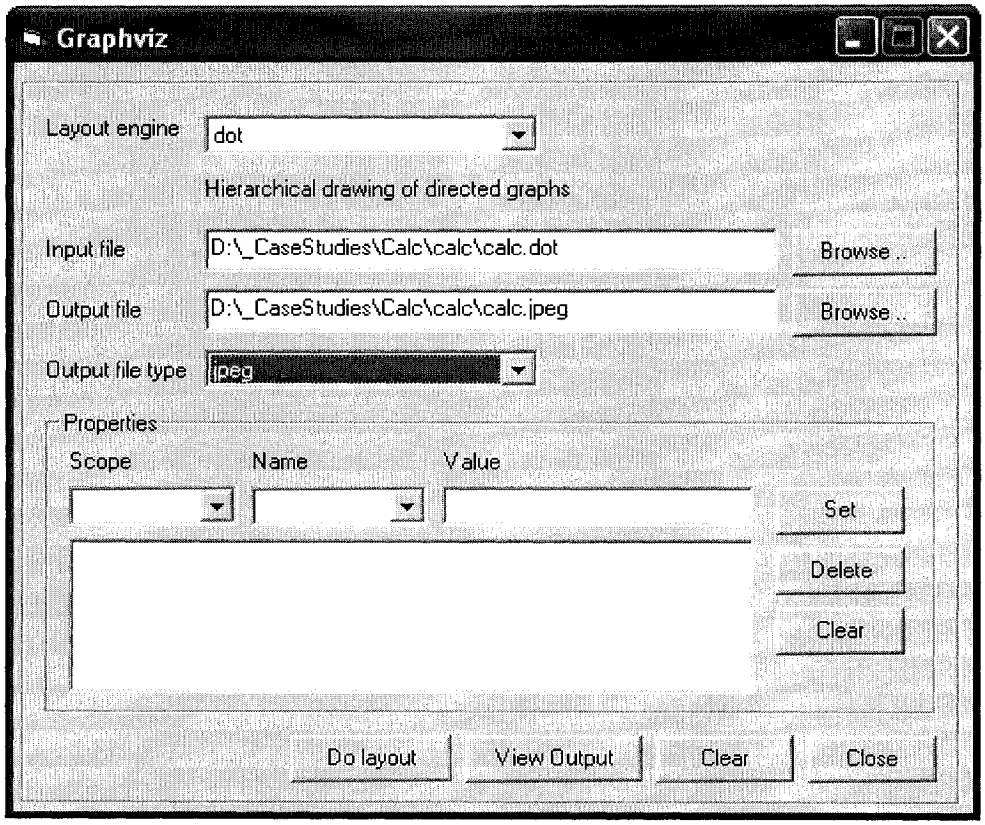


Figure 3.4.3.5.1: A screenshot of Graphviz program

In this thesis, the *dot* format is used. *Dot* is one of the layout programs in Graphviz software. The layout program is able to perform the hierarchical drawing of directed graphs. The input file format is of type *dot*. A sample concept lattice in the Graphviz format (.dot) is shown in Figure 3.4.3.5.2.

```
graph conceptlattice {
label = "concept lattice"
node9[label="\n11-24 11-23 11-22 11-19 11-21 11-20 11-29 11-27 11-26",fontsize=10,labelfloat=true];
node8[label="\n5-15 5-14",fontsize=10,labelfloat=true];
node7[label="plus\n11-33 11-32",fontsize=10,labelfloat=true];
node6[label="minus\n6-15 6-14 11-37 11-36",fontsize=10,labelfloat=true];
node5[label="multiply\n7-15 7-14 11-41 11-40",fontsize=10,labelfloat=true];
node4[label="\n8-15 8-14",fontsize=10,labelfloat=true];
node3[label="divide\n11-45 11-44",fontsize=10,labelfloat=true];
node2[label="sum\n9-21 9-19 9-18 11-49 9-16 9-15 11-48",fontsize=10,labelfloat=true];
node1[label="average\n10-21 11-53 11-52 10-19 10-18 10-16 10-15",fontsize=10,labelfloat=true];
node0[label="\n",fontsize=10,labelfloat=true];
node1 -- node0;
node2 -- node0;
node3 -- node0;
node4 -- node1;
node4 -- node3;
node5 -- node0;
node6 -- node0;
node7 -- node0;
node8 -- node1;
node8 -- node2;
node8 -- node7;
node9 -- node4;
node9 -- node5;
node9 -- node6;
node9 -- node8;
}
```

Figure 3.4.3.5.2: A sample concept lattice in Graphviz format (.dot)

The Graphviz program has a layout algorithm that re-organizes and converts the concept lattice into the *.dot* format. A sample visualization of an *execution dependency lattice* generated by the Graphviz program is shown in Figure 3.4.3.5.3.

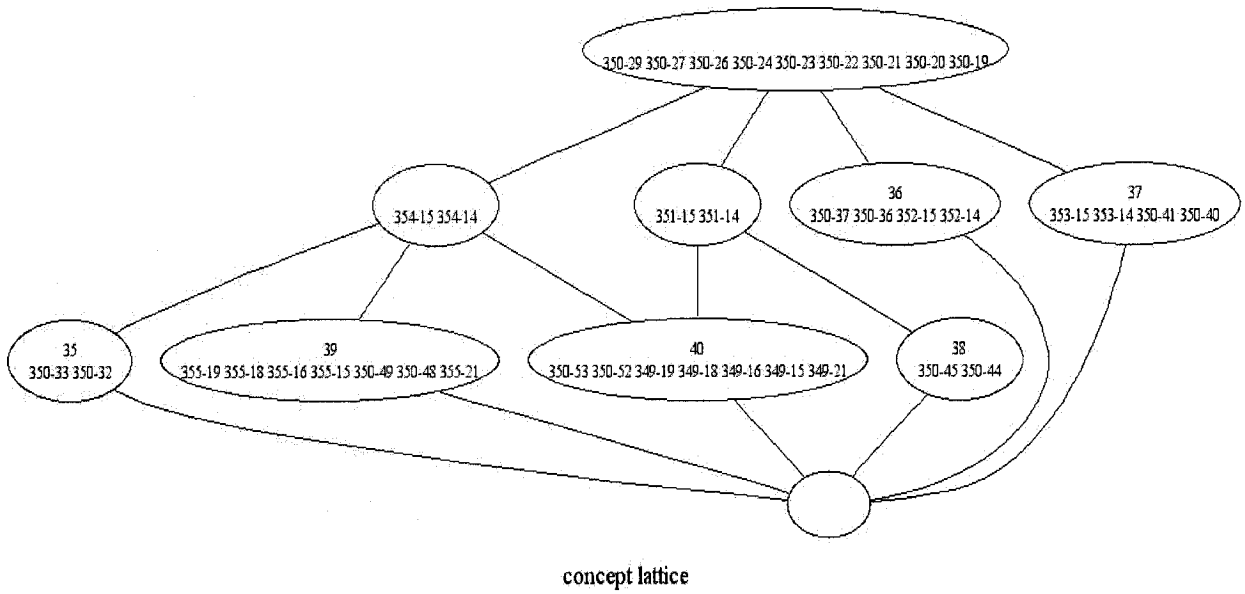


Figure 3.4.3.5.3: A sample *execution dependency lattice* layouts

In order to distinguish between objects and attributes, the nodes in the lattice are designed to contain two parts. The object or test case ID is always on the upper line and the attribute or list of statement IDs is placed on the lower line. The test case ID can be either a number or text as long as it is a unique identifier. For the improvement of the comprehensibility of the lattice, one can name a test case related to a particular program feature or tested program function by manually editing the FCA context file.

3.4.3.6 Locating the program change

In step P6, a user interaction is required to specify the node number where a particular change to the program is made. It should be noted that the program change has to be located within the *execution dependency lattice* for the test case selection to be performed. Every node in the lattice is identified by a node ID which one can look up in the dot

formatted lattice file (see Figure 3.4.3.5.2). Once the node is specified by a user, the test case selection process continues with step P7.

3.4.3.7 Conducting regression test case selection

Step P7 combines the program change specified by a user (P6) and the FCA results (P5) and determines the list of test cases to be retested after the program modification is performed. We implemented the algorithm for regression test case selection following the proposed method in section 3.4.2.

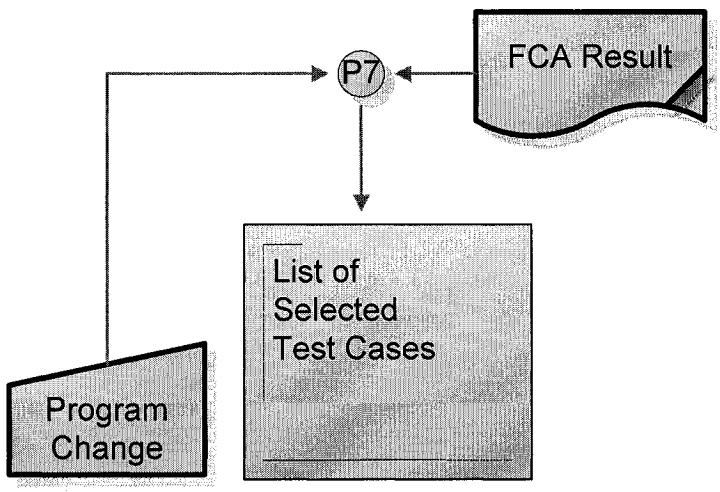


Figure 3.4.3.7.1: Step P7 – Conducting regression test case selection.

The algorithm is briefly described in the flowchart in Figure 3.4.3.7.2:

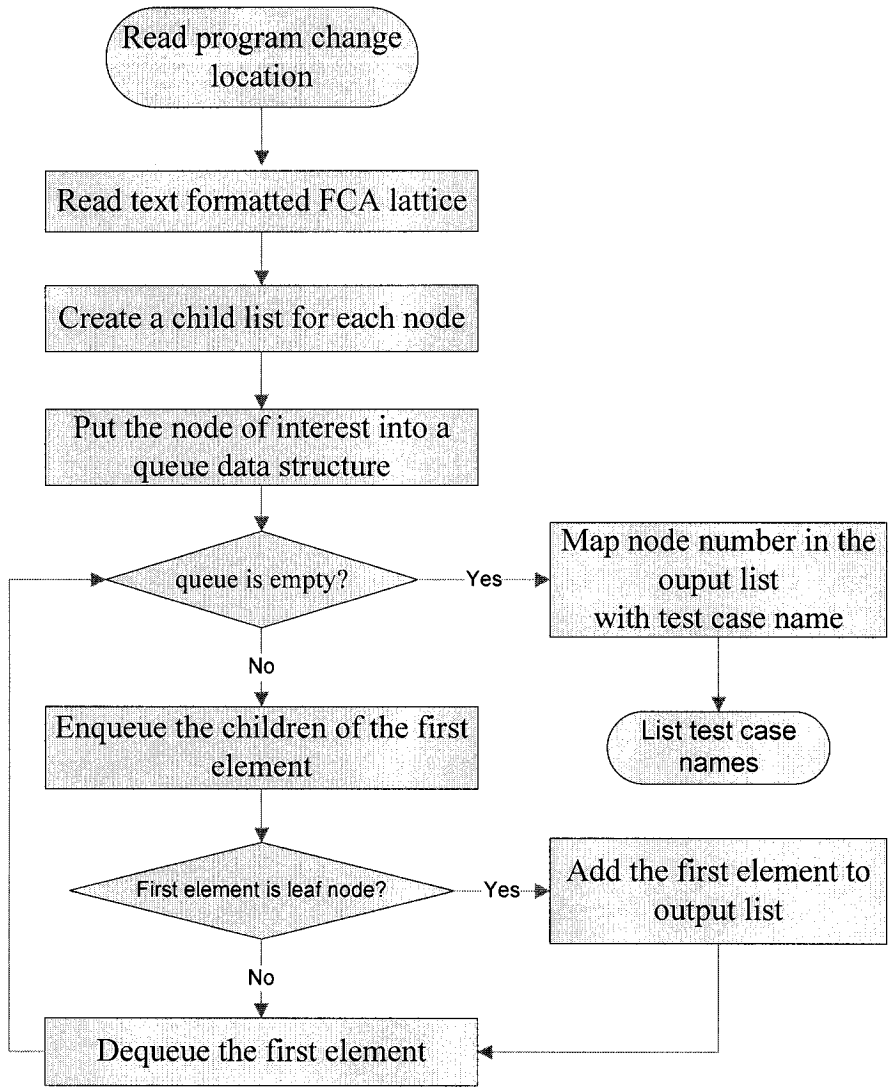


Figure 3.4.3.7.2: Flowchart – Regression test case selection

The algorithm is implemented as a command-line program. The source code for this algorithm can be found in Appendix D. The resulting output from the test case selection is shown in Figure 3.4.3.7.3. It lists the test case names or IDs that can be reused to test the modified program.

```
Please enter the number of node of interest >> 14

****** Building a child list for each node ************
node 0 : []
node 1 : [0]
node 2 : [0]
node 3 : [0]
node 4 : [0]
node 5 : [3, 4]
node 6 : [1, 5]
node 7 : [2, 6]
...
node 17 : [2, 10, 15, 16]
node 18 : [7, 14, 17]

****** Parse Dependency Tree & Collect leaves **********
Queue : []

Enqueue node of interest -> 14
Queue : [14]
Enqueue children of node 14
Queue : [14, 8, 11, 13]
Output => []
Dequeue head

Queue : [8, 11, 13]
Enqueue children of node 8
Queue : [8, 11, 13, 5]
Output => []
Dequeue head


Queue : [11, 13, 5]
Enqueue children of node 11
Queue : [11, 13, 5, 10]
Output => []
Dequeue head

...

Queue : [0]
Enqueue children of node 0
Queue : [0]
Output => [10, 9, 3, 4]
Dequeue head

****** List of test cases to be retested after node 14 has
changed

Test case #1 : MODE
Test case #2 : MEDIAN
Test case #3 : SD
Test case #4 : VAR
```

Figure 3.4.3.7.3: A sample run of the regression test case selection program.

3.4.3.8 Visualizing the test case selection result

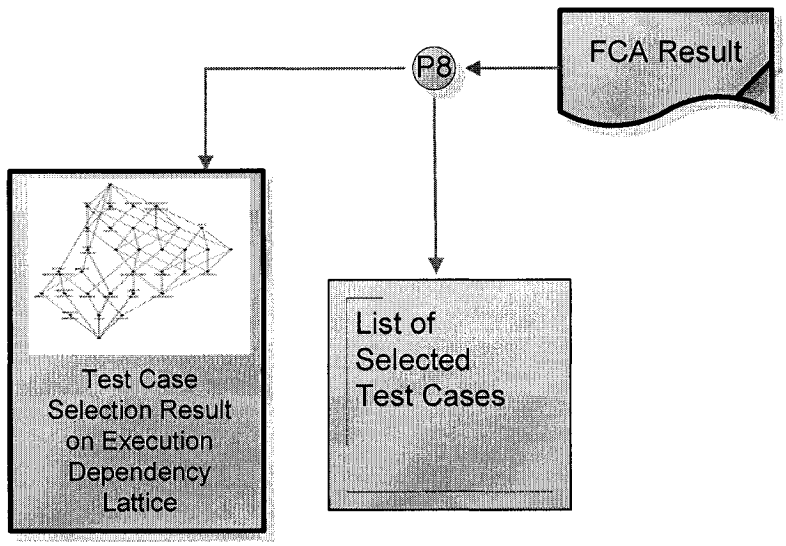


Figure 3.4.3.8.1: Step P8 – Visualizes test case selection results within the execution dependency lattice.

Step P8 visualizes the test case selection results from step P7 by highlighting the modified node and the test cases that are relevant (have to be retested) on the execution dependency lattice. The implementation is described in the flowchart in Figure 3.4.3.8.2.

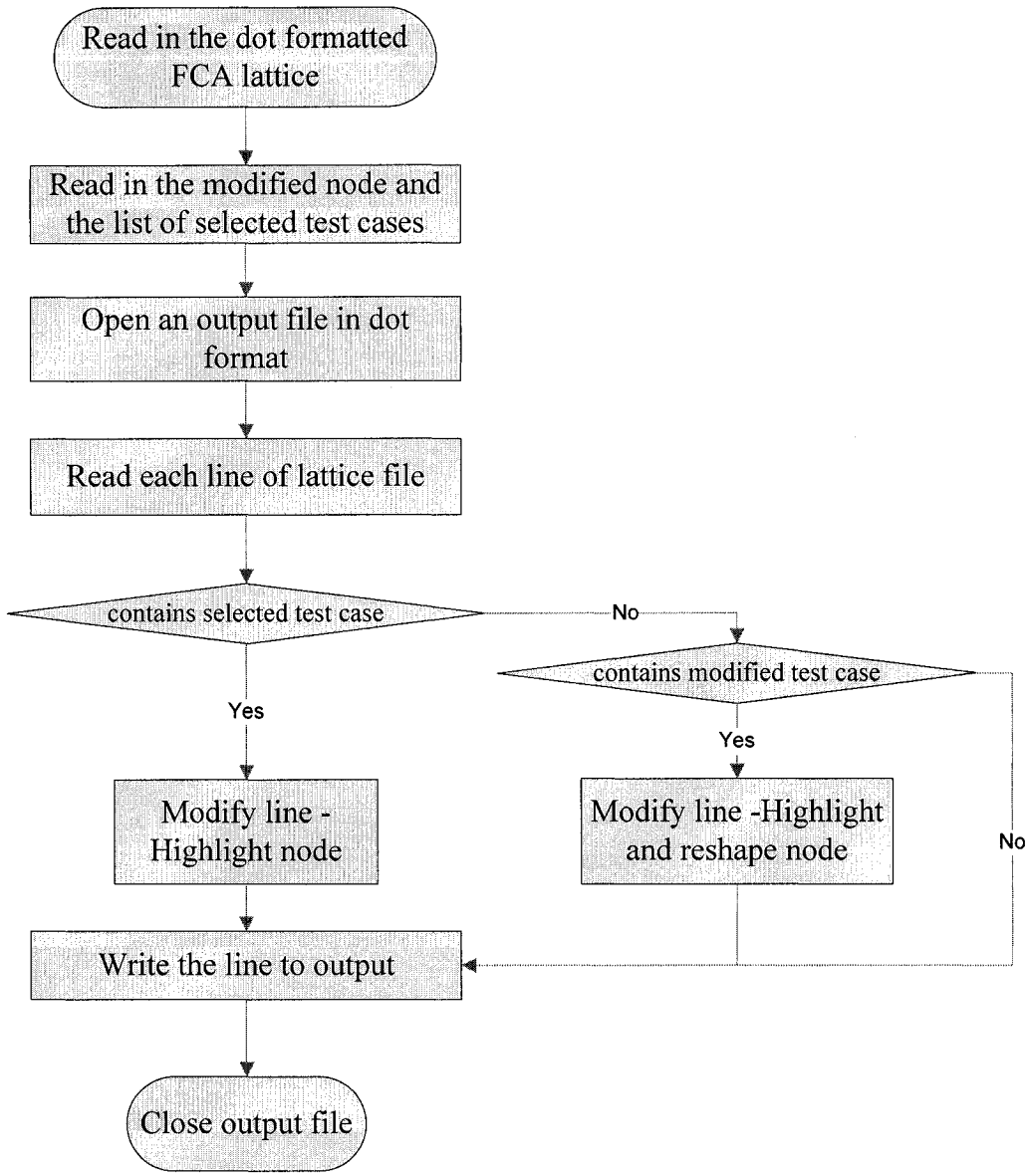


Figure 3.4.3.8.2: Flowchart – Visualizing test case selection result.

Once the output file is created, the Graphviz program will highlight the modified node as well as the selected test cases' nodes within the execution dependency lattice.

Figure 3.4.3.8.3 illustrates this step. For readability purposes, groups of program statements are replaced by an alias; attributes in each node are presented by names

instead of the list of program statement numbers. Further explanation will proceed in Chapter 4. The shaded diamond-shaped node represents the modified node. All other shaded nodes represent the test cases that need to be retested after at least one statement in the modified node has been changed.

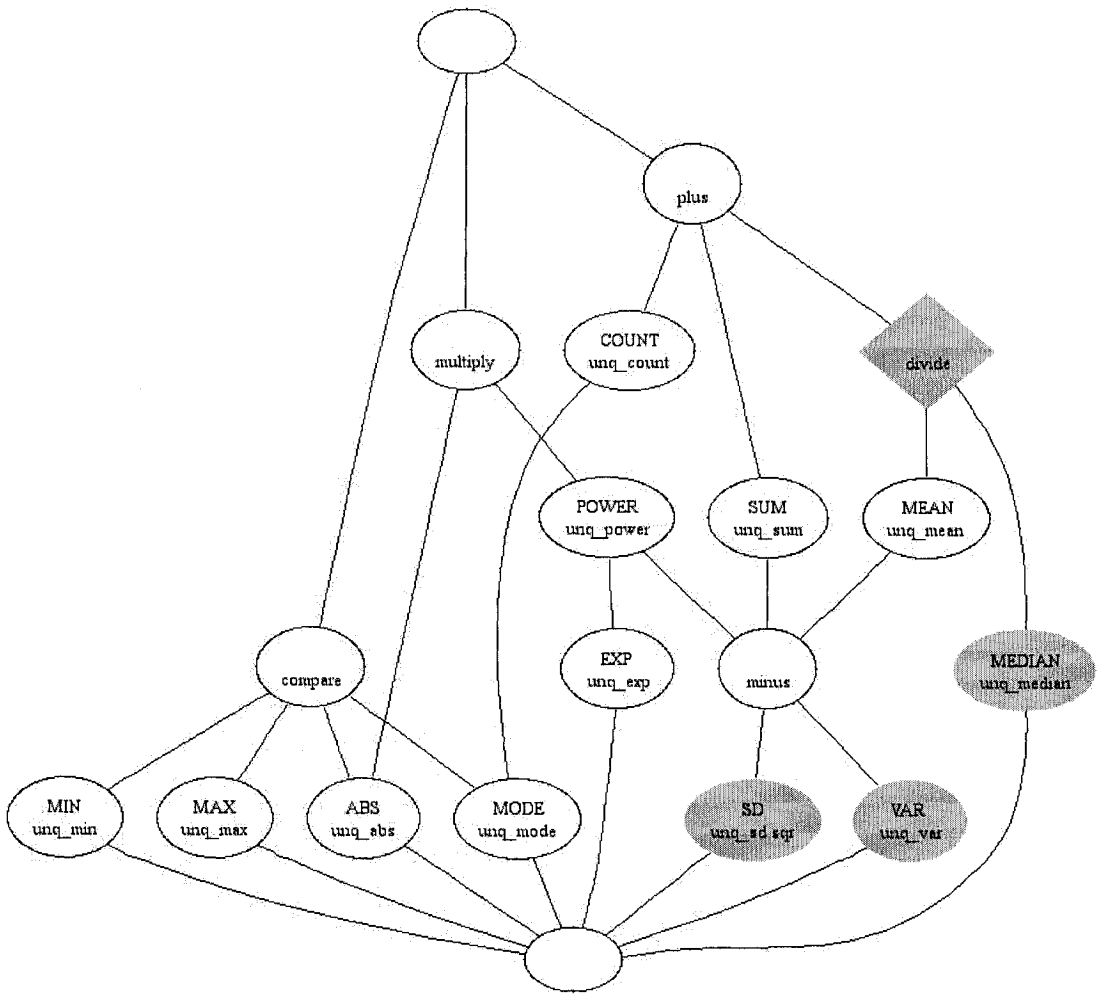


Figure 3.4.3.8.3: A sample *execution dependency lattice* after visualizing the test case selection results.

4. Case Studies

This chapter introduces two case studies that were conducted to demonstrate how the FCA-TCSS tool can be applied. The first case study demonstrates how FCA-TCSS derives an execution dependency lattice, while the second case study shows how the tool can perform regression test case selection. The problem solving steps (P0-P8) stated in chapter 3 will be re-applied in the case studies. Steps P0-P5 which generate the *execution dependency lattice* will be explained in the first case study, a simple java program called 'Calc' that simulates a simple calculator. Steps P6-P8 which focus on the test case selection will be illustrated in detail in the second case study, a larger java program called 'Stat' that simulates a statistics calculator.

4.1 Retrieve Execution Dependency Structure

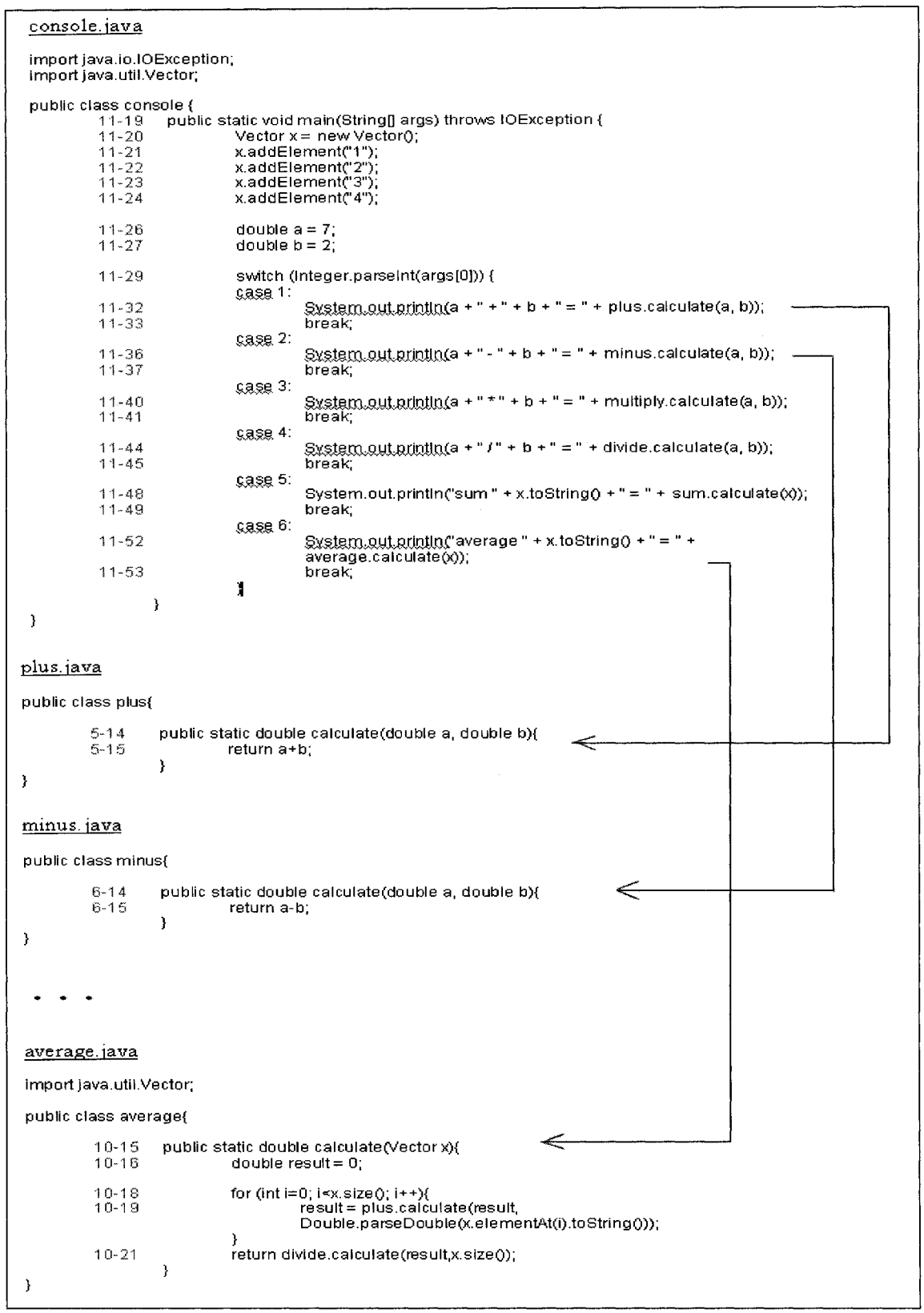
This case study is based on a small program to allow for statement-level representation of all the major processes involved in the execution dependency analysis. A simple java program, *Calc*, is introduced to illustrate the retrieval of *execution dependency lattice* by the FCA-TCSS tool. The complete source code for the *Calc* program can be found in Appendix B. The *Calc* program has 6 features:

1. Plus: finds the result of adding two given numbers.
2. Minus: finds the result of subtracting the second number from the first number.
3. Multiply: finds the result of multiplying two given numbers.
4. Divide: finds the result of dividing the first number by the second number.
5. Sum: finds the result of adding all numbers in the given list.

6. Average: finds the result of adding numbers in the given list and divides by the number of the list elements.

The program is composed of 7 classes. Six of them correspond to the 6 features listed above while the last class, *Console*, acts as a main menu and program starting point. Given the user's choice of program feature, the *Console* class calls another class to perform the selected function as illustrated in Figure 4.1.1. In this figure, each line of code is marked with a statement ID to facilitate execution trace reading in later parts.

Steps P0 to P5 of FCA-TCSS will generate an *execution dependency lattice* for this program. The *Calc* program is selected as the program of interest (P0) in FCA-TCSS. The test suite for *Calc* is inputted into the tool (P1). It is composed of six test cases - each test case executes one of *Calc*'s features. For example, the first test case executes *Plus* feature, the second test case executes *Minus* feature, and so on. Next, the *Calc* program is parsed into the instrumentation tool (P2). The instrumented program is then compiled by Java compiler. The instrumented source code can be found in Appendix C. After running the instrumented program with the test suite (P3), the program execution traces are generated. The instrumentation tool stores the execution traces in PostgreSQL DBMS.



console.java

```
import java.io.IOException;
import java.util.Vector;

public class console {
        11-19   public static void main(String[] args) throws IOException {
        11-20           Vector x = new Vector();
        11-21           x.addElement("1");
        11-22           x.addElement("2");
        11-23           x.addElement("3");
        11-24           x.addElement("4");

        11-26           double a = 7;
        11-27           double b = 2;

        11-29           switch (Integer.parseInt(args[0])) {
                        case 1:
        11-32                   System.out.println(a + " + " + b + " = " + plus.calculate(a, b));
        11-33                   break;
                        case 2:
        11-36                   System.out.println(a + " - " + b + " = " + minus.calculate(a, b));
        11-37                   break;
                        case 3:
        11-40                   System.out.println(a + " *" + b + " = " + multiply.calculate(a, b));
        11-41                   break;
                        case 4:
        11-44                   System.out.println(a + " /" + b + " = " + divide.calculate(a, b));
        11-45                   break;
                        case 5:
        11-48                   System.out.println("sum " + x.toString() + " = " + sum.calculate(x));
        11-49                   break;
                        case 6:
        11-52                   System.out.println("average " + x.toString() + " = " +
                                average.calculate(x));
        11-53                   break;
                        }
                }
}
```

plus.java

```
public class plus{

        5-14    public static double calculate(double a, double b){
        5-15            return a+b;
                }
}
```

minus.java

```
public class minus{

        6-14    public static double calculate(double a, double b){
        6-15            return a-b;
                }
}
```

* * *

average.java

```
import java.util.Vector;

public class average{

        10-15   public static double calculate(Vector x){
        10-16           double result = 0;

        10-18           for (int i=0; i<x.size(); i++){
        10-19                   result = plus.calculate(result,
                                Double.parseDouble(x.elementAt(i).toString()));
                        }
        10-21           return divide.calculate(result,x.size());
                }
}
```

Figure 4.1.1: Listing of Calc program

Querying the execution traces from the repository, FCA-TCSS creates a FCA context file by assembling all execution traces together with the test case names. Figure 4.1.2 shows the FCA context file for Calc program.

```
plus:11-19 11-20 11-21 11-22 11-23 11-24 11-26 11-27 11-29 11-32
5-14 5-15 11-33
minus:11-19 11-20 11-21 11-22 11-23 11-24 11-26 11-27 11-29 11-36
6-14 6-15 11-37
multiply:11-19 11-20 11-21 11-22 11-23 11-24 11-26 11-27 11-29 11
-40 7-14 7-15 11-41
divide:11-19 11-20 11-21 11-22 11-23 11-24 11-26 11-27 11-29 11-44
8-14 8-15 11-45
sum:11-19 11-20 11-21 11-22 11-23 11-24 11-26 11-27 11-29 11-48
9-15 9-16 9-18 9-19 5-14 5-15 9-19 5-14 5-15 9-19 5-14 5-15 9-19
5-14 5-15 9-21 11-49
average:11-19 11-20 11-21 11-22 11-23 11-24 11-26 11-27 11-29
11-52 10-15 10-16 10-18 10-19 5-14 5-15 10-19 5-14 5-15 10-19 5-14
5-15 10-19 5-14 5-15 10-21 8-14 8-15 11-53
```

Figure 4.1.2: FCA context for Calc program

FCA-TCSS performs the Formal Concept Analysis (P4). The system extracts commonalities and variability out of this context. The complete relation table and the concept list can be found in Appendix E. The context table is shown in Figure 4.1.2. The result from the analysis is then prepared in the required format (.dot) of the Graphviz program. Then, Graphviz program draws the *execution dependency lattice* from dot formatted file (P5). The picture of the *execution dependency lattice* for the *Calc* program is displayed in Figure 4.1.3.

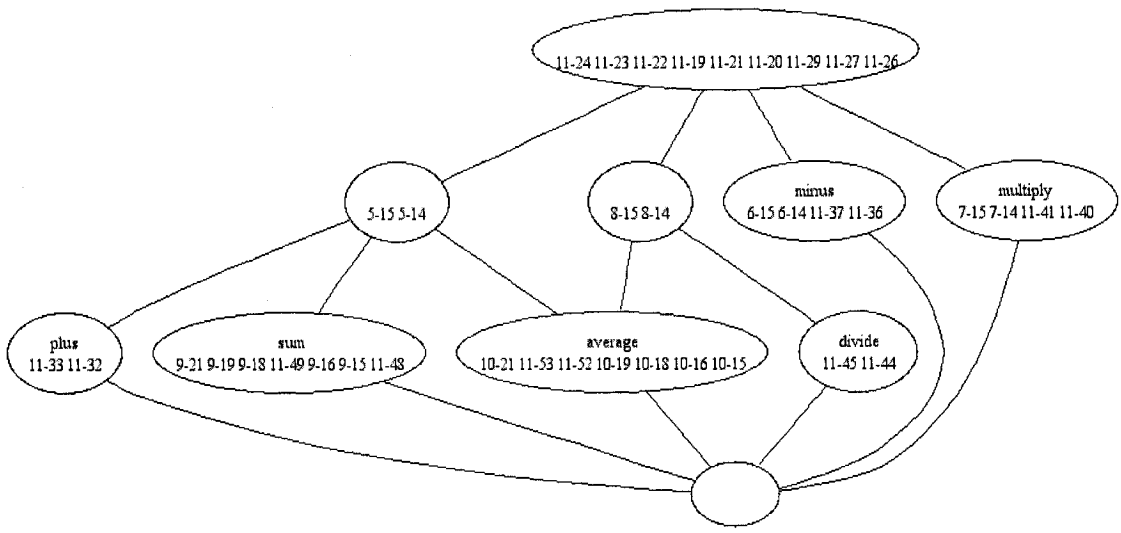


Figure 4.1.3: The *execution dependency lattice* of Calc program

As stated earlier, the FCA object corresponds to the test case and its attribute is the list of statement IDs (file# - statement#) executed by that particular test case. The *execution dependency lattice* provides some useful information for the test case selection. First, one will be able to obtain a non-redundant execution trace of a particular test case by collecting the execution trace elements (program statement IDs) starting from a node of interest to the root node. For example, in Figure 4.1.3, starting from *plus* node, if one collects all of the statement IDs along the path until the root node is reached, one will obtain a complete non-redundant execution trace which consists of: 11-33 11-32 5-15 5-14 11-24 11-23 11-22 11-19 ....11-26.

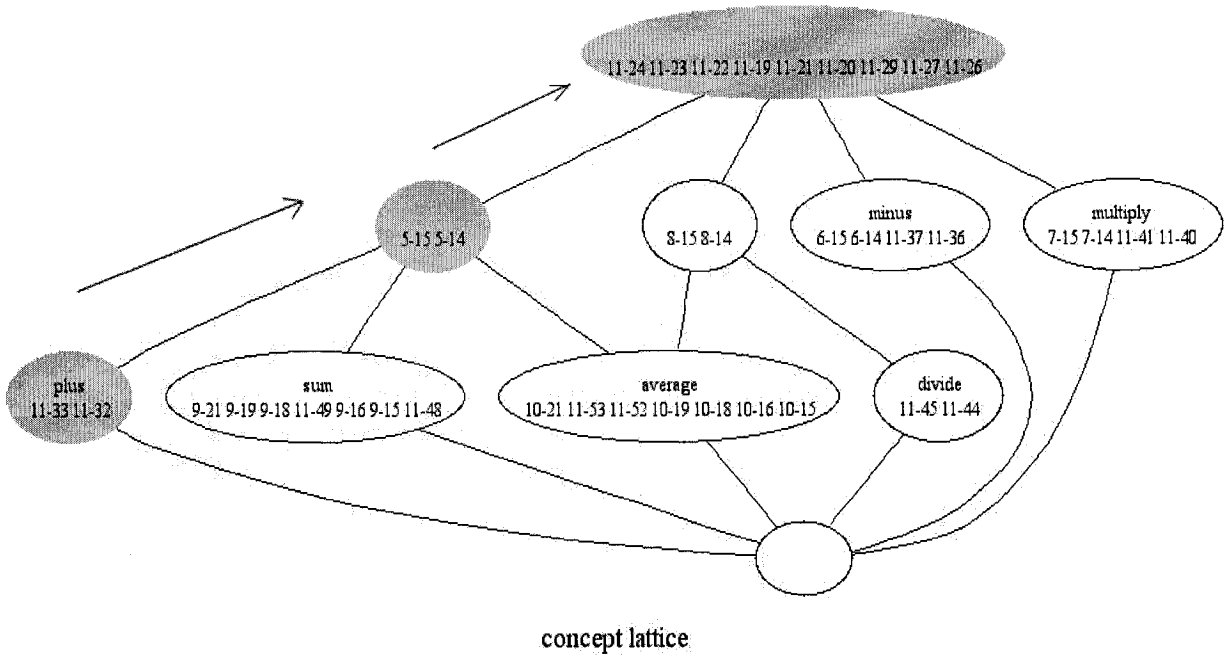


Figure 4.1.4: How to obtain an execution trace of plus test case from the lattice

Second, the *execution dependency lattice* can identify a set of execution trace elements that are shared by a particular group of test cases. If one traverses the *execution dependency lattice* upwards starting from a particular set of test cases, joining point implies the sharing relationships and therefore the execution dependency among test cases. Therefore, one can obtain the complete list of shared executing statements among test cases by collecting the statement IDs from the joining node to the root node. For example, plus, sum and average test case join at "5-15 5-14" node, so the shared execution trace is the combined list of statement IDs collected from "5-15 5-14" and traverse upwards. The complete list of shared program statements will be '5-15 5-14 11-24 11-23 11-19.... 11-26'.

Finally, the most important information obtained from the *execution dependency lattice* is that all the test cases that execute a particular program statement is a set of test cases (objects) presented at the node where the program statement (attribute) of interest appear. For example, to obtain all the test cases that execute program statement # 5-15, one simply reads a concept that contains statement ID 5-15 by passing down the attributes and passing up the objects. The concept will be ({*plus*, *sum*, *average*},{11-24 11-23 11-19.... 11-26, 5-15, 5-14}). Therefore, the list of test cases that execute program statement # 5-15 is *plus*, *sum*, and *average*.

4.2 Performing Regression Test Case Selection

The second case study demonstrates how the regression test case selection is performed. Since this case study puts the emphasis on the test case selection processes, it needs a readable but richer lattice than that of the first case study. A Java program, 'Stat', was designed to have more features and therefore yields a richer concept lattice. The Stat program provides the following statistics functions:

1. MIN: return the smallest number of the given list
2. MAX: return the largest number of the given list
3. MEAN: return the average of all numbers in the given list; the sum of all values divided by the total number of values
4. MODE: return the number that occurs most often.
5. MEDIAN: return the middle value in a set of numbers arranged in order of magnitude
6. COUNT: return the number of all members of the given list.

7. SUM: return the sum of all values from the given list.
8. VAR (Variance): return the average squared deviation about the mean
9. SD (Standard Derivation): return the square root of the variance
10. ABS (Absolute): return the absolute value of the given number

The FCA context file in this case study is designed to minimize the lattice node size so that each object does not have too many attributes. This is done because nodes that contain a large number of attributes will quickly create several challenges. The comprehension process will suffer due to the sheer size and information stored in a node, and the visual representation of the nodes in a lattice will be more difficult and challenging due to the required size of the node(s). For simplification reasons, we introduced an alias for the nodes. The alias approach allows us to compact the size of the node and provides some meaningful interpretation of the node content which allows for an improved readability of the *execution dependency lattice*. The context is still in the same format 'object: list of attributes' as defined earlier. However, the group of executed statement IDs is an attribute in this case and it is replaced by an alias.

In this case study, the program has 10 features and each feature is tested once; therefore, the FCA context contains 10 execution traces. Each group of statement IDs (attributes) was named as shown in the context in Figure 4.2.1.

```
MIN:unq_min compare
MAX:unq_max compare
MEAN:unq_mean plus divide
MODE:unq_mode compare unq_count plus
MEDIAN:unq_median plus divide
COUNT:unq_count plus
SUM:unq_sum plus
VAR:unq_var unq_sum plus minus power divide unq_mean
SD:unq_sd unq_sum minus unq_mean plus divide power sqr
ABS:unq_abs compare multiply
```

Figure 4.2.1: FCA context for Stat program

The execution traces were grouped based on their cohesion and nature of program execution. In a program that contains functions, procedures, or libraries, there typically exist some pieces of execution traces that are always executed together. In this case study, such pieces of execution traces are named after their functionality e.g. *compare*, *plus*, and *divide*. For the remaining execution trace elements that cannot be grouped, they will be named with prefix 'unq'.

For example,

MIN: unq_min compare

MAX:unq_max compare

If it were formatted as in the first case study, it would look similar to the following line.

MIN: 5-34 2-21 2-22 2-23 2-44 2-45 2-46 3-35

MAX: 8-44 2-21 2-22 2-23 2-44 2-45 2-46 4-52

Both the MIN and MAX object have the *compare* part in their attribute list. This corresponds to a situation where both the MIN and the MAX test case share some execution trace elements. In this particular example, the *compare* alias would correspond to '2-21 2-22 2-23 2-44 2-45 2-46', *unq_min* represents '5-34 3-35', and *unq_max* represents '8-44 4-52'.

In this case study, steps P0-P5 are executed in the same fashion as in the first case study. The product of step P5 is the concept lattice in dot format shown in Figure 4.2.2.

```
graph conceptlattice {
label = "concept lattice"
node15[label="\n",fontsize=10,labelfloat=true];
node14[label="\ncompare",fontsize=10,labelfloat=true];
node13[label="MIN\nunq_min",fontsize=10,labelfloat=true];
node12[label="MAX\nunq_max",fontsize=10,labelfloat=true];
node11[label="\nplus",fontsize=10,labelfloat=true];
node10[label="\ndivide",fontsize=10,labelfloat=true];
node9[label="MEAN\nunq_mean",fontsize=10,labelfloat=true];
node8[label="COUNT\nunq_count",fontsize=10,labelfloat=true];
node7[label="MODE\nunq_mode",fontsize=10,labelfloat=true];
node6[label="MEDIAN\nunq_median",fontsize=10,labelfloat=true];
node5[label="SUM\nunq_sum",fontsize=10,labelfloat=true];
node4[label="\npower minus",fontsize=10,labelfloat=true];
node3[label="VAR\nunq_var",fontsize=10,labelfloat=true];
node2[label="SD\nsqr unq_sd",fontsize=10,labelfloat=true];
node1[label="ABS\nmultiply unq_abs",fontsize=10,labelfloat=true];
node0[label="\n",fontsize=10,labelfloat=true];
node1 -- node0;
node2 -- node0;
node3 -- node0;
node4 -- node2;
node4 -- node3;
node5 -- node4;
node6 -- node0;
node7 -- node0;
node8 -- node7;
node9 -- node4;
node10 -- node6;
node10 -- node9;
node11 -- node5;
node11 -- node8;
node11 -- node10;
node12 -- node0;
node13 -- node0;
node14 -- node1;
node14 -- node7;
node14 -- node12;
node14 -- node13;
node15 -- node11;
node15 -- node14;
}
```

Figure 4.2.2: FCA result of Stat program in dot format

The Graphviz program is used to visualize the FCA result of the *Stat* program and creates the resulting *execution dependency lattice* shown in Figure 4.2.3.

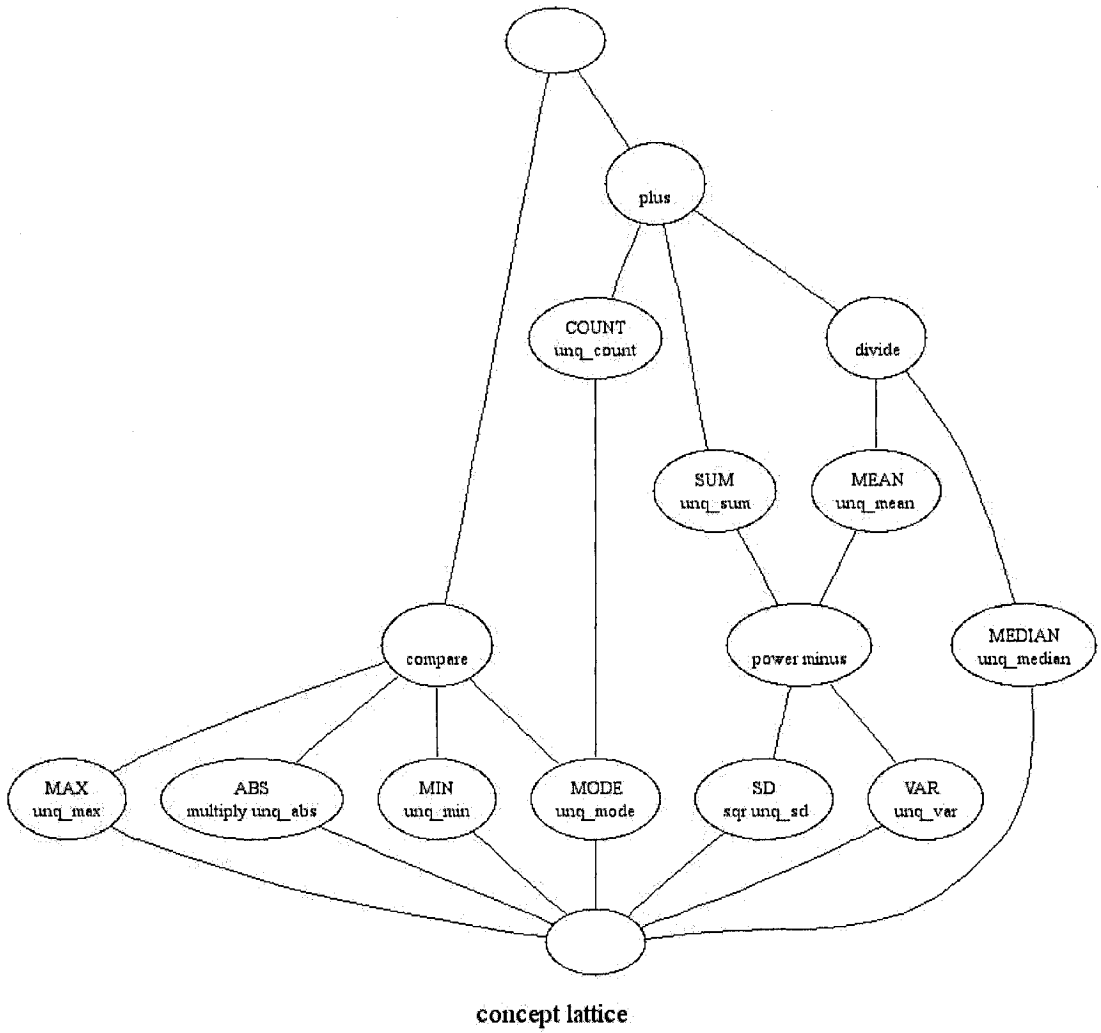


Figure 4.2.3: *Execution dependency lattice* of Stat program

As demonstrated previously in the first case study, the execution dependency lattice can be used to identify the dependency among pieces of execution traces. When the *MEDIAN* test case is executed, the source code artefacts represented by the *unq_median*, *divide* and *plus* aliases are tested. Similarly, when the *SD* test case is executed, all upward reachable source code artefacts including *sqr*, *unq_sd*, *power*, *minus*, *unq_sum*, *unq_mean*, *divide*, and *plus* are executed. This creates the basic scenario for the regression test case selection which will be discussed in the remaining part of this section.

Assuming the given program change is located at the divide node (node #10 in Figure 4.2.2), the FCA-TCSS performs the test case selection algorithm presented in section 3.4.3.7. The following discussion will explain how this algorithm computes the regression test case selection.

From the given scenario, the program modification is starting at the *divide* node, which means that at least one statement in the *divide* group of executed statements has been modified. Considering the FCA context for the *Stat* program, one can see that *divide* exists in the attribute list of MEAN, MEDIAN, VAR, and SD. Figure 4.2.4 highlights all the test cases that execute the program statements represented by *divide*.

```
MIN:unq_min compare
MAX:unq_max compare
MEAN:unq_mean plus divide
MODE:unq_mode compare unq_count plus
MEDIAN:unq_median plus divide
COUNT:unq_count plus
SUM:unq_sum plus
VAR:unq_var unq_sum plus minus power divide unq_mean
SD:unq_sd unq_sum minus unq_mean plus divide power sqr
ABS:unq_abs compare multiply
```

Figure 4.2.4: FCA context for Stat program

Based on our assumption, MEAN, MEDIAN, VAR and SD are included in the candidate list. The lattice shows that MEAN, MEDIAN, VAR, and SD can be obtained by performing an in-depth reachable analysis. However, our proposed algorithm excludes non-leaf candidate nodes. This is due to the fact that each and every attribute of the non-leaf candidate nodes is already included in the combined attribute list of the leaf candidate nodes. This assumption also holds in this case study since every attribute of the

MEAN test case is already included in the combined attribute list of MEDIAN, VAR, and SD. Consequently, the final resulting test case list is MEDIAN, VAR, and SD.

This concludes that the presented explanation and case study supports the validity of our test case selection method and algorithm. Finally, the fact that the algorithm enables the regression test case selection fulfills the presented research goals.

Figure 4.2.5 displays the screenshot after executing the test case selection algorithm. The results listed at the end of this analysis are the test cases to be retested.

```
Please enter the number of node of interest >> 10

********** Building a child list for each node ***********
node 0 : []
node 1 : [0]
node 2 : [0]
node 3 : [0]
node 4 : [2, 3]
...
node 14 : [1, 7, 12, 13]
node 15 : [11, 14]

*************** Parse Dependency Tree & Collect leaves ******************
Queue : []

Enqueue node of interest -> 10
Queue : [10]
Enqueue children of node 10
Queue : [10, 6, 9]
Output => []
Dequeue head

Queue : [6, 9]
Enqueue children of node 6
Queue : [6, 9, 0]
Output => [6]
Dequeue head

...

Queue : [0, 0]
Enqueue children of node 0
Queue : [0, 0]
Output => [6, 2, 3]
Dequeue head

Queue : [0]
Enqueue children of node 0
Queue : [0]
Output => [6, 2, 3]
Dequeue head

****** List of test cases to be retested after node 10 has changed *******

Test case #1 : MEDIAN
Test case #2 : SD
Test case #3 : VAR
```

Figure 4.2.5: Screenshot from the regression test case selection module

A complete screenshot can be found in Appendix F and the source code of test case selection algorithm is shown in Appendix D.

In the last step (P8), the FCA-TCSS visualizes the results from the test case selection step by highlighting in the *execution dependency lattice* the modified nodes and the relevant reusable test cases. The *execution dependency lattice* in the *.dot* format is modified by adding some extra properties to distinguish these nodes affected by the change. The modification in the dot file is shown as bold text in Figure 4.2.6. The text "shape=diamond,style=filled,color=grey", is added into the property of node 10 *(divide)* which is the modified node to reshape and highlight *divide* node. Then, the text "style=filled,color=grey" is added into the property of node 2, 3 and 6 to highlight *SD, VAR,* and *MEDIAN* nodes.

```
graph conceptlattice {
label = "concept lattice"
node15[label="\n",fontsize=10,labelfloat=true];
node14[label="\ncompare",fontsize=10,labelfloat=true];
node13[label="MIN\nunq_min",fontsize=10,labelfloat=true];
node12[label="MAX\nunq_max",fontsize=10,labelfloat=true];
node11[label="\nplus",fontsize=10,labelfloat=true];
node10[label="\ndivide",fontsize=10,labelfloat=true,shape=diamond,style=filled,color=grey];
node9[label="MEAN\nunq_mean",fontsize=10,labelfloat=true];
node8[label="COUNT\nunq_count",fontsize=10,labelfloat=true];
node7[label="MODE\nunq_mode",fontsize=10,labelfloat=true];
node6[label="MEDIAN\nunq_median",fontsize=10,labelfloat=true,style=filled,color=grey];
node5[label="SUM\nunq_sum",fontsize=10,labelfloat=true];
node4[label="\npower minus",fontsize=10,labelfloat=true];
node3[label="VAR\nunq_var",fontsize=10,labelfloat=true,style=filled,color=grey];
node2[label="SD\nsqr unq_sd",fontsize=10,labelfloat=true,style=filled,color=grey];
node1[label="ABS\nmultiply unq_abs",fontsize=10,labelfloat=true];
node0[label="\n",fontsize=10,labelfloat=true];
node1 -- node0;
node2 -- node0;
node3 -- node0;
node4 -- node2;
node4 -- node3;
node5 -- node4;
node6 -- node0;
node7 -- node0;
node8 -- node7;
node9 -- node4;
node10 -- node6;
node10 -- node9;
node11 -- node5;
node11 -- node8;
node11 -- node10;
node12 -- node0;
node13 -- node0;
node14 -- node1;
node14 -- node7;
node14 -- node12;
node14 -- node13;
node15 -- node11;
node15 -- node14;
}
```

Figure 4.2.6: Dot formatted FCA result of Stat program that visualize the modified node and its relevant test cases' nodes.

At the end of the final step (P8), Graphviz visualizes the *execution dependency lattice* that shows the result of regression test case selection as illustrated in Figure 4.2.7.

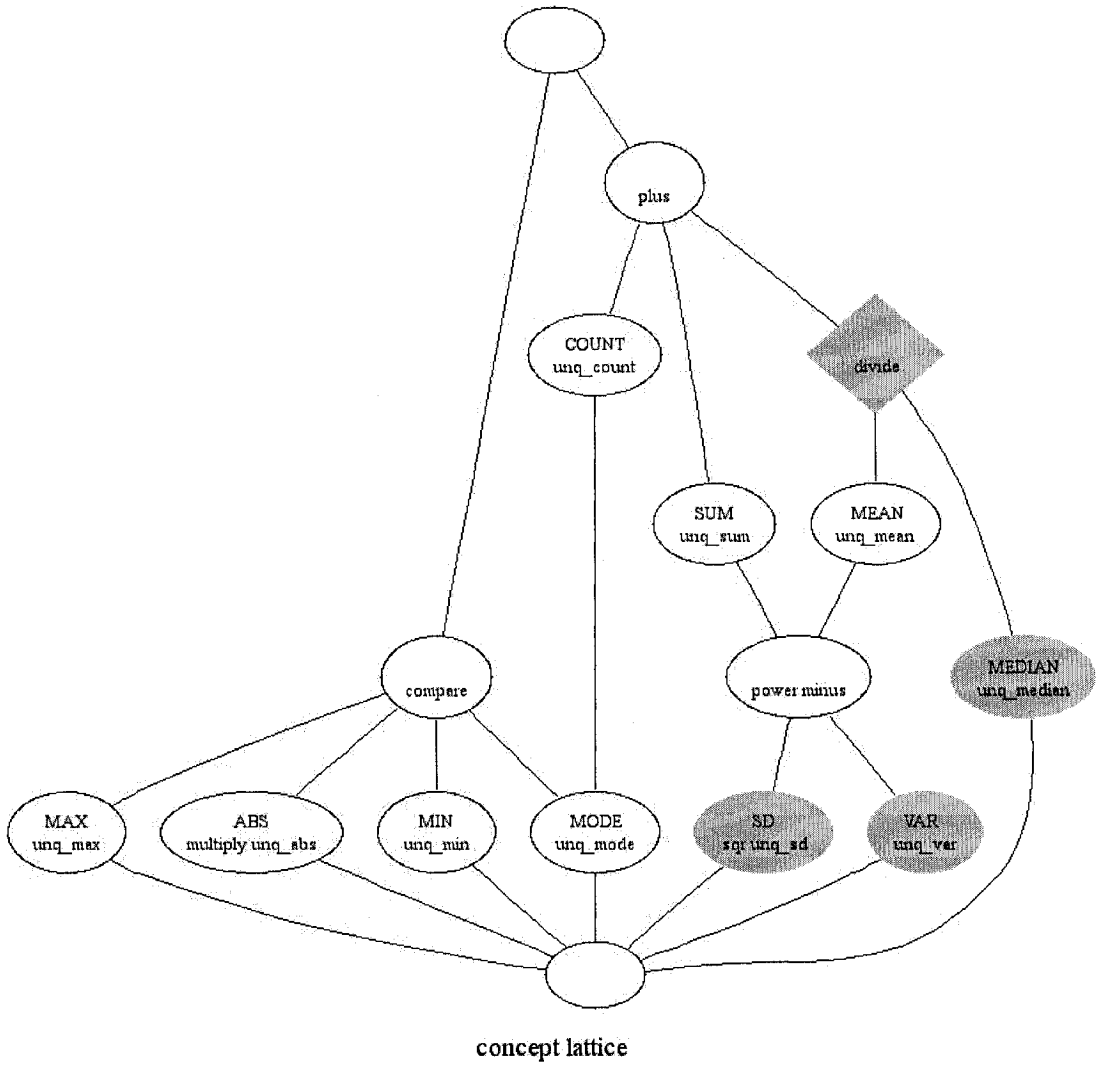


Figure 4.2.7: Highlighting relevant test cases' nodes in the *Stat* program.

4.3 Discussion

Based on the results from our initial experiments and observations, our test case selection technique can successfully reduce the number of test cases and therefore the associated testing cost. However, the completeness of an execution dependency concept lattice, which depends on the coverage of the test suite(s) applied, directly affects the accuracy of

the results from FCA-TCSS. If the program is run with too few test cases (assuming low coverage), the concept lattice created by the FCA algorithm will be small and imprecise. Imprecise in this context refers to situations where the FCA is based on too few data points (execution traces and statement coverage) to provide sufficient concepts and the execution dependency information. On the other hand, if the program is run by various types of input and executes more parts of program, the coverage will typically be larger. Therefore, the lattice will provide a more complete and detailed representation of the execution dependencies.

One can obtain the boundary of the number of concepts (the number of lattice nodes) by defining the extreme cases for the FCA inputs. It is possible that a program yields only one concept. Such a program would follow the same execution path for every test case run. A typical example is the simple structural program that has no conditional/branching statements (i.e. if-else, case, goto, etc.). This could also be caused by the quality and/or quantity of the test case. If there is only one test case available, this will yield only one concept. The other extreme is that the program may result in a number of concepts that is close to the number of program statements in the program. As a result, the boundary of the number of concepts can be stated as $1 \leq$ *number of concepts* $\leq$ *number of program statements*.

In this research, we assume that the test cases are based on a statement test coverage criteria, as defined by Rapps and Weyuker in [RAP82, RAP85]. This will guarantee that the *execution dependency lattice* yielded should give sufficient information for predictive

regression test case selection. Our hypotheses then are based on the assumption that the *execution dependency lattice* provides sufficient information.

The advantages of our FCA-based approach over other test case selection techniques are various. Firstly, our approach is computationally inexpensive compared with other dynamic dependency-based test selection techniques (i.e. dynamic slicing). The presented test case selection technique becomes a low cost alternative to other test selection techniques, due to the limited analysis required as part of FCA. There is no need for static analysis with respect to parsing or other fact extraction techniques. Our strategy also utilizes one of FCA's main characteristic, that is, the resulting lattice always presents non-redundant information. This is because before extracting commonalities and variability out of the context table, FCA algorithm removes any kind of redundancy from both object and attribute list. Thus, the *execution dependency lattice* obtained by means of FCA contains non-redundant and therefore compact information. Secondly, the presented approach is independent of the programming language or environment for which the analysis is performed; the inputs to the FCA analysis are execution traces, which can be generated in most existing programming environments. Thirdly, our presentation of the results from the analysis is intuitive. The fact that the FCA results can be visualized in a hierarchical structure facilitates the understanding of execution dependencies and improves the interpretation of these results. Fourthly, our approach can provide good accuracy in test case selection where sufficient quality inputs are provided. The quality and quantity aspect corresponds to test cases traversing all the program statements. In the presented research, we performed the analysis at the program statement level which is the level of granularity that usually gives high accuracy of the analysis.

The information complexity can be further reduced by raising the level of abstraction with respect to the granularity of the analysis. Finally, since our approach is based on dynamic analysis, it overcomes several challenges typically faced by static analysis techniques, e.g. issues related to inclusion of run-time libraries, pointers, polymorphism, and so forth. These issues are resolved by basing the analysis on the actual execution traces (dynamic information) which are collected during run-time.

The previous discussion has introduced several advantages of our approach; however, it must be noted that the presented approach also has its limitations. The issue of scalability can be viewed as one of the major limitations. The approach conducts the analysis at the program statement level and the size of the *execution dependency lattice* is bound to the number of program statements in the source code to be analyzed. The *execution dependency lattice* can become unreadable when applying the approach to a large program. Relying on execution trace also binds the accuracy level of the analysis to the coverage and therefore the quality of the existing test suite. As stated before, the presented approach can result in very accurate results but the level of accuracy is directly dependent on the coverage created by the existing test suite. In regression test case selection, the technique attempts to select the reusable test case out of the existing test suite. However, one typically has no control of the quality (coverage) achieved by these test cases. Given a test suite with poor coverage, the system may not reflect all possible executions that involve the modified component and therefore the analysis will not consider all possible statements. Finally, the proposed approach will have a limitation with respect to its applicability for certain types of change. Our approach is applicable to

modification and deletion types of change but not addition (i.e. the integration of a new subsystem). However, this limitation is similar to the limitations found in other regression test case selection techniques.

5. Conclusions and Future Work

In this research, we proposed a new technique for identifying regression test case selection by means of Formal Concept Analysis. A literature review of background knowledge relevant to the thesis was provided, with a focus primarily on FCA, its current applications, impact analysis, and regression testing. The goals of this research were described and the hypotheses and assumptions were defined according to these goals. A methodology for identifying reusable test cases using FCA was presented. An automated tool, the FCA Test Case Selection System (FCA-TCSS), was implemented as part of the problem solving approach. The FCA-TCSS consists of several functionalities including collecting execution traces, implementing an FCA algorithm to extract execution dependency information, performing regression test case selection, and visualizing the result of each analysis. The FCA algorithm written originally by Lindig was re-implemented in Java to increase its portability and the portability of the FCA-TCSS tool. As part of this research, several modules were implemented and external tools integrated (instrumentation tool and a graph drawing software) to support the identification of regression test cases. We have demonstrated each step of the proposed methodology and illustrated its applicability with the help of two initial case studies. Finally, we provided a discussion on the advantages and the limitations of our approach.

Future work could include applying the results of our analysis to be integrated as measure and estimation with respect to the reusability of existing tests as part of regression testing. Scalability issues need to be further addressed. One possible solution to reduce the size of the *execution dependency lattice* would be to replace a group of executed statements with

an alias. This is actually similar to what we explained in *Stat* case study. However, the trade-offs of this solution are the lower level of granularity and the overhead cost for source code grouping and alias reference. Another feasible solution to resolve the scalability issue is to omit the unrelated parts of the lattice. In other words, it will increase the readability if one visualizes only the area of interest or the area that contains the modified part and selected test cases. As a matter of fact, the limitation of our approach in terms of scalability refers only to the FCA lattice size. Our approach can actually provide very good scalability for the lattice in text format (before visualization). Therefore, we suggest that any extension of our work omit unnecessary visualization and utilize the analysis result in text format instead of the lattice format.

Furthermore, it would be of interest to define the method to prioritize the test cases selected by our technique in order to further reduce the number of test cases. The prioritizing work could be accomplished based on impact analysis or test coverage criteria.

Finally, an additional evaluation should be carried out to compare the effectiveness of our approach with respect to its performance and accuracy with other selective regression techniques.

6. References

[ANA99] *Anaconda*, http://www.mathematik.tu-darmstadt.de/ags/ag1/Software/Anaconda/Welcome_en.html.

[ATT00] AT&T Labs-Research, *Graphviz*, http://www.graphviz.org.

[AUE04] Auer, S.; *Formal Concept Calculator*, http://www.advis.de/soeren/fca/index.php.

[BAU04] University of Stuttgart, *Bauhaus Project*, http://www.iste.uni-stuttgart.de/ps/bauhaus/.

[BAT93] Bates, S.; Horwitz, S.; *Incremental Program Testing using Program Dependence Graphs*, In the Conference Record of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, 1993, p.384–396.

[BEC01] Becker, P.; Eklund, P.W.; *Prospects for Formal Concept Analysis in Document Retrieval,* Australian Document Computing Symposium (ADCS01), University of Sydney, Basser Department of Computer Science, 2001, p.5-9.

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

[BRA99] Brandenburg, F.J., *Graphlet*, Universität Passau, http://www.infosun.fmi.uni-passau.de/Graphlet.

[CHA04] Charland, P.; *Enhancing Traditional Behavioral Testing through Program Slicing*, M.Comp.Sc. thesis, Concordia University (Canada), Sep 2004.

[CHE94] Chen, Y.F.; Rosenblum, D.S.; Vo., K.P.; *TestTube: A System for Selective Regression Testing*, In Proceedings of the 16th International Conference on Software Engineering, May 1994, p. 211-220.

[COL011] Cole, R.J; *The Management and Visualization of Document Collections using FCA*, School of Information Technology, Griffith University, 2001.

[COL012] Cole, R.J.;Eklund, P.W.; *Structured Ontology and IR for Email Search and Discovery*, Australian Document Computing Symposium (ADCS01), University of Sydney, Basser Department of Computer Science, 2001, p.5-9.

[COL013] Cole, R.J.; Eklund, P.W.; *Browsing Semi-Structured Web Texts using Formal Concept Analysis,* In Proceedings of the 9th International

Conference on Conceptual Structures, LNAI 2120, Springer Verlag, 2001, p.319-332.

[CON03] Concept Explorer Project, http://sourceforge.net/projects/conexp.

[DOS99] DOS Programs of the Darmstadt Research Group on Formal Concept Analysis, http://www.mathematik.tu-darmstadt.de/ags/ag1/Software/DOS-Programme/Welcome_en.html.

[DEU99] van Deursen, A.; Kuipers, T.; *Identifying Objects Using Cluster and Concept Analysis*, In Proceedings of the 1999 International Conference on, 16-22 May 1999, p. 246 –255.

[EIS011] Eisenbarth, T.; Koschke, R.; Simon, D.; *Feature-Driven Program Understanding Using Concept Analysis of Execution Traces*, In Proceedings of the International Workshop on Program Comprehension, 2001.

[EIS012] Eisenbarth, T.; Koschke, R.; Simon, D.; *Aiding Program Comprehension by Static and Dynamic Feature Analysis*, In Proceedings of the International Conference on Software Maintenance, IEEE Computer Society Press, Nov 2001.

[EIS013] Eisenbarth, T.; Koschke, R.; Simon, D.; *Derivation of Feature Component Maps by means of Concept Analysis*, Software Maintenance and Reengineering, 2001. Fifth European Conference, Mar 2001.

[EMA03] Email Analysis Pty Ltd, *Mail-Sleuth,* http://www.mail-sleuth.com.

[FIS81] Fischer, K.F.; Raji, F.; Chruscicki, A.; *A Methodology for Retesting Modified Software*, In Proceedings of the Nat'l. Tele. Conference B-6-3, Nov 1981, p. 1-6.

[GOO75] Goodenough, J.B.; Gerhart, S.L.; *Toward a Theory of Test Data Selection.* In Proceedings of the International Conference on Reliable Software, Los Angeles, California, 1975, p. 493–510.

[GNU] GNU Project, http://www.gnu.org.

[GRA01] Graves, T.L. et al. *An Empirical Study of Regression Test Selection Techniques*, ACM Transactions on Software Engineering and Methodology (TOSEM) Volume 10, Issue 2, Apr 2001, p. 184 – 208.

[HAR88] Harrold, M.; Sofia, M.; *An Incremental Approach to Unit Testing During Maintenance*, In Proceedings of Conference on Software Maintenance, 1988, p. 362-367.

[HAR90] Hartmann, J.; Robson, D.; *Techniques for Selective Revalidation.* IEEE Software, Jan 1990, p.31-38.

[IEEE] *IEEE Standard Glossary of Software Engineering Terminology*, report IEEE Std 829.1998 (Revision of IEEE Std 829-1983).

[JAL04] JaLaBA - Online Java Lattice Building Application for Concept Lattices, http://maarten.janssenweb.net/jalaba/JaLaBA.pl.

[JOS94] Jos van Eijndhoven, Graphplace program, ftp://ftp.dcs.warwick.ac.uk/people/Martyn.Amos/packages/graphplace.

[KAR04] University Karlsruhe, Collection of Visualization Tools, http://i11www.ira.uka.de/cosin/tools/index.php.

[KEA88] Keables, J.; Roberson, K.; von Mayrhauser, A.; *Data Flow Analysis and its Application to Software Maintenance*, Software Maintenance, 1988., In Proceedings of the Conference, Oct 1988, p.335 – 347.

[KIM99] Kim, K.; Park, J.; Yoon, Yong-Ik; *Graph of Change Impact Analysis for Distributed Object-Oriented Software,* In Proceedings of the Fuzzy Systems Conference, FUZZ-IEEE '99, IEEE International, Volume 2, Aug 1999, p.1137-1141.

[KUI00] Kuipers, T.; Moonen, L.; *Types and Concept Analysis for Legacy Systems*, Program Comprehension, 2000, In Proceedings of the 8th International IWPC, Oct-Nov 2000.

[LAW03] Law, J.; Rothermel, G.; *Whole Program Path-Based Dynamic Impact Analysis*, Software Engineering, 2003, In Proceedings of the 25th International Conference, May 2003, p.308-318.

[LEE98] Lee, M.; *Change Impact Analysis of Object-Oriented Software*, A Ph.D. dissertation in Information Technology at George Mason University, Fairfax, Virginia, 1998.

[LEE00] Lee, M.; Offutt, A.J.; Alexander, R.T.; *Algorithmic Analysis of the Impact of Changes to Object-Oriented Software*, Technology of Object-Oriented Languages and Systems, 2000, TOOLS 34, In Proceedings of the 34th International Conference on 30 July-4 Aug 2000, p.61-70.

[LIN95] Lindig, C.; *Concept-based Component Retrieval*, In Jana Köhler, Fausto Giunchiglia, Cordell Green, and Christoph Walther, editors, Working Notes of the IJCAI-95 Workshop: Formal Approaches to the Reuse of Plans, Proofs, and Programs, Montreál, Aug 1995, p. 21-25.

[LIN97] Lindig, C.; Snelting, G.; *Assessing Modular Structure of Legacy Code Based on Mathematical Concept Analysis*, Software Engineering, 1997., In Proceedings of the 19th International Conference, May 1997, p.349-359.

[LIN00] Lindig, C.; *Introduction to Formal Concept Analysis*, Harvard University, 2000, http://www.st.cs.uni-sb.de/~lindig/talks/.

[LIN02] Lindig, C.; *Fast Concept Analysis*, In Gerhard Stumme, editors, Working with Conceptual Structures - Contributions to ICCS 2000, Shaker Verlag, Aachen, Germany, 2000.

[LOY93] Loyall, J.P.; Mathisen, S.A.; *Using Dependence Analysis to Support the Software Maintenance Process*, Software Maintenance,1993. CSM-93, In Proceedings of the Conference, Sep 1993, p.282 – 291.

[MCQ05] McQuillan, J.A.; Power, J.F.; *A Survey of UML-Based Coverage Criteria for Software Testing*. Technical Report NUIM-CS-TR-2005-08, Department of Computer Science, NUI Maynooth, Ireland, Aug 2005.

[MIC] Microsoft Outlook, http://www.microsoft.com/outlook/.

[MON02] Moonen, L.; *Lightweight Impact Analysis using Island Grammars,* Program Comprehension, 2002, In Proceedings of the 10th International Workshop, Jun 2002, p.219-228.

[MOR90] Moreton, R., *A Process Model for Software Maintenance,* Journal Information Technology, Volume 5, 1990, p.100-104.

[MYE04] Myers, G. J.; *Art of Software Testing*, John Wiley & Sons, Inc., New York, NY, USA, 2004.

[NAV04] Navicon Company, http://www.navicon.de.

[ORS04] Orso, A.; Apiwattanapong, T.; Law, J.; Rothermel, G.; Harrold, M.J.; *An Empirical Comparison of Dynamic Impact Analysis Algorithms,* Software Engineering, 2004. ICSE 2004. In Proceedings of the 26th International Conference, May 2004, p.491-500.

[OST88] Ostrand, T.J. ; Weyuker, E.J., *Using Dataflow Analysis for Regression Testing*, Sizth Annual Pacific Northwest Software Quality Conference, Sep 1988, p.233-47.

[POS] PostgreSQL Relational Database System, http://www.postgresql.org/.

[RAP82] Rapps, S.;Weyuker, E. J.; *Data Flow Analysis Techniques for Test Data Selection*, In Proceedings of the 6th Irk Conference Software Engineering, Sep 1982, p. 272-277.

[RAP85] Rapps, S.;Weyuker, E. J.; *Selecting Software Test Data Using Data Flow Information*, IEEE Z'ranu. on Software Engineering, Apr 1985, p. 367-375.

[REN04] Ren, X.; Shah F.; Tip, F.; Ryder, F.; Chesley, O.; *A Tool for Change Impact Analysis of Java Programs*, In Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 2004), Vancouver, BC, Canada, Oct 2004, p.432-448.

[REP97] Reps, T.; Siff, M.; *Identifying Modules via Concept Analysis*, In Proceedings of the International Conference on Software Maintenance, Bari, Italy, Oct 1997, p.170-179.

[ROT93] Rothermel, G.; Harrold, M.; *A Safe, Efficient Algorithm for Regression Test Selection*, In Proceedings of the IEEE Software Maintenance Conference, 1993, p. 358-367.

[ROT96] Rothermel, G.; Harrold, M.; *Analyzing Regression Test Selection Techniques*, IEEE TSE, Aug 1996, p.529-551.

[SNE96] Snelting, G.; *Re-engineering of Configurations Based on Mathematical Concept Analysis*, TOSEM 5(2), 1996.

[SNE00] Snelting, G.; Tip, Frank; *Reengineering Class Hierarchies Using Concept Analysis*, ACM Transactions on Programming Languages and Systems (TOPLAS), May 2000, p.540-582.

[STU03] Stumme, G., Formal Concept Analysis, Otte-von-Guericke-Universität Magdeburg, 2003.

[TOC01] *Tockit Project - Framework for Concept Knowledge Processing*, http://tockit.sourceforge.net.

[TON01] Tonella, P.; *Concept Analysis for Module Restructuring*, Software Engineering, IEEE Transactions on , Volume 27 , Issue 4 , Apr 2001, p.351 – 363.

[TON03] Tonella, P., *Using a Concept Lattice of Decomposition Slices for Program Understanding and Impact Analysis*, IEEE Transactions on Software Engineering, Volume 29, No.6, Jun 2003.

[TOS01] *ToscanaJ Project*, DSTC, The University of Queensland, the Technical University of Darmstadt, http://toscanaj.sourceforge.net/.

[WAT99] Waters, R.; Rugaber, S.; Abowd, G.D.; *Architectural Element Matching using Concept Analysis*, Automated Software Engineering, 14th IEEE International Conference, Oct 1999, p. 291 -294.

[WHI92] White, L.J.; Leung, H.K.N., *A Firewall Concept for Both Control-flow and Data-flow in Regression Integration Testing*, In Proceedings of the Conference on *Soft*ware Maintenance, Nov 1992, p. 262-70.

[WIL82] Wille, R.; Restructuring Lattice Theory, *An Approach based on Hierarchies of Concepts*, In Rival, I., ed., Ordered Sets. Reidel, Dordrecht-Boston, 1982, p.445—470.

[WIL94] Queille, J.P., Voidrot, J.F., Wilde, N., Munro, M.; *The Impact Analysis Task in Software Maintenance: A Model and a Case Study*, In Proceedings of the International Conference on Software Maintenance, Victoria, Canada, IEEE Comp.Soc., Press, Sep 1994.

Appendices

Appendix A: Summary of Regression Test Selection Techniques [ROT96]

SUMMARY OF OUR FRAMEWORK-BASED EVALUATIONS OF REGRESSION TEST SELECTION TECHNIQUES

TECHNIQUE	INCLUSIVENESS	PRECISION	EFFICIENCY	GENERALITY
LINEAR EQUATION (minimization) (intra)	unsafe (all categories)	selects non-mt tests	worst case: exponential in $\|P\|$ in practice: unknown correspondence: $O(max(\|P\|,\|P'\|)^2)$	level: intra mods: not control flow criteria: control/dataflow requires: segment traces, linear equation solver
LINEAR EQUATION (non-minimization) (inter)	safe for controlled regression testing	selects non-mt tests selects all tests through modified procedures	worst case: exponential in $\|P\|$ in practice: unknown correspondence: $O(max(\|P\|,\|P'\|)^2 * log(max(\|P\|,\|P'\|))$	level: inter mods: handles all criteria: control/dataflow requires: function traces, linear equation solver
SYMBOLIC EXECUTION	not safe (for deletions)	selects non-mt tests	worst case: exponential in $\|P\|$ (may not terminate) in practice: unknown correspondence: $O(max(\|P\|,\|P'\|)^2 * log(max(\|P\|,\|P'\|))$	level: intra/inter mods: not deletions criteria: partition requires: symbolic execution
PATH ANALYSIS	not safe (for deletions or additions)	selects no non-mt tests	worst case: exponential in $\|P\|$ in practice: unknown correspondence: not required	level: intra mods: not deletions/additions criteria: path requires: statement traces, algebraic design
DATAFLOW (incremental)	not safe (all categories)	selects non-mt tests	worst case: $O(\|T\| * \|P'\|^2)$ per modification in practice: unknown correspondence: not required	level: intra/inter mods: only dataflow affecting criteria: dataflow requires: basic block traces, static and incremental dataflow analysis tools
PROGRAM DEPENDENCE GRAPH	not safe (for deletions)	selects non-mt tests less precise than SDG technique	worst case: $O(\|T\| * (max(\|P\|,\|P'\|)^3)$ or $O(\|T\| * (max(\|P\|,\|P'\|)^4)$ in practice: unknown correspondence: $O(max(\|P\|,\|P'\|)^2)$	level: intra mods: not deletions criteria: PDG requires: statement traces, control dependence, slicing, dataflow analysis tools
SYSTEM DEPENDENCE GRAPH	not safe (for deletions)	selects non-mt tests more precise than PDG technique	worst case: $O(\|T\| * (max(\|P\|,\|P'\|)^3)$ or $O(\|T\| * (max(\|P\|,\|P'\|)^4)$ in practice: unknown correspondence: $O(max(\|P\|,\|P'\|)^2)$	level: intra/inter mods: not deletions criteria: PDG requires: statement traces, control dependence, slicing, dataflow analysis tools
MODIFICATION BASED	not safe (all categories) (minimization)	unknown	worst case: $O(\|T\| * \|P'\|^2)$ per logical modification for analysis; test selection is not automated. in practice: unknown correspondence: $O(max(\|P\|,\|P'\|)^2)$	level: intra/inter mods: doesn't handle all criteria: none requires: statement traces, static/dynamic control and data dependence analysis
FIREWALL	safe for controlled regression testing if T is reliable	selects non-mt tests selects all tests through modified procedures	worst case: $O(\|T\| * (max(\|P\|,\|P'\|))$ in practice: "efficient" correspondence: $O(max(\|P\|,\|P'\|)^2 * log(max(\|P\|,\|P'\|))$	level: inter mods: handles all criteria: none requires: function traces
CLUSTER IDENTIFICATION	safe for p.r.t.	selects non-mt tests less precise than graph walk technique	worst case: $O(\|T\| * max(\|P\|,\|P'\|)^3)$ in practice: unknown correspondence: not required	level: intra mods: handles all criteria: none requires: statement traces CFGs, control dependence
SLICING	not safe (some categories)	selects non-mt tests	worst case: $O(\|T\| * \|P'\|^2)$ per modification in practice: unknown correspondence: not required	level: intra mods: doesn't handle all criteria: none requires: statement traces, static/dynamic control and data dependence analysis
GRAPH WALK	safe for controlled regression testing	selects non-mt tests (but not in practice) most precise safe technique	worst case: $O(\|T\| * (max(\|P\|,\|P'\|)^2)$ in practice: $O(\|T\| * min(\|P\|,\|P'\|)$ (without dataflow information) correspondence: not required	level: intra/inter mods: handles all criteria: none requires: statement traces, dataflow analysis (optional)
MODIFIED ENTITY	safe for controlled regression testing	selects non-mt tests selects all tests through modified procedures less precise than graph walk, cluster ident.	worst case: $O(\|T\| * \|P\|)$ in practice: "efficient" correspondence: $O(max(\|P\|,\|P'\|) * log(max(\|P\|,\|P'\|))$	level: inter mods: handles all criteria: none requires: function traces, code database

Appendix B: Source Code of Calc Program

plus.java

```
public class plus{

        5-14    public static double calculate(double a, double b){
        5-15            return a+b;
                }
}
```

minus.java

```
public class minus{

        6-14    public static double calculate(double a, double b){
        6-15            return a-b;
                }
}
```

multiply.java

```
public class multiply{

        7-14    public static double calculate(double a, double b){
        7-15            return a*b;
                }
}
```

divide.java

```
public class divide{

        8-14    public static double calculate(double a, double b){
        8-15            return a/b;
                }
}
```

sum.java

```
import java.util.Vector;

public class sum{

        9-15    public static double calculate(Vector x){
        9-16            double result = 0;

        9-18            for (int i=0; i<x.size(); i++){
        9-19            result = plus.calculate(result,
                                Double.parseDouble(x.elementAt(i).toString()));
                }
```

```
9-21            return result;
            }
}
```

average.java

```
import java.util.Vector;

public class average{

10-15   public static double calculate(Vector x){
10-16       double result = 0;

10-18       for (int i=0; i<x.size(); i++){
10-19           result = plus.calculate(result,
                Double.parseDouble(x.elementAt(i).toString()));
            }
10-21       return divide.calculate(result,x.size());
        }
}
```

console.java

```
import java.io.IOException;
import java.util.Vector;

public class console {
11-19   public static void main(String[] args) throws IOException {
11-20       Vector x = new Vector();
11-21       x.addElement("1");
11-22       x.addElement("2");
11-23       x.addElement("3");
11-24       x.addElement("4");

11-26       double a = 7;
11-27       double b = 2;

11-29       switch (Integer.parseInt(args[0])) {
            case 1:
                //Trace 1
11-32           System.out.println(a + " + " + b + " = " + plus.calculate(a, b));
11-33           break;
            case 2:
                //Trace 2
11-36           System.out.println(a + " - " + b + " = " + minus.calculate(a, b));
11-37           break;
            case 3:
                //Trace 3
11-40           System.out.println(a + " * " + b + " = " + multiply.calculate(a, b));
11-41           break;
            case 4:
                //Trace 4
11-44           System.out.println(a + " / " + b + " = " + divide.calculate(a, b));
11-45           break;
            case 5:
                //Trace 5
11-48           System.out.println("sum " + x.toString() + " = " + sum.calculate(x));
11-49           break;
            case 6:
                //Trace 6
11-52           System.out.println("average " + x.toString() + " = " +
```

```
                    average.calculate(x));
11-53               break;
                }
            }
}
```

Appendix C: Instrumentation result of Calc program

plus.java

```
package calc;


/**
 * @author Pabhanin Leelahapant
 */

public class plus{

   public static double calculate(double a, double b) {/*_I*/instr.InstrUtil.showLine(5, 14, "D:\\trace.txt");/*I_*/
         /*_I*/{instr.InstrUtil.showLine(5, 15, "D:\\trace.txt");/*I_*/return a+b;/*_I*/}/*I_*/
   }
}
```

minus.java

```
package calc;

/**
 * @author Pabhanin Leelahapant
 */

public class minus{

   public static double calculate(double a, double b) {/*_I*/instr.InstrUtil.showLine(6, 14, "D:\\trace.txt");/*I_*/
         /*_I*/{instr.InstrUtil.showLine(6, 15, "D:\\trace.txt");/*I_*/return a-b;/*_I*/}/*I_*/
   }
}
```

multiply.java

```
package calc;

/**
 * @author Pabhanin Leelahapant
 */
public class multiply{

   public static double calculate(double a, double b) {/*_I*/instr.InstrUtil.showLine(7, 14, "D:\\trace.txt");/*I_*/
         /*_I*/{instr.InstrUtil.showLine(7, 15, "D:\\trace.txt");/*I_*/return a*b;/*_I*/}/*I_*/
   }
}
```

divide.java

```
package calc;

/**
 * @author Pabhanin Leelahapant
 */

public class divide{
```

```
   public static double calculate(double a, double b) {/*_I*/instr.InstrUtil.showLine(8, 14, "D:\\trace.txt");/*I_*/
         /*_I*/{instr.InstrUtil.showLine(8, 15, "D:\\trace.txt");/*I_*/return a/b;/*_I*/}/*I_*/
   }
}
```

sum.java

```
package calc;

import java.util.Vector;

/**
 * @author  Pabhanin Leelahapant
 */

public class sum{

   public static double calculate(Vector x) {/*_I*/instr.InstrUtil.showLine(9, 15, "D:\\trace.txt");/*I_*/
         /*_I*/instr.InstrUtil.showLine(9, 16, "D:\\trace.txt");/*I_*/double result = 0;

                  /*_I*/{instr.InstrUtil.showLine(9, 18, "D:\\trace.txt");/*I_*/for (int i=0; i<x.size(); i++){
                        /*_I*/{instr.InstrUtil.showLine(9, 19, "D:\\trace.txt");/*I_*/result = plus.calculate(result,
Double.parseDouble(x.elementAt(i).toString()));/*_I*/}/*I_*/
                  }/*_I*/}/*I_*/
                  /*_I*/{instr.InstrUtil.showLine(9, 21, "D:\\trace.txt");/*I_*/return result;/*_I*/}/*I_*/
   }
}
```

average.java

```
package calc;

import java.util.Vector;

/**
   * @author  Pabhanin Leelahapant
 */

public class average{

   public static double calculate(Vector x){/*_I*/instr.InstrUtil.showLine(10, 15,  "D:\\trace.txt");/*I_*/
         /*_I*/instr.InstrUtil.showLine(10, 16 "D:\\trace.txt");/*I_*/double result = 0;

                  /*_I*/{instr.InstrUtil.showLine(10, 18,  "D:\\trace.txt");/*I_*/for (int i=0; i<x.size(); i++){
                        /*_I*/{instr.InstrUtil.showLine(10, 19, "D:\\trace.txt");/*I_*/result = plus.calculate(result,
Double.parseDouble(x.elementAt(i).toString()));/*_I*/}/*I_*/
                  }/*_I*/}/*I_*/
                  /*_I*/{instr.InstrUtil.showLine(10, 21, "D:\\trace.txt");/*I_*/return divide.calculate(result,x.size());
/*_I*/}/*I_*/
   }
}
```

console.java

```
package calc;

import java.io.IOException;
import java.util.Vector;

/*
 * @author Pabhanin Leelahapant
```

```
*/

public class console {
	public static void main(String[] args) throws IOException {/*_I*/instr.InstrUtil.showLine(11, 19,
"D:\\trace.txt");/*I_*/
		/*_I*/instr.InstrUtil.showLine(11, 20, "D:\\trace.txt");/*I_*/Vector x = new Vector();
		/*_I*/{instr.InstrUtil.showLine(11, 21, "D:\\trace.txt");/*I_*/x.addElement("1");/*_I*/}/*I_*/
		/*_I*/{instr.InstrUtil.showLine(11, 22, "D:\\trace.txt");/*I_*/x.addElement("2");/*_I*/}/*I_*/
		/*_I*/{instr.InstrUtil.showLine(11, 23, "D:\\trace.txt");/*I_*/x.addElement("3");/*_I*/}/*I_*/
		/*_I*/{instr.InstrUtil.showLine(11, 24, "D:\\trace.txt");/*I_*/x.addElement("4");/*_I*/}/*I_*/

		/*_I*/instr.InstrUtil.showLine(11, 26, "D:\\trace.txt");/*I_*/double a = 7;
		/*_I*/instr.InstrUtil.showLine(11, 27, "D:\\trace.txt");/*I_*/double b = 2;

		/*_I*/{instr.InstrUtil.showLine(11, 29, "D:\\trace.txt");/*I_*/switch (Integer.parseInt(args[0])) {
			case 1:
				//Trace 1
				/*_I*/{instr.InstrUtil.showLine(11, 32,
"D:\\trace.txt");/*I_*/System.out.println(a + " + " + b + " = " + plus.calculate(a, b));/*_I*/}/*I_*/
				/*_I*/{instr.InstrUtil.showLine(11, 33,
"D:\\trace.txt");/*I_*/break;/*_I*/}/*I_*/
			case 2:
				//Trace 2
				/*_I*/{instr.InstrUtil.showLine(11, 36,
"D:\\trace.txt");/*I_*/System.out.println(a + " - " + b + " = " + minus.calculate(a, b));/*_I*/}/*I_*/
				/*_I*/{instr.InstrUtil.showLine(11, 37,
"D:\\trace.txt");/*I_*/break;/*_I*/}/*I_*/
			case 3:
				//Trace 3
				/*_I*/{instr.InstrUtil.showLine(11, 40,
"D:\\trace.txt");/*I_*/System.out.println(a + " * " + b + " = " + multiply.calculate(a, b));/*_I*/}/*I_*/
				/*_I*/{instr.InstrUtil.showLine(11, 41,
"D:\\trace.txt");/*I_*/break;/*_I*/}/*I_*/
			case 4:
				//Trace 4
				/*_I*/{instr.InstrUtil.showLine(11, 44,
"D:\\trace.txt");/*I_*/System.out.println(a + " / " + b + " = " + divide.calculate(a, b));/*_I*/}/*I_*/
				/*_I*/{instr.InstrUtil.showLine(11, 45,
"D:\\trace.txt");/*I_*/break;/*_I*/}/*I_*/
			case 5:
				//Trace 5
				/*_I*/{instr.InstrUtil.showLine(11, 48,
"D:\\trace.txt");/*I_*/System.out.println("sum " + x.toString() + " = " + sum.calculate(x));/*_I*/}/*I_*/
				/*_I*/{instr.InstrUtil.showLine(11, 49,
"D:\\trace.txt");/*I_*/break;/*_I*/}/*I_*/
			case 6:
				//Trace 6
				/*_I*/{instr.InstrUtil.showLine(11, 52,
"D:\\trace.txt");/*I_*/System.out.println("average " + x.toString() + " = " + average.calculate(x));/*_I*/}/*I_*/
				/*_I*/{instr.InstrUtil.showLine(11, 53,
"D:\\trace.txt");/*I_*/break;/*_I*/}/*I_*/
		}/*_I*/}/*I_*/
	}
}
```

Appendix D: Source Code of Test Case Selection Algorithm

```
/*
 *
 * Collect all the nodes to be retest from the program dependency lattice
 *
 */
package fca.util;


import java.io.BufferedReader;
import java.io.File;
import java.io.FileInputStream;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.Vector;

/**
 * @author Pabhanin Leelahapant
 */
public class RetestNodeCollector{

	public static void main(String[] args) throws IOException {
		BufferedReader in  = new BufferedReader(new InputStreamReader(System.in));

		System.out.print("Please enter the number of node of interest >> ");
		String strNodeOfInterest = in.readLine();

		String fileName = "D:\\_CaseStudies\\Calc2\\calc2.dot";

		//read text file
		File fInput;
		FileInputStream fisInput = null;
		String line;

		try {
			/* == open file & check if it is valid == */
			fInput = new File(fileName);
			if (!fInput.exists()){
				System.out.println("File " + fileName + " does not exist");
				//return null
			}
			if (!fInput.canRead()){
				System.out.println("File " + fileName + " cannot be read");
				//return null
			}

			fisInput = new FileInputStream(fInput);
			BufferedReader dotFile = new BufferedReader(new InputStreamReader(fisInput));

			/* == Create a child list == */
			Vector vChildList = new Vector();
			Vector vTmp = new Vector();

			// == -- flush all headers
			do {
				line = dotFile.readLine();
			}
			while(line!=null && !(line.startsWith("node1 --")));
```

```
int i = 1;
while(!line.startsWith("}")){

        vTmp = new Vector();

        // get the child list for each node
        System.out.println(line);

        while (!(line.startsWith("node" + (i) + " --"))){
                if (line.startsWith("}")) break;

                vTmp.add(line.substring(12 + String.valueOf(i-1).length(),
                line.length()-1));

                line = dotFile.readLine();
        }
        System.out.println(vTmp.toString());

        vChildList.add(vTmp);
        i++;
}

/* == Parse Dependency Tree == */
System.out.println("*************** Parse Dependency Tree & Collect leaves
******************");

Vector v = new Vector();
Vector vLeaves = new Vector();
System.out.println("v : " + v.toString());

// == -- Put the node of interest in the queue
v.addElement(strNodeOfInterest);
System.out.println("v : " + v.toString());

// == -- Loop until there's no element left
while (!v.isEmpty()){
        // == -- Get the children collection of the first node
        String firstElem = String.valueOf(v.firstElement());
        Vector vChildren = (Vector)vChildList.elementAt(Integer.parseInt(firstElem));
        System.out.println("children of " + firstElem + " : " + vChildren.toString());

        // == -- Enqueue its children
        v.addAll(vChildren);
        System.out.println("v : " + v.toString());

        // == -- Collect leaf nodes
        if (!vChildren.isEmpty()){
                String firstChild = String.valueOf(vChildren.firstElement());

                System.out.println(vChildren.firstElement());

                // == -- 1> Check if it's a leaf
                // == -- 2> Exclude duplicate node
                if (firstChild.compareTo("0")==0 && !vLeaves.contains(firstElem)){
                        vLeaves.addElement(firstElem);
                }
        }

        // == -- Dequeue the head
        v.removeElementAt(0);
        System.out.println("v : " + v.toString());
        System.out.println("vLeaves : " + vLeaves.toString());
```

```
                    }

                }
                catch (IOException e){
                    System.out.println("Error : " + e.getMessage());
                    return;
                }
            }

}
```

Appendix E: Screenshot of Java FCA Calculation Module

********** Welcome to Java FCA **********
****
******* Context has 2 hashtables **********

******* 1. Context.hashObjects **********
******* Common attributes from Objects perspective ********
multiply : [7-15, 11-24, 7-14, 11-23, 11-22, 11-19, 11-21, 11-20, 11-41, 11-40, 11-29, 11-27, 11-26]
sum : [11-29, 11-27, 11-26, 5-15, 11-24, 5-14, 9-21, 11-23, 11-22, 11-21, 11-20, 9-19, 9-18, 11-19, 11-49, 9-16, 9-15, 11-48]
average : [11-29, 10-21, 11-27, 11-26, 5-15, 11-24, 5-14, 11-23, 11-53, 11-22, 11-21, 11-52, 11-20, 10-19, 10-18, 8-15, 8-14, 10-16, 10-15, 11-19]
minus : [11-24, 11-23, 11-22, 11-19, 11-21, 11-20, 6-15, 6-14, 11-37, 11-36, 11-29, 11-27, 11-26]
divide : [11-24, 11-23, 11-22, 11-19, 11-21, 11-20, 11-45, 11-44, 8-15, 8-14, 11-29, 11-27, 11-26]
plus : [11-24, 11-23, 11-22, 11-19, 11-21, 11-20, 5-15, 5-14, 11-33, 11-32, 11-29, 11-27, 11-26]
Order of Objects is = [plus, minus, multiply, divide, sum, average]

******* 2. Context.hashAttributes **********
******* Common objects from Attribute perspective ********
hshAttributes.size = 39
5-14 : [sum, average, plus]
11-41 : [multiply]
11-40 : [multiply]
8-15 : [average, divide]
8-14 : [average, divide]
10-21 : [average]
11-37 : [minus]
11-36 : [minus]
11-33 : [plus]
11-32 : [plus]
9-21 : [sum]
6-15 : [minus]
6-14 : [minus]
10-19 : [average]
10-18 : [average]
10-16 : [average]
10-15 : [average]
11-29 : [multiply, sum, minus, average, plus, divide]
9-19 : [sum]
11-27 : [multiply, sum, minus, average, plus, divide]
9-18 : [sum]
11-26 : [multiply, sum, minus, average, plus, divide]
9-16 : [sum]
11-24 : [multiply, sum, minus, average, plus, divide]
9-15 : [sum]
11-23 : [multiply, sum, minus, average, plus, divide]
11-22 : [multiply, sum, minus, average, plus, divide]
11-21 : [multiply, sum, minus, average, plus, divide]
11-53 : [average]
11-20 : [multiply, sum, minus, average, plus, divide]
11-52 : [average]
7-15 : [multiply]
7-14 : [multiply]
11-19 : [multiply, sum, minus, average, plus, divide]
11-49 : [sum]
11-48 : [sum]
11-45 : [divide]

11-44 : [divide]
5-15 : [sum, average, plus]

****
RELATION TABLE
--- each number in a bitset represent the bit that is set corresponding its position in obj/attr orderedlist
****
Order of Objects is [plus, minus, multiply, divide, sum, average]
Obj[0] = plus
common objs = {1, 3, 4, 5, 6, 7, 10, 11, 12, 13, 14, 37, 38}
Obj[1] = minus
common objs = {1, 3, 4, 5, 6, 7, 10, 11, 12, 31, 32, 33, 34}
Obj[2] = multiply
common objs = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12}
Obj[3] = divide
common objs = {1, 3, 4, 5, 6, 7, 10, 11, 12, 27, 28, 35, 36}
Obj[4] = sum
common objs = {1, 3, 4, 5, 6, 7, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21}
Obj[5] = average
common objs = {1, 3, 4, 5, 6, 7, 10, 11, 12, 13, 14, 22, 23, 24, 25, 26, 27, 28, 29, 30}
****
Order of Attributes is [7-15, 11-24, 7-14, 11-23, 11-22, 11-19, 11-21, 11-20, 11-41, 11-40, 11-29, 11-27, 11-26, 5-15, 5-14, 9-21, 9-19, 9-18, 11-49, 9-16, 9-15, 11-48, 10-21, 11-53, 11-52, 10-19, 10-18, 8-15, 8-14, 10-16, 10-15, 6-15, 6-14, 11-37, 11-36, 11-45, 11-44, 11-33, 11-32]
Attr[0] = 7-15
common attr = {2}
Attr[1] = 11-24
common attr = {0, 1, 2, 3, 4, 5}
Attr[2] = 7-14
common attr = {2}
Attr[3] = 11-23
common attr = {0, 1, 2, 3, 4, 5}
Attr[4] = 11-22
common attr = {0, 1, 2, 3, 4, 5}
Attr[5] = 11-19
common attr = {0, 1, 2, 3, 4, 5}
Attr[6] = 11-21
common attr = {0, 1, 2, 3, 4, 5}
Attr[7] = 11-20
common attr = {0, 1, 2, 3, 4, 5}
Attr[8] = 11-41
common attr = {2}
Attr[9] = 11-40
common attr = {2}
Attr[10] = 11-29
common attr = {0, 1, 2, 3, 4, 5}
Attr[11] = 11-27
common attr = {0, 1, 2, 3, 4, 5}
Attr[12] = 11-26
common attr = {0, 1, 2, 3, 4, 5}
Attr[13] = 5-15
common attr = {0, 4, 5}
Attr[14] = 5-14
common attr = {0, 4, 5}
Attr[15] = 9-21
common attr = {4}
Attr[16] = 9-19
common attr = {4}
Attr[17] = 9-18
common attr = {4}
Attr[18] = 11-49
common attr = {4}

Attr[19] = 9-16
common attr = {4}
Attr[20] = 9-15
common attr = {4}
Attr[21] = 11-48
common attr = {4}
Attr[22] = 10-21
common attr = {5}
Attr[23] = 11-53
common attr = {5}
Attr[24] = 11-52
common attr = {5}
Attr[25] = 10-19
common attr = {5}
Attr[26] = 10-18
common attr = {5}
Attr[27] = 8-15
common attr = {3, 5}
Attr[28] = 8-14
common attr = {3, 5}
Attr[29] = 10-16
common attr = {5}
Attr[30] = 10-15
common attr = {5}
Attr[31] = 6-15
common attr = {1}
Attr[32] = 6-14
common attr = {1}
Attr[33] = 11-37
common attr = {1}
Attr[34] = 11-36
common attr = {1}
Attr[35] = 11-45
common attr = {3}
Attr[36] = 11-44
common attr = {3}
Attr[37] = 11-33
common attr = {0}
Attr[38] = 11-32
common attr = {0}
****** Finish calculating lattice **********
****** Call : PrintToScreen

********** Concept Lattice
****
object list = [plus, minus, multiply, divide, sum, average]
attribute list = [7-15, 11-24, 7-14, 11-23, 11-22, 11-19, 11-21, 11-20, 11-41, 11-40, 11-29, 11-27, 11-26, 5-15, 5-14, 9-21, 9-19, 9-18, 11-49, 9-16, 9-15, 11-48, 10-21, 11-53, 11-52, 10-19, 10-18, 8-15, 8-14, 10-16, 10-15, 6-15, 6-14, 11-37, 11-36, 11-45, 11-44, 11-33, 11-32]
number of objects = 6
number of attributes = 39
number of concepts = 10
concept[0] = ({},{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38})
concept[1] = ({5},{1, 3, 4, 5, 6, 7, 10, 11, 12, 13, 14, 22, 23, 24, 25, 26, 27, 28, 29, 30})
concept[2] = ({4},{1, 3, 4, 5, 6, 7, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21})
concept[3] = ({3},{1, 3, 4, 5, 6, 7, 10, 11, 12, 27, 28, 35, 36})
concept[4] = ({3, 5},{1, 3, 4, 5, 6, 7, 10, 11, 12, 27, 28})
concept[5] = ({2},{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12})
concept[6] = ({1},{1, 3, 4, 5, 6, 7, 10, 11, 12, 31, 32, 33, 34})
concept[7] = ({0},{1, 3, 4, 5, 6, 7, 10, 11, 12, 13, 14, 37, 38})
concept[8] = ({0, 4, 5},{1, 3, 4, 5, 6, 7, 10, 11, 12, 13, 14})

concept[9] = ({0, 1, 2, 3, 4, 5},{1, 3, 4, 5, 6, 7, 10, 11, 12})

********** Lattice in Graphplace format **********
%%!PS
%%%% number of objects: 6
%%%% number of attributes: 39
%%%% number of concepts: 10

() () (11-24 11-23 11-22 11-19 11-21 11-20 11-29 11-27 11-26) 9 node
() () (5-15 5-14) 8 node
(plus) () (11-33 11-32) 7 node
(minus) () (6-15 6-14 11-37 11-36) 6 node
(multiply) () (7-15 7-14 11-41 11-40) 5 node
() () (8-15 8-14) 4 node
(divide) () (11-45 11-44) 3 node
(sum) () (9-21 9-19 9-18 11-49 9-16 9-15 11-48) 2 node
(average) () (10-21 11-53 11-52 10-19 10-18 10-16 10-15) 1 node
() () () 0 node

1 0 edge
2 0 edge
3 0 edge
4 1 edge
4 3 edge
5 0 edge
6 0 edge
7 0 edge
8 1 edge
8 2 edge
8 7 edge
9 4 edge
9 5 edge
9 6 edge
9 8 edge

********** Lattice in GraphViz format **********
graph conceptlattice {
label = "concept lattice"
node9[label="\n11-24 11-23 11-22 11-19 11-21 11-20 11-29 11-27 11-26",fontsize=10,labelfloat=true];
node8[label="\n5-15 5-14",fontsize=10,labelfloat=true];
node7[label="plus\n11-33 11-32",fontsize=10,labelfloat=true];
node6[label="minus\n6-15 6-14 11-37 11-36",fontsize=10,labelfloat=true];
node5[label="multiply\n7-15 7-14 11-41 11-40",fontsize=10,labelfloat=true];
node4[label="\n8-15 8-14",fontsize=10,labelfloat=true];
node3[label="divide\n11-45 11-44",fontsize=10,labelfloat=true];
node2[label="sum\n9-21 9-19 9-18 11-49 9-16 9-15 11-48",fontsize=10,labelfloat=true];
node1[label="average\n10-21 11-53 11-52 10-19 10-18 10-16 10-15",fontsize=10,labelfloat=true];
node0[label="\n",fontsize=10,labelfloat=true];
node1 -- node0;
node2 -- node0;
node3 -- node0;
node4 -- node1;
node4 -- node3;
node5 -- node0;
node6 -- node0;
node7 -- node0;
node8 -- node1;
node8 -- node2;
node8 -- node7;
node9 -- node4;
node9 -- node5;
node9 -- node6;

```
node9 -- node8;
}
```

Finish writing

Appendix F: Sample Run of Test Case Selection Module

```
Please enter the number of node of interest >> 10

********** Building a child list for each node **********
node 0 : []
node 1 : [0]
node 2 : [0]
node 3 : [0]
node 4 : [2, 3]
node 5 : [4]
node 6 : [0]
node 7 : [0]
node 8 : [7]
node 9 : [4]
node 10 : [6, 9]
node 11 : [5, 8, 10]
node 12 : [0]
node 13 : [0]
node 14 : [1, 7, 12, 13]
node 15 : [11, 14]

********** Parse Dependency Tree & Collect leaves **********
Queue : []

Enqueue node of interest -> 10
Queue : [10]
Enqueue children of node 10
Queue : [10, 6, 9]
Output => []
Dequeue head

Queue : [6, 9]
Enqueue children of node 6
Queue : [6, 9, 0]
Output => [6]
Dequeue head

Queue : [9, 0]
Enqueue children of node 9
Queue : [9, 0, 4]
Output => [6]
Dequeue head

Queue : [0, 4]
Enqueue children of node 0
Queue : [0, 4]
Output => [6]
Dequeue head

Queue : [4]
Enqueue children of node 4
Queue : [4, 2, 3]
Output => [6]
Dequeue head

Queue : [2, 3]
Enqueue children of node 2
```

Queue : [2, 3, 0]
Output => [6, 2]
Dequeue head

Queue : [3, 0]
Enqueue children of node 3
Queue : [3, 0, 0]
Output => [6, 2, 3]
Dequeue head

Queue : [0, 0]
Enqueue children of node 0
Queue : [0, 0]
Output => [6, 2, 3]
Dequeue head

Queue : [0]
Enqueue children of node 0
Queue : [0]
Output => [6, 2, 3]
Dequeue head

****** List of test cases to be retested after node 10 has changed *******

Test case #1 : MEDIAN
Test case #2 : SD
Test case #3 : VAR