NOTE TO USERS

This reproduction is the best copy available.

Program Slicing based Source Code Feature Extraction

Susmita Haldar

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

April, 2005

© Susmita Haldar, 2005

Library and Archives Canada

Published Heritage Branch

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque et Archives Canada

Direction du Patrimoine de l'édition

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence
ISBN: 0-494-04445-4
Our file Notre référence
ISBN: 0-494-04445-4

NOTICE:
The author has granted a non-exclusive license allowing Library and Archives Canada to reproduce, publish, archive, preserve, conserve, communicate to the public by telecommunication or on the Internet, loan, distribute and sell theses worldwide, for commercial or non-commercial purposes, in microform, paper, electronic and/or any other formats.

The author retains copyright ownership and moral rights in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

AVIS:
L'auteur a accordé une licence non exclusive permettant à la Bibliothèque et Archives Canada de reproduire, publier, archiver, sauvegarder, conserver, transmettre au public par télécommunication ou par l'Internet, prêter, distribuer et vendre des thèses partout dans le monde, à des fins commerciales ou autres, sur support microforme, papier, électronique et/ou autres formats.

L'auteur conserve la propriété du droit d'auteur et des droits moraux qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

In compliance with the Canadian Privacy Act some supporting forms may have been removed from this thesis.

While these forms may be included in the document page count, their removal does not represent any loss of content from the thesis.

Conformément à la loi canadienne sur la protection de la vie privée, quelques formulaires secondaires ont été enlevés de cette thèse.

Bien que ces formulaires aient inclus dans la pagination, il n'y aura aucun contenu manquant.

Abstract

Program Slicing Based Source Code Feature Extraction

Susmita Haldar

Program comprehension is an essential part of software maintenance. As software systems are becoming larger, understanding the whole program without prior knowledge is a hard task for the developer and the maintainer. Our approach of feature extraction reduces the program understanding complexity by identifying features based on input and output statements at the source code level. The presented feature extraction approach is a semi-automatic approach that only requires source code and test cases to identify and extract features. This approach utilizes program slicing, a program reduction technique to extract statements that implement an identified feature. The algorithm is implemented within the CONCEPT (Comprehension Of Net-CEntered Programs and Techniques) project. A case study using an open source project called JUnit was conducted to evaluate the applicability of the proposed approach.

Acknowledgments

I would like to express my gratitude to my supervisor Dr. Juergen Rilling for his kind support and guidance throughout this work. Without his guidance the accomplishment of this task would be impossible.

Also, I would like to give thanks to Beverly and Josh Beherman for proofreading this thesis.

Finally, special thanks to my parents, my husband, and my brother for their encouragement and support to help me accomplish this research work.

Table of contents

Table of figures vi
List of tables viii
1 Introduction 1
1.1 Features 1
1.1.1 Functional feature 3
1.1.2 Non-functional feature 3
1.2 Motivation 4
1.3 Thesis outline 4
2 Background/Literature survey 6
2.1 Software Comprehension and reverse engineering 6
2.2 Documentation versus source code-driven feature extraction 8
2.2.1 Non-source code based 9
2.2.2 Feature extraction based on source code 17
2.3 Source code analysis based 21
2.4 Discussion and limitation of existing feature extraction methods 30
2.5 Existing tools 35
2.6 Limitations of feature extraction tools 40
3 Source code-based feature extraction approach 43
3.1 Motivation 43
3.2 Feature categorization 46
3.3 Research approach 50
3.4 Feature extraction algorithm 55
3.5 Description of the backward slicing algorithm 59
3.6 Description of the forward slicing algorithm 61
4 Implementation 72
4.1 Concept system architecture 72
4.2 Feature extraction algorithm implementation architecture 74
4.3 Case study and experimental result 77
4.3.1 Experimental results 81
4.4 Application of extracted features 91
4.5 Related work 103
5 Conclusions and future work 109
Bibliography 112
Appendix 1 The Account.java program 121
Appendix 2 An example of the computation of dynamic backward algorithm [Kor95] 122
Appendix 3 Identified output statements and output feature criteria from the ResultPrinter class for test case 1 124

Table of figures

Figure 1 Use case diagram of an automated banking system 11
Figure 2 Diagram of a Use Case Map representation [Amy00b]..................................... 13
Figure 3 The phases of feature oriented domain analysis [Kru93] 16
Figure 4 An example of source code and the PDG representation of the source code 26
Figure 5 The computed slice from Figure 4 26
Figure 6 A sample source code 28
Figure 7 Forward slicing computation of Figure 6 using Korel's algorithm 28
Figure 8 Computed slice using Horwitz approach for slicing criterion <3,c> 29
Figure 9 Traditional approaches to feature extraction 44
Figure 10 Feature extraction process based on our research 45
Figure 11 Example of a static feature 48
Figure 12 Example of a dynamic feature 48
Figure 13 Example of an informational feature 49
Figure 14 Example of a functional feature 49
Figure 15 Method for extracting output feature using backward slicing 51
Figure 16 An example of the output feature extraction technique 52
Figure 17 Provides an overview of the feature extraction approach for input features. ... 53
Figure 18 Input feature extraction techniques using the combination of backward and forward slicing 54
Figure 19 Algorithm for computation of input and output feature 58
Figure 20 a) Last definition example b) First usage reference example 61
Figure 21 Modified method from backward slicing algorithm 62
Figure 22 Modified procedure added to the algorithm 63
Figure 23 Sample program with a constructor call 64
Figure 24 Execution Trace of the sample program of Figure 23 64
Figure 25 Integrate backward slicing with forward slicing algorithm 65
Figure 26 Removable blocks of the sample program 66
Figure 27 Execution trace and blocks traces of the sample program 67
Figure 28 Highlighted output feature of the account program 68
Figure 29 The extracted input feature 71
Figure 30 System architecture of the CONCEPT framework 73
Figure 31 High level view of the feature extraction algorithm implementation 76
Figure 32 A screen capture of the part of the output features from JUnit 77
Figure 33 VectorTest class 80
Figure 34 Test results generated by JUnit 80
Figure 35 Executed junit and file id for each execution that was used in 3 separate experiments 82
Figure 36 Partial listing of modified VectorTest class 88
Figure 37 Output produced by the modified Vector Test class 88
Figure 38 A feature extracted by the sample program 89
Figure 39 Test case with input feature 90
Figure 40 Output from the AllTest class 90
Figure 41 An example of an input feature 91
Figure 42 Measuring coupling and cohesion of the system 93

Figure 43 The overlapping computation formula and overlapping example 97
Figure 44 Applications of feature extraction technique 98
Figure 45 Partial listing of ResultPrinter 101
Figure 46 Example of the limitation with the invoking objects 102
Figure 47 Differences between two features identified with different test cases 106

List of tables

Table 1 Disadvantages of Feature extraction based on the document based technique.... 32
Table 2 The disadvantages of feature extraction from source code 34
Table 3 Limitations of current feature extraction tools 42
Table 4 An overview of the JUnit project 78
Table 5 The computed features [Continued] 83
Table 6 The result obtained from the JUnit when applying feature extraction algorithm 85
Table 7 Features overlapping percentage from JUnit 95
Table 8 The result of Feature 1 and Feature 2 overlapping 96

1 Introduction

From a user perspective, a software system can be viewed as a black box which provides a set of features that end users must utilize in order to facilitate their tasks. Software systems have to evolve in order to respond to market needs. Software developers tend to give preference to the end users' requirements as the modifications usually starts with the customers' request to make changes [Meh02]. They start enhancing an existing application by familiarizing themselves with the software application. However, the software systems are typically large, and might be written in different or obsolete programming languages that the programmer might not be familiar with [Eis01]. Therefore, it is essential to provide both developers and maintainers with means to focus their attention on these parts of the source code that are relevant to perform the desired maintenance task. This is done to reduce time and cost associated with these maintenance tasks. One approach to reduce the comprehension complexity is to identify a group of source code statements or components which correspond to a certain requirement or feature of the system [Mur01].

1.1 Features

Having a mental representation of the application is an important factor in providing effective software maintenance and evolution [May95]. Furthermore, understanding how a certain feature is implemented is crucial in program understanding, especially when the understanding is directed to a certain goal such as modifying or extending the features [Eis01

In order to deal with identifying features and their applications, the traditional definition of features needs to be understood. Different researchers have used different definitions of feature based on the context of their research area [Kru93, Tur99b, Wil95, Eis01b, Meh01a]. The IEEE introduced the following two definitions to describe software features [IEE90]:

Definition 1: "A distinguishing characteristic of a software item (for example, performance, portability, or functionality)."

Definition 2: "A software characteristic specified or implied by requirements documentation (for example, functionality, performance, attributes or design constraints)."

In both of the above definitions, a feature is described as a high-level requirement view without detailing any implementation issues.

On the other hand, in [Kan90], Kang et al. describe a feature as a prominent or distinctive user-visible aspect, quality, or characteristic of software system or systems. Features are often regarded as the attributes of a system that directly affect the end-users. The end-users have to make decisions regarding the availability of features in the system.

In [Eis01], Eisenbarth et al. provide the following feature definition:

"A feature f is a realized functional requirement (the term feature is intentionally defined weakly because its exact meaning depends on the specific context). Generally, the term feature also subsumes non-functional requirement."

However, in the context of their paper only functional features were relevant. Specifically, they [Eis01] considered feature as an observable result of value to a user.

Eisenbarth et al's definition introduces a categorization of features into functional and non-functional features.

1.1.1 Functional feature

Kang et al. [Kan90] defined functional features f_{funct} as services that are provided by the applications. In addition, according to them, features of this type can be found in the user manual and the requirements specification document. Based on the definition of feature [Eis01], f_{funct} can be described as a realized functional requirement implemented by the system. The IEEE [IEE90] described the term functional requirement as a system or software requirement that specifies a function that a system or software system or system component must be capable of performing. These are software requirements that define the behavior of the system. Specifically, functional requirements define the fundamental process or transformation that software and hardware components of the system perform on inputs to produce outputs.

1.1.2 Non-functional feature

A non-functional feature f_{nfunct} can be described as a realized non-functional requirement implemented by the system. In [Kul00], Kulak et al. described non-functional requirements as addressing the hidden areas of the system that are important to the user although the users may not apprehend it. They do not deal with the functionality of a system. Rather, they relate to the system's overall success. In all, non-functional requirements are the constraints, limitations, and specifications on performance. Examples of non-functional requirements are the ability of a software application to run on UNIX, or for a software system to work in real time etc.

Xavier Franch et al. [Fra98] defined f_{nfunct} as "any constraint referred to a subset of the non-functionality attributes that are in use in a particular software unit", where non-functionality attributes are defined as "any attribute of software which serves as a means to describe it and possibly to evaluate it. Among the most widely accepted, we can mention: time and space efficiency, reliability and usability."

1.2 Motivation

Program comprehension is an essential part of software evolution and software maintenance. A software system's code base that is not comprehensible cannot be changed. Programmers attempt to understand only how certain specific features are reflected in the code [Raj02, Eis01a]. The user views the features in terms of the functionality the system is performing, and the developer views the features in terms of the implementation of the feature [Tur99a]. This research investigates different techniques and approaches that can be applied to guide programmers during the comprehension process. The motivation of this research is two-fold. Firstly, existing feature extraction techniques and approaches are surveyed and categorized based on their underlying approaches and feature types extracted. Secondly, we present a semi automated feature extraction approach that utilizes source code analysis to identify features in the source code.

1.3 Thesis outline

The remainder of the thesis is as follows. Section 2 describes the background and a literature survey of existing methods of feature extraction relevant to this thesis. Section 3 provides the definition of features and introduces the techniques for extracting features in

this research. Section 4 presents the CONCEPT's system architecture, and discusses implementation issues, and presents a case study. Finally, section 5 provides conclusions and discusses some future work.

2 Background/Literature survey

Software modification starts with a maintenance request, which is usually expressed in terms of domain concepts or program features that have to be enhanced or changed. The majority of maintenance tasks involve perfective maintenance activities which are caused by changes in the functional requirements, corresponding to features in the software system [Boh96]. In order to add or modify any feature, the existing features need to be examined so that the changes do not create any undesirable effect in the system. One approach to comprehend these existing features in the system is to extract these functional features from the system to focus the comprehension and maintenance process on these parts.

2.1 Software Comprehension and reverse engineering

Software reverse engineering research is concerned with developing tools and methodologies to aid in the program understanding and management of the ever increasing number of legacy systems. According to Von Mayrhauser and Vans [May95], "program comprehension" or "program understanding" constitutes a process that uses existing knowledge to acquire new knowledge. The system requirements are likely to change while the system is being developed because the environment is changing. A change in a system to make it meet its requirements more effectively is referred to as perfective maintenance. Adaptive maintenance is used to change a system in order to meet new requirements. Finally, corrective maintenance is used when there is a need to change a system to correct deficiencies in the way it meets its requirements [Boh96]. Boehm [Boe81] described that the software development effort is largely devoted to

maintaining existing systems rather than developing new systems. The proportion of resources and time devoted to maintenance range from 50% to 80% [McC92].

As a result, for years researchers have tried to comprehend how programmers understand programs throughout software maintenance and evolution process [May95]. In addition, reverse engineering is concerned with the analysis of existing software systems to make them more understandable for maintenance, re-engineering, and evolution purposes [Mul94]. Chikofsky and Cross [Chi90] defined reverse engineering as "analyzing a subject system to identify its current components and their dependencies, and to extract and create system abstractions and design information." Current reverse engineering technology concentrates on retrieving information by using analysis tools, and by abstracting programs bottom-up by recognizing plans in the source code [Ric90, Ton96]. The main principle of such tools basically is to help maintainers to understand the program [Rug94]. According to Rugaber [Rug92, Rug95] the process of reverse engineering must focus on mapping the gap between bottom-up code analysis, and top-down synthesis of the description of the application, application domain and programming language etc. In addition, code analysis is intuitively a bottom-up exercise [Nel96]. However, the code does not contain all the information that is needed. It helps if knowledge about architecture and design tradeoffs exists. However, these are not available often [Mul00]. Hence, code analysis necessitates higher level meaning to be extracted from code fragments, and higher level concepts to be mapped to lower level implementations. According to Shneiderman [Shn80], programs are comprehended by bottom-up strategy which involves reading source code and then mentally chunking low-level software artifacts into meaningful, higher-level abstractions. These abstractions are

further grouped until a high-level understanding of the program is formed. Next, in top down strategy [Bro83], programs are comprehended by reconstructing knowledge about the application domain and mapping that to the source code. In all, top down strategy includes formulating hypotheses and confirming them by examining the program.

Reverse engineering by itself involves only analysis, not change to the system. Reverse engineering is the basis for the following activities (listed based on their level of impact). The activities include re-documentation, design recovery, restructuring and reengineering [Chi90]. Re-documentation, or recreation of documentation, means revision of system documentation at the same level of abstraction. Design recovery is mainly used when there is a need for perfective maintenance. In this phase, re-documentation is used with the aid of domain knowledge and other external information where possible to create a model of the system at a higher level of abstraction. Restructuring is used when preventive maintenance is needed. It includes lateral transformation of the system within the same level of abstraction. Reengineering involves a combination of reverse engineering for comprehension, and a reapplication of forward engineering to reexamine which functionalities need to be retained, deleted or added [Nel96].

2.2 Documentation versus source code-driven feature extraction

Features in terms of system functionality can be extracted from the engineering-based requirements document, which usually provides the description of the requirement, design and architectural details of the life cycle of a software development. In order to extract the code fragments that are associated with a feature, source code analysis is needed. There are several techniques for source code-driven feature extraction. As a

result, feature extraction can be categorized as source code and non-source code based feature extraction.

There are two major approaches to extract functional features from a system.

(1) Features can be extracted from the documentation of the software such as the requirements document or user manuals when features are viewed according to the problem domain.

(2) Features can be extracted and reverse engineered from the source code by identifying which program artifacts correspond to the implementation of a functional requirement of the system.

2.2.1 Non-source code based

As the need arises to identify those parts of a system that are crucial for the programmer and maintainer to understand, a possible solution is, if valid and complete documentation exists, to read the documentation. Good sources for analyzing the main functionality of the software system are requirements specification documents, user manuals, white papers etc. In what follows we discuss the major techniques and approaches relevant to feature analysis and extraction based on non-source code based sources.

Features Analysis based on requirements specification

When software developers are concentrated on the problem domain, they tend to look for information related to a particular function or feature in the system. A requirements specification document states the functional and non-functional requirements of the system which serves as a baseline for the developer to implement the system [Kir97]. In [Dav82] Davis identified features as a key organization mechanism for requirements

specification. Software requirements specification is part of the first phase of system development which includes preparing a complete description of the system's external behavior. It is a fundamental stage of system development, since specification defects will become increasingly difficult to repair when the system is proceeding to the subsequent stage of its life cycle [Dav93]. The concrete result of requirements specification is the SRS Software Requirement Specification [Kir97]. The requirements specification ideally captures all the important behavioral characteristics of a software system. Hence, According to Turner [Tur99a], feature can be viewed as a grouping or modularization of individual requirements within that specification during the analysis of the requirements specification document. From a programmer perspective, a feature is an abstract description of a functionality described in detail in the specification [Won99].

Use cases

A use case is defined as "a sequence of transactions performed by a system, which yields an observable result of value for a particular actor"[jac97]. Use cases and scenarios are very common approaches used in the requirements and specification phase. They capture most of the requirements, which include all functional requirements and also non-functional requirements such as response times, performance, etc. A use case is a high-level description of how the software will be used. It identifies a software user or an actor and how the user interacts with the system. Hence, a single use case describes a subset of a system's functionality in terms of the interactions between the system and a set of users or actors. It specifies the intended behavior of a system. It is initiated by a particular user, and serves the purpose of delivering some meaningful unit of work, service, or value to

the initiator. When capturing requirements, use case views the system as a black box [But97]. They are suitable for defining functional requirements in the early stages of system development when the inner structure of the system has not been defined. Also, they can be used as a basis for defining this structure in terms of classes, packages, etc., and can be used for defining test cases. Since use cases do not deal with the mechanics inside the system but focus on how the system is perceived from the outside, they are the most useful approach in discussions with end users to make sure that the requirement of the system will meet the end users demand [Li01]

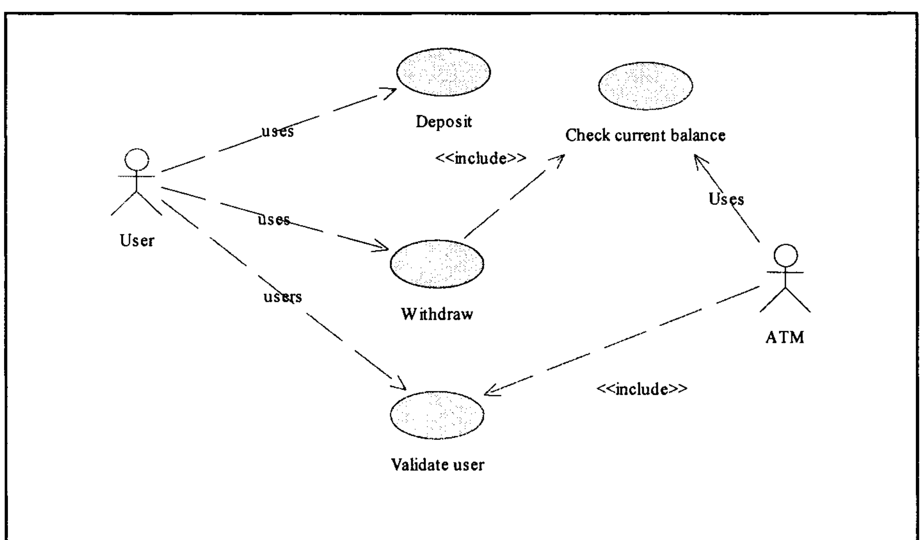


Figure 1 Use case diagram of an automated banking system

Use cases have quickly become a widespread practice for capturing functional requirements. This is especially true in the object-oriented community where they originated. However, their applicability is not limited to object-oriented systems [Mal99]. An example of a use case diagram is given in **Figure 1**.

If proper naming conventions are used for the use cases, programmers can obtain a general idea about the system and its functionality. Checking current balance is used by

the ATM system when withdraw use case is executed to verify whether the current balance shows there are sufficient funds to withdraw the given amount of money.

Each use case can occur under different situations called "scenarios". For example, a customer withdrawing money from the ATM machine can have the following scenarios:

- Customer requests $300 to withdraw from the account. The current balance is verified by the ATM system, and the user has a balance of $400. Hence, this amount will be withdrawn with a receipt from the ATM system.
- Customer requests $300 from checking account, but he has only $200 in his account balance. The ATM system will inform customer that he has "insufficient funds."

User manuals

User manuals describe how a user interacts with the application. Traditionally, functional features are described as the services that are provided by the software application. Features of this type can be found in user manuals. Operational features are described as the features that are related to the operation of applications from the user's perspective; that is, how user interactions with the applications occur. Hence, user manuals are a good source for identifying operational features as they contain a detailed description of the user interaction with the application [Kan90]. The step-by-step information described in the user manual provides some background on the application domain, and therefore the extraction of the feature.

Use Case Maps

Use Case Maps (UCMs), as proposed by R.J.A Buhr [Buh96, Buh98], are a scenario-based notation for describing the organizational structure of complex systems and their evolving behavior in an abstract way. It bridges the modeling gap between use cases or requirements and detailed design, and aids in visualizing the architectural entities of an application. In [Amy00a], UCMs were proposed as a notation for describing features. They showed related use cases in a map-like diagram, and captures functional requirements in terms of cause and effect relationship scenarios of the abstract components. [Amy00a]. In all, UCMs are used for capturing requirements, evaluating architectures, validating and detecting feature interaction. They illustrate reactive or distributed systems in terms of casual paths that are followed through the optional components caused by the occurrence of stimuli. When the UCMs illustrates the components, they are referred to as bound, and if the components are not shown in the diagram, then the UCMs can be referred to as unbound.

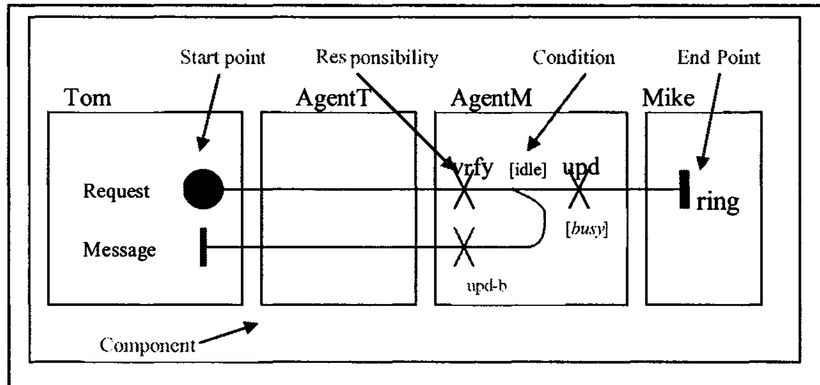


Figure 2 Diagram of a Use Case Map representation [Amy00b]

Components shown in UCMs can represent software entities such as objects, databases, functional entities, network entities, etc. as well as non-software entities such as users, actors, processors etc. UCMs involve concurrency and partial orderings of activities, and

they link causes such as, preconditions and triggering events to effects such as post conditions and resulting events composed of responsibilities [Amy02]. A short example of a simple bound UCMès description, taken from [Amy00b] is described below and the diagram is shown in **Figure 2**.

Figure 2 shows a UCM where a user called "Tom" is trying to set up a telephone call connecting with another user "Mike" through some network of agents. Tom and Mike each have an agent responsible for managing subscribed telephone features such as Outgoing Call Screening. At first, Tom sends a connection request (request) to the network through his agent AgentT. This request causes the called agent to verify (vrfy) whether Mike's telephone line is idle or busy. If Mike's phone is idle, then there will be some status update (upd) and a ring signal will be activated on Mike's side (ring). Otherwise, a different update will occur (upd-b) and an appropriate message (stating that Mike is not available) will be prepared and sent back to Tom (message) [Amy00b].

Domain analysis

The development and maintenance of large and complex software systems require a clear understanding of the desired system features. Domain analysis is a process for understanding requirements in a particular problem domain. It helps in understanding program features by clearly defining the features and capabilities common to systems in this application domain before implementing the system. As described in [Kan90] "domain analysis is the systematic exploration of software systems that define and develop commonality, defines the features and capabilities of a class of related software systems". R. Pietro-Diaz [Pie90] defined domain analysis as "a process by which

information used in developing software system is identified, captured, and organized with the purpose of making it reusable when creating new systems." Domain analysis approach can support a mapping from the problem space to appropriate objects and classes, while considering the design context for patterns and frameworks. Domain analysis creates a domain model which captures the essential entities in a domain and the relationships among these entities. Several domain analysis methods exist, including feature oriented domain analysis (FODA) [Tur99a]. According to [Coh98, Doi98, Gri98], "a feature represents one or more domain requirements, and this feature analysis becomes an important aspect of domain analysis." Domain products, representing the common functionality and architecture of applications in a domain, are produced from domain analysis. The FODA method focuses on identifying factors that can cause differences among applications in a domain, both at the functional and the architectural level. In addition, this method uses those identified factors to parameterize domain products.

In [Tur99a], the term feature is referred to as the capabilities of systems in a domain. They typically seek to distinguish the features that represent basic, core functionality from those that represent variant, optional functionality [Tur99b]. Domain analysis processes existing and potential software applications in order to extract and pack reusable assets [Suc00]. The feature/contextual view of many domain analysis methods should become an essential part of object technology for reuse. Described below, the feature oriented domain analysis (FODA) method establishes three phases of a domain analysis [Kan90], as illustrated in **Figure 3** [Kru93].

Context analysis: The context analysis phase provides the context model, which is used to define or establish the scope or bounds of a domain analysis. A context model is

represented with a structure diagram and context diagram where structure diagram includes informal block diagrams. The context diagram is represented with data flow diagrams illustrating data flows between a generalized application within the domain and the other entities and abstractions with which it communicates [Kru93]. The domain analyst interacts with users and domain experts to establish the bounds of the domain and establish a proper scope for the analysis. The analyst also gathers sources of information for performing the analysis.

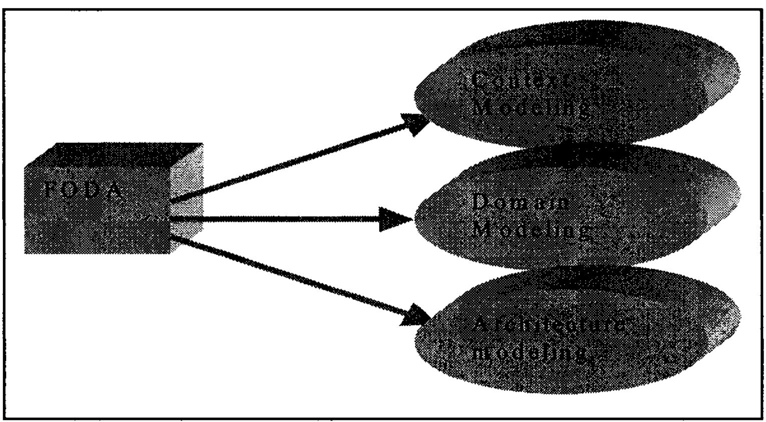


Figure 3 The phases of feature oriented domain analysis [Kru93]

Domain modeling: Domain modeling provides a description of the problem space in the domain that is addressed by software. The FODA domain modeling process includes three models including the feature model, entity relationship model and functional model. The feature model captures common features and differences in the applications in a domain. The functional model provides a behavioral and functional view of the system.

Architecture modeling: Architecture modeling is used to create the software architecture for implementing solutions to the problems in the domain. FODA architecture modeling provides a high level design of the applications which aids in domain product reuse. To

achieve successful software reuse, commonalities of related systems must be discovered and represented in a form that can be exploited in developing similar systems [Kru93, Kan90]. In [Kan98], Kang et al. extended the feature-oriented domain analysis to feature-oriented reuse method (FORM). This is a systematic method that focuses on capturing commonalities and differences of applications in a domain in terms of "features" and using the analysis results to develop domain architecture and components.

2.2.2 Feature extraction based on source code

For developers to comprehend a software system, it would be advantageous to have some domain knowledge and proper documentation of the program to be able to identify the basic entities and functionalities of the system. However, documentation might often be unavailable or out-of-date and domain experts may not be available. The original developer's memory about the source code of the program fades. Some programmers who were involved in the early development of the application might eventually leave. The complexity of the software increases as modifications are applied in the source code base [Leh80]. Typically, a programmer tries to understand how a requirement is implemented in the source code by first identifying the requirement, and then trying to map the source code that agrees with the implementation of the feature. The remainder of this section presents different approaches to comprehend source code based on feature extraction.

Concept analysis

Concept analysis is a mathematical technique (the mathematical foundation was laid by Birkoff in 1940) which provides a way to identify the grouping of objects that have

common attributes. Birkoff [Bir40] proved that for every binary relation between certain objects and attributes, a lattice could be constructed that provides remarkable insight into the structure of the original relation. Concept analysis starts with a relation, or boolean table, T between a set of objects O and a set of Attributes A, where relation R could be defined as a binary relation between O and A. A formal context C can be defined as C=(O, A, R), where R is a binary relation between O and A. Concept analysis has been used to evaluate class hierarchies, to identify modules, to recover components, to derive feature component map and to identify feature component relationships [Sne94, Sne97, Sti97].

A concept lattice can be used to identify possible ways of partitioning the program into modules [Sti97]. Concept analysis has been used for analyzing feature relationships with the components or the feature implementation in a system. The binary relation of concept analysis has been used to derive the feature component map that states which components are required when a feature is invoked.

In order to derive the feature component map via concept analysis, one has to define the formal context (objects, attributes, relation) and to interpret the resulting concept lattice accordingly [Eis01b]. Eisenbarth et al. [Eis01a] used a combination of static and dynamic analyses to localize a feature. Concept analysis was used to derive correspondences between features and components implementing a specific set of related features. The goal of this dynamic analysis was to find out which subprograms contributed to a given set of features. For each feature, a scenario corresponding to a sequence of user inputs triggering system actions was executed. The process was automated to a great extent. The general process [Eis01a] can be described as follows:

- Identify the set of relevant features $F= \{f1, f2, f3......Fn\}$ where feature corresponds to functional requirements. Identify scenarios $A = \{S1, S2,..Sq\}$ which should cover all the identified features in F.
- Next, execution summaries are generated where all required subprograms $O=\{s1...sp\}$ for each scenario are produced. In addition, a subprogram is a lowest level of components in the program, and it is a function or procedure according to the programming language.
- In the next step, the relation table R such that $(S1,s1), (S1,s2),...,(Sq,sp) \varepsilon R$ are created.
- Perform concept analysis for (O, A, R) where concept analysis is used to derive the detailed relationships between features and executed subprograms.
- Identify relationships between scenarios and subprograms.
- Perform static dependency analysis along the static dependency graph (SDG) in order to narrow the executed subprograms to those that form self-contained and understandable feature-specific components.

In [Eis01b], a technique was derived to get the feature-component correspondence to utilize dynamic information and concept analysis.

Test cases

Test cases can be used to identify and localize a system's features. The test cases represent knowledge of the feature requirements to ensure that the feature implementation conforms to the desired requirements [Tur99a]. Especially, test cases are executed to see what components are executed with the given test case. The existence of

test cases allows checking the complete scope of system requirements collected by the individual features.

Regression test cases have been used to identify features with the intention of evaluating software. They are full of information about system features. Regression testing is defined as selective retesting of a system or component to verify that modifications have not caused undesired effects and that the system or component still complies with its specified requirements [IEE90].

Before a new version of a software product is released, the old test cases are run against the new version to make sure that all the old capabilities still work. The reason they might not work is because extension or modification of the new code to a program can easily introduce errors into code that should not be changed. By exercising each feature with their associated test cases using code profilers and similar tools, code can be located and refactored to create components. The steps in identifying the feature and creating a component that maps to the feature have to start with identifying the source code associated with features that need evolution. The next step is to create components based on the extracted code [Meh02]. In [Meh01a], Mehta describes how information about the legacy system features can be attained. His team identified features of the legacy systems using test cases by analyzing the existing regression test cases and interviewing the software developers and the users of the system. Test cases tell a legacy system's story which can be used to identify features that the end users are most interested in. After collecting the test cases, procedures are developed to identify the code associated with that feature(s). Next, in order to create component(s), the extracted code is used. Finally, for validation purposes, the components that map to a feature are inserted back into the

legacy system. This process bridges the gap between the problem domains, where the users are more concerned with the functionality of the system, to the solution domain, which includes the software components that developers see [Meh01a]. The following technique, called Software Reconnaissance [Wil95], uses test cases to extract features.

Software reconnaissance

Software reconnaissance is a dynamic analysis method described by Norman and Scully in [Wil95]. The code that implements features can often be found by executing the program twice: once with the feature and once without. Meanwhile, the parts of the program that were executed the first time but not the second time are marked. These parts are likely to be in or near code that implements the feature.

Software reconnaissance uses test cases as probes to locate code for a particular product feature. The program is first instrumented in much the same way that programs are instrumented to determine test coverage. Then it is run with a few test cases that exhibit the desired feature and with a few others that do not. The executions of the instrumented code produces trace files showing which code components were used in each test [Whi01].

2.3 Source code analysis based

Program slicing [Wei84] is a well known technique used for analysis. Hence, in this section we will provide a general introduction of program slicing and its category. Next, some background information about slicing based feature extraction techniques will be described.

Program Dependence Graph

A Program dependence graph (PDG) is a language independent program representation directed graph, which together with operations such as program slicing, can form the basis for powerful programming tools. It can aid in understanding programs, analyzing and localizing features and debugging [Hor92].

Kunrong Chen, el al [Che00] used the dependence graph for localizing features. Ottenstein and Ottenstein [Ott84] introduced procedure dependence graph (PDG), which is a graph representation of a procedure where vertices are connected by data and control edges. The vertices represent statements such as assignment statements, input/output statements or regions of the code. The data dependence edges represent data flow in a procedure. Control dependence edges represent conditions on which a statement or region depends [Hor92]. In addition, each procedure has a special entry vertex that corresponds to the entry point of the procedure [Che00]. The System dependence graph (SDG) contains the collection of procedure dependence graphs (PDG) with additional vertices and edges to represent procedure calls, rather than just single procedure programs. A single system consists of a main procedure and a collection of auxiliary procedures. The assumption in that case is system developers usually do not need access to the statement level information when trying to locate feature.

In [Che00], an abstract system dependence graph (ASDG) representing a higher level of abstraction of the program is proposed. It can be constructed using a subset of the information of the SDG. The algorithms used to construct SDG can be used to construct ASDG also. In the C language, an ASDG has almost the same representation of SDG except instead of representing statements as vertices; ASDG consists of vertices that

represent components which include functions and global variables. Call edge represents function call, and data flow edge represents flow of data from a function to a global variable. Next, search scenarios are investigated for locating features. Localization of feature using ASDG starts with the search through the code. In each step of the search, a single component is chosen for visit. All visited components and their neighbors constitute a search graph. The search follows control flow and data flow dependencies among the program components. At the beginning the search graph contains only the starting component. Each visit to a component expands the search graph, and the process continues until all the components implementing the feature or concept are located. Top-down, Bottom-up, backward data flow, and Forward data flow strategies are available for search graph. For instance, in the top-down strategy, the functionality of the whole program is summarized in the top-most function main() or top class of the program. If the top class or main () method does not implement the sought feature, the features should be implemented by any of the called function or classes. Hence, the searching has to continue for the feature, and moving down through the call graph towards more and more specialized functions or classes, the programmer ultimately finds the classes or functions that participate in the concept. If the origin or destination of data is sought, then the programmer follows the data flows rather than control flows [Che01].

Program Slicing

Program slicing is a program reduction technique originated by Mark Weiser [Wei79,Wei84] that determines the statements S which are relevant for a particular computation obtained by deleting statements that are not relevant for that particular

computation in program P. Program slicing represents the same behavioral representation of the original program and the slice consists of all statements in the program P which may affect the value of variable V at the instruction point I based on the data flow and control flow information of the program. Hence, the slice is constructed based on the slicing criterion $C = <I, V>$ composed of the program location and the variable respectively. Program slicing has several application areas mainly in software engineering and development such as, debugging [Wei84, Del01, Kor88b], software maintenance and program comprehension [Gal89, Tip95, Har01], testing [Gop91, Gup92, Har95], ripple effect analysis [Wan96], restructuring [Tip95], etc. For instance, when the programmers debug their program, they try to focus on the section of the code that might cause the bug in the application. It is unnecessary for the programmer to analyze the sections of the source code which are irrelevant to influence the fault. In all, slicing focuses attention on those parts of the program that may contain the fault and removes the unnecessary information for that particular computation in the program slicing. Next, since slicing transforms a larger program to a smaller one, it reduces the load a programmer has to take into memory in order to understand a program. Next, Slicing can be divided into several categories such as the following:

Static slicing

Based on the original definition of Weiser [Wei84] a static program slice S, which is also referred to as static backward slice, consists of all statements in program P that may affect the value of variable V at some point P. The slice is defined for a slicing criterion $C=(x, V)$, where x is a statement in program P and V is a subset of variables in P. Given C,

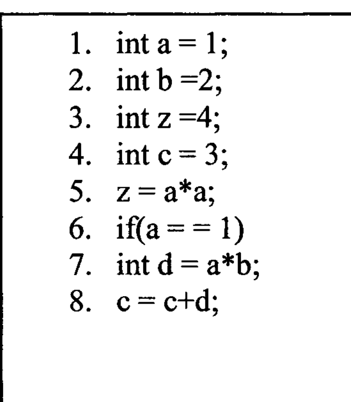
the slice consists of all statements in P that potentially affect variables in V at position x. A static slice includes all the statements that affect variable V for a set of all possible inputs at the point of interest according to data and control dependencies. Static slices which include only statically available information are computed by finding consecutive sets of indirectly relevant statements. Several researchers tried to extend the original static slicing algorithm, as defined, for example, in [Agr94, Cho94] who computed backward static slices based in the presence of arbitrary control flow. [Liv94, Lyl93] have proposed algorithms that compute static slicing in the presence of pointers.

Dynamic slicing

A dynamic program slice is that part of a program that "affects" the computation of a variable of interest during program execution on a specific program input. The concept of dynamic program slicing was presented for the first time by Korel and Lasky [Kor88] in 1988. Dynamic program slicing refers to a collection of program slicing methods that are based on program execution. This may significantly reduce the size of program slices since run-time information, collected during program execution, is used to compute them. Dynamic program slicing was originally proposed only for program debugging, but its application has been extended to program comprehension, software testing, and software maintenance. Different types of dynamic program slices, together with algorithms to compute them, have been proposed in the literature (examples include, [Agr90, Gop91, Kor94, Tib99, and Son99]). Slicing can further be divided into two categories, forward slicing and backward slicing.

Backward slicing

Backward slicing starts with a slicing criteria specified as a tuple $C = <i,v>$, where i is a program statement and v is a set of variables. It produces a program slice, which is a set of statements that might affect the values of variables in the set v at statement i. The variables in v are restricted to be either defined or used in the statement i. A slice under the above definition is also known as a backward program slice [Wei79]. Slicing is represented by a program dependency graph (PDG), where the PDG shows the data and control dependencies by backward traversing the execution trace.



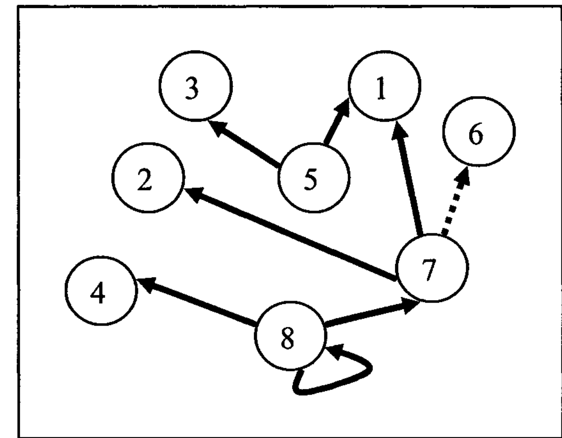


Figure 4 An example of source code and the PDG representation of the source code

1. int a = 1;	1. int a = 1;
2. int b = 2;	2. int b = 2;
3. int z = 4;	3.
4. int c = 3;	4. int c = 3;
5. z = a*a;	5.
6. if (a = =1)	6. if (a = =1)
7. int d = a*b;	7. d = a*b;
8. c = c + d;	8. c = c + d;

Figure 5 The computed slice from Figure 4

Figure 4 provides both a sample source code and the program dependency graph computed by traversing the program backward using the backward slicing algorithm. The solid arrows represent data dependencies and the dotted arrow indicates control dependencies in the program In **Figure 5**, a slice is computed for variable c in statement 8 based on the program dependency graph of **Figure 4**.

Initially, statement 8 has data dependency for variable c with itself as the value of c is used and defined at the same statement. Also, it has a data dependency with statement 4 for variable c as c is defined in this statement. For variable d, statement 8 has a data dependency with statement 7 as statement 7 includes the last definition of d. Next, statement 7 has a control dependency with statement 6.

Subsequently, statement 7 has data dependencies for a, b with statement number 1 and statement number 2 as the last definitions of a and b are provided at statements number 1 and 2 respectively. Statement number 5 has a data dependency with statement 1 for variable a and with statement 3 for the definition of z. However, statements 3 and 5 are not part of the slice because these statements do not affect the computation of c at statement 8.

Forward slicing

Most of the existing slicing algorithms are based on "backward" analysis, either traversing the program dependency graph backward (static slicing) or tracing the execution trace backward to derive dynamic dependence relations (dynamic slicing). The notion of forward slice computation has two different interpretations in the literature. Korel [Kor94] introduced a dynamic forward slicing algorithm that is based on dynamic

forward analysis. In fact, in this algorithm dynamic slices are computed during program execution (at run-time) and no major recording of the execution trace is required. The forward algorithm computes slices for all variables defined during the program execution. The major advantage of the dynamic forward approach is that its space complexity is bounded, as opposed to the dynamic backward slicing approaches. Additionally, it computes slices for all variables up to a point of interest. This forward algorithm will result in the same slice for a variable of interest as the dynamic backward algorithms.

Horwitz [Hor90] provided a different interpretation of forward slicing. Horwitz defined a forward slicing as these program elements that are potentially affected by the values of the variables in *V* at statement *i* [Hor90]. Korel's dynamic forward slicing algorithm on the other hand identifies all the statements that have relevant control/data dependencies influencing the particular execution at the particular position.

```
1. b = 2;
2. a = 1;
3. c= a*a;
4. d=a*b;
5. c = c+d;
```

Figure 6 A sample source code

	Node ↓ Slice →	A	b	c	d
0^0	At the beginning	{}	{}	{}	{}
1^1	b=2;	{}	{1}	{}	{}
2^2	a=1;	{2}	{1}	{}	{}
3^3	**c=a*a;**	{2}	{1}	**{2,3}**	{}

Figure 7 Forward slicing computation of Figure 6 using Korel's algorithm

Horwitz's forward slicing approach has application in the area of impact analysis by identifying all statements that are potentially affected by the slicing criterion in future executions. On the other hand, Korel's forward slice has applications in testing and debugging.

Figure 6 shows a simple source code of a program, and **Figure 7** shows the slicing computation in Korel's approach for computing slice using dynamic forward algorithm. **Figure 7** illustrates that for slicing criteria <3, c>, the slice is computed for all the variables at the same time. For variable c at statement 3, the slice is 2, 3.

Figure 8 shows the dynamic slice computed using Horwitz's forward slicing approach for the sample source code of **Figure 6**. As we can see, the forward slicing computation approach by Horwitz takes the statements that are affected by the value of c in statement number 3, and the slice is computed for only variable c.

```
3. c = a*a;
5. c = c+d;
```

Figure 8 Computed slice using Horwitz approach for slicing criterion <3,c>

Concept assignment and slicing

Having automated techniques for extracting subcomponents according to high level criterion is advantageous for program comprehension and reverse engineering. The combination of program slicing and concept assignment are automated source code extraction techniques that use a criterion and program source code as input and generates parts of the program's source code as output [Har02]. Mark Harman et al. in [Har02] showed how slicing and concept assignment could be combined to perform unified source code extraction, which extracts code identified by a concept assignment criterion.

They used several algorithms, such as executable concept slicing, key statement analysis and concept dependence analysis. Concept assignment is defined by Biggerstaff et al. as "a process of recognizing concepts within a computer program and building up an understanding of the program by relating recognized concepts to portions of the program, in operational context and to one another" [Big93]. This approach has the advantage that the code extracted is executable, and that the criterion for extraction is expressed at a high level in terms of domain specific concepts. This is in contrast to pure slicing which extracts subprograms based upon a low level criteria set of variables.

2.4 Discussion and limitation of existing feature extraction methods

Feature extraction and localization are a crucial point in program understanding. The current methods for extracting and localizing features have some potential drawbacks. The limitations of the existing feature extraction approaches can be summarized as follows:

Feature extraction methods in general might be expensive if they can not be automated. This is because they involve often significant human interaction. Another factor is that most of the techniques for feature analysis are based on some form of domain analysis, or test cases, and that these resources might not be readily available. In what follows, we discuss the limitations of the different categories of feature extraction techniques.

Document-driven approach

Challenges for the document-driven approach are that it has to deal with inadequate and incomplete documentation of a system. The documents might not conform to the existing

application because as requirements evolve, documents sometimes may not be modified to reflect the latest system changes. In addition, requirements specification documents might not be available for the application system. Even if these documents exist, they might not be clear [Won99]. Documents are reflecting requirement from a user's point of view; however, they may not reflect the implementation of the requirements.

Use cases are frequently used to describe functional system features at the requirement or specification level. The use case representation corresponds to a collection of intended uses for a proposed system. However, there are differences between use cases and features. The link between features and implementation of the system requirements are more direct than with use cases. With use cases, the domain model objects have associations with the solution domain artifacts only when the solution domain shares the object model of the problem domain. A large application can require very large numbers of use cases for their descriptions. Moreover, when describing many extensions, alternatives, and options, the domain engineers can easily lose their way when constructing new systems [Gri98].

Domain analysis approaches to feature extraction can provide guidance during the comprehension of the overall system functionality; however, they do not provide much support for the automatic or semi-automatic extraction and analysis of features at the source code level. For the domain analysis, application domain knowledge is modeled independently of systems to support the forward engineering of product families. Evolution focused solely on the problem domain may suggest changes that degrade the structures of the original code [Idr00]. Domain analysis depends on the availability of domain experts and the designers involved in the domain modeling. However these might

not be readily or no longer be available. **Table 1** summarizes the limitations of the document bases techniques of feature extraction.

Criteria / Technique	Availability Of document	Relates to the systems implementation or to the solution domain	Traceability and accuracy
Use cases	Not always available	Mostly related to the problem domain, and is used for requirements gathering	An use case diagram might not have been updated to reflect any requirement changes.
Use Case Maps	Only very limited availability, since it is a more recent approach and not officially part of UML	May not have the same implementation as described in the Use Case Maps	Depends on the maintenance of the use case maps; might not reflect the latest requirements and features implemented.
Requirement and specification	Availability might be limited and programmers might be reluctant to read it	Problem domain oriented and most of the time does not conform to the solution domain requirements	Sometimes the documents are not adequate
Domain analysis	Domain models are often not available	Might not be related to the system implementation	Domain model and source code might be disjunctive

Table 1 Disadvantages of Feature extraction based on the document based technique

When extracting features using dependence graphs, there is a need to integrate these with other dependency graph analysis tools, to analyze the code and extract the associated ASDG (Abstract syntax dependency graph). Even though interfaces exist to facilitate the navigation through an ASDG, the analyst has to perform the localization and identification of the feature implementations manually [Che01].

When extracting feature using regression test cases, the assumption is that valid test cases are readily available. These test cases might be undocumented, not complete or might not even reflect the current implementation (depending on the quality of the test and how these were maintained). In addition, it is not often possible to identify what group of test cases will exercise a given feature.

Software Reconnaissance [Wil95] does not locate all the code corresponding to features. It deals with one feature at a time and gives little insight into connections between a set of related features. If a set of related features rather than a single feature is to be considered, one would repeat the analysis using each feature separately and then specifically required subprograms. However, even the relationships among pairs of features cannot be identified.

Slicing can be used to compute, identify, and extract program features. However, both the static and dynamic slices have some drawbacks. A static slice is less effective in identifying code that is uniquely related to a given feature because, in general, it includes a larger portion of program code with a great deal of common utility code. On the other hand, collecting dynamic slices may consume excessive time and file space. Also, most of the existing slicing algorithms are still limited in their support of object-oriented programming languages.

Thomas Eisenbarth et al. [Eis01a] combine static and dynamic analysis with concept analysis to derive correspondences between features and computational units. Concept analysis additionally yields the computational unit jointly required for a set of features. Here, in order to do the concept analysis, scenarios which represent a feature need to be provided. In order to create a scenario or test cases, domain knowledge is needed. However, this method is not helpful for extracting the feature without any test cases or domain knowledge. **Table 2** summarizes the limitations of the source code analysis techniques.

Category	Technique	Limitations of the current technique
Concept Analysis	A mathematical foundation technique for obtaining binary relations between features and components	– Scenarios or test cases are needed in order to find the components that correspond to a particular feature – The accuracy of the feature depends on how adequate the scenario is – Domain knowledge is needed to create a scenario, however, domain knowledge might not be available
Test Cases	A dynamic analysis method that uses test cases to execute and find the components that are executed based on the given feature	– Valid test cases are required – It is not always possible to know what group of test cases will exercise a given feature – After enhancing any features in the system, other test cases are needed to verify that the old features do not have any undesired effects because of the modification – Relationships among features are not identified
Static Slicing	A program analysis technique that uses only parsing information to compute a slice	– The extracted feature based on static slicing could be too large
Dynamic Slicing	A program slice is taken based on a particular input	– May consume excessive time and space
Program Dependence Graph	Based on slicing, but has some extension as the whole system is represented in an abstract system dependency graph (ASDG) and the programmer has to search through the graph to identify which one is the feature they are looking for	– An automated tool is required to represent the System dependence graph and search graph – The technique needs some human intervention as the analyst needs to start searching the graph for a specific feature

Table 2 The disadvantages of feature extraction from source code

2.5 Existing tools

This section describes different methods for feature extraction that can be used in localizing features. Since a particular feature can be implemented in different ways, it is very difficult, if not impossible, to implement a fully automatic feature locator. However, currently several tools exist that aid in the process of extracting feature. Following is a survey of some of the existing tools:

grep

The maintainer often looks for the implementation of feature location manually with the help of simple tools such as grep. The most widely used technique is based on string pattern matching and uses the similarity of identifiers to program concept names [Che00]. For example, when searching for the location of copy and paste, the programmer may want to search for identifiers "paste", "copyPaste", "copy", "xcopy" and so on. When the appropriate identifier is found, the programmer studies the surrounding code to decide whether this is truly the location that implements the feature, or whether the similarity of names is just a coincidental correspondence. Also, the full extent of the concept's location must be established. Generally the feature is implemented not only in the place where the identifiers were found. It could also spread over the system. The maintainer starts software modifications after concepts are located and conceptual dependencies are established.

RIPPLES

A tool RIPPLES [Che01] was developed for maintaining software when features need to be changed or modified. Usually software changes require identifying features of the system by mapping the features from the problem domain to the software components. Software change also requires keeping track of the change propagation. The RIPPLES tool supports both concept location and propagation and combines automatic static code analysis with human intelligence. Change propagation process can be described as follows:

When any software entities are being changed, the changed component may affect other components of the system. In fact, modification of a component may cause the system to be temporarily inconsistent, as the requirements provide relationships between the change component and its neighbors which are no longer valid. To fix this, secondary changes are introduced in the neighbors, but they may cause new inconsistencies. This situation continues until the system becomes consistent again. This process is called change propagation [Raj97].

Two components in a conceptual dependence have a connection that is not discovered by static code analysis, yet both participate in the same concept. A change in one may require a change in the other. Conceptual dependencies have to be discovered manually during location and added to the ASDG. The abstract system dependence graph (ASDG) is the basic data structure of tool RIPPLES. ASDG represents dependencies among software components. For C programming language, the vertexes are functions, function arguments, and global variables and types. The edges represent data flows; control flows, and defines use relationships. ASDG is derived from a finer granularity system

dependence graph. ASDG [Che00] represents dependencies among software components. Location is a computer-assisted search approach. It is a step-by-step process where in each step; one component is selected for investigation. The mark on the edge means the direction of the change propagation as forward or backward. The program analyzer parses the code, constructs ASDG and stores it in the database. Location operations include mark, unmark and locate. Propagation is supported by change, add component, delete component and skip. During propagation, ASDG changes are called conceptual dependencies. It is the task of the programmer, not of the tool, to acquire all necessary domain and programming knowledge. The programmers make all decisions. They choose the starting component, make changes, determine the end of the process, etc.

xSuds

The xSuds methodology described in [Meh02], suggests using a code-profiling tool in order to extract feature. The most closely related technology is the xSuds tool that can identify program feature in a C program. xSuds tool suite, as part of Bellcore telecommunications system, is a software understanding and diagnosis system which helps to map features from the problem domain to the solution domain. xSuds first creates a representation of the program's control graph, thus laying out its structure. Next, each time the maintainer runs various test cases, xSuds stores the execution trace which records how much time each test has exercised a software component. Effective use of xSuds requires only that the maintainer has a basic understanding of the program's features and can identify the test cases of each feature.

xVue is one of the xSuds tools that help maintainers to locate features and identify feature interactions. xVue uses heuristics involving the control graph, execution trace, and the maintainer's knowledge to help locate features and identify feature interactions. xVue software developers locate code within a large system that specifically implements a particular feature.

A software system may provide its users with many different but related features. xVue helps maintainers locate code that implements a particular feature by using trace data to map features onto the code components that implement them. xVue uses one of several heuristics to identify feature-related code. The simplest and frequently most effective heuristic identifies blocks that the first test case executes but the second does not. Studying differences in execution between similar test cases often provide surprising insights into unfamiliar code. Although the accuracy of identifying feature code depends strongly on how adequately the test cases define the feature, tools like xVue, with relatively simple heuristics, effectively focus attention on relevant source code. xVue quickly narrows the search space by highlighting a few lines of code associated with a feature [Agr98].

Bauhaus tool

The Bauhaus project [Kos04], developed at the University at Stuttgart, performs research on techniques to support program understanding of legacy code. More specifically, it performs research on the recovery of the system's architecture that consists of its components, connectors, and constraints. Information about the system is exclusively extracted from the source code in a semi-automatic way that actively involves the user

with one of these environments. Czeranski et al. [Cze00] used the Bauhaus tools to analyze the xfig program. xfig's (an open source drawing program that runs on a variety of UNIX platforms) architecture was recovered and the entire maintenance task described in the handbook of the developer of Bauhaus tool was performed. Using Bauhaus tool information about the system is exclusively extracted from the source code as this is the only reliable source of information, in a semi-automatic way that involves the Bauhaus user (presumably a software maintainer or auditor). Information extracted from the source code is represented in a resource graph (RG), which abstracts global information such as call, types, and use relations. The RG information can be described by an entity relationship model. The entities are programming language constructs such as functions, types, and variables, and abstract analyses such as abstract data types, components, and subsystems. Examples of relationships range from information that can be directly extracted from the source code such as function calls to more abstract concepts. Entities are represented as nodes and relationships between the entities are represented as edges [Cze00].

Recon2

Wilde et al. [Wil95] developed a program feature location technology called Software Reconnaissance. The technology is based on the analysis of test cases. The instrumented program is tested with two sets of test cases: one set of test cases with the feature, and the other set without. For C software, students at the University of West Florida have developed a public domain reconnaissance tool called Recon2. The feature location in Recon2 [Wil02] is performed by analyzing the two sets of event traces. In a large and

frequently modified system, the code for a feature is often not contiguous or located in obvious places. The results depend both on the user's ability to find good test cases and on the way the original designer may have combined features in the code. Recon2 does not necessarily find all the code related to a particular feature, but it usually finds good starting points for a search.

2.6 Limitations of feature extraction tools

All of the described feature extraction tools have some drawbacks. Most of them require some type domain knowledge or human intelligence for extracting features. In all, they are semi automatic techniques. The limitations of each of the tools are described briefly below.

The tool "grep" has several limitations in identifying features [Raj02] as the following:

1) When the programmer is unable to guess the appropriate program identifiers, it does not work. In fact, it is based on the ability of a programmer to derive a meaningful identifier that grep searches through the source code.
2) When the concepts are hidden more implicitly in the source code, it is not possible to find the feature using the grep command

As a result, the grep technique depends a lot on the maintainers and developers programming expertise. It does not work properly unless naming conventions were used that clearly encode domain concepts, and the program is not structured. Another problem is after many cycles of maintenance by maintainers, the vocabulary used to describe the software may no longer be the same as its creation. Hence, the grep techniques are likely to become less useful in legacy systems having older and heavily maintained code.

The Bauhaus tool, which is based on Rigi, has a few limitations. It is relatively slow, which can cause a long waiting periods for large graphs. As a result, it sometimes interrupts the fluent use of the tool. It is not possible to automatically lay out Graphs with more than 500 nodes due to limitations of the external layout graphlet [Cze00]. Rigi offers no drag-and-drop functionality, which forces the user to move nodes via clipboard.

The recon2 tool is useful in many program understanding situations. However, it is a complement rather than a replacement for other tools [Raj02]. The results depend both on the user's ability to find good test cases and on the way the original designer may have combined features in the code. Recon2 does not necessarily find all the code related to a particular feature, but it usually finds good starting points for a search.

The XSuds tool needs test cases to exercise the features. Hence, the programmer needs to have some pre-knowledge or domain knowledge to create the test cases, although the accuracy of identifying feature code depends strongly on how adequately the test cases define the feature.

When using the Ripple tool, it is the task of the programmer to acquire all necessary domain and programming knowledge [Che01]. Programmers have to make all decisions: choose the starting component, make changes, and determine the end of the process, etc. Hence, Ripple tools need human intervention. **Table 3** on the following page summarizes the limitations of the current feature extraction tools.

Tool	Techniques used	Limitations and cost
Grep-Unix tool	String-Pattern-Matching	– Need manual searching for identifying the feature location – Features are detected by searching where the identifiers match the string pattern of the feature. If proper naming convention for identifiers is not used, it is hard to locate features. – Not a fully atomic technique as needs human intervention
xSuds tool suite (part of Bellcore)	Test-case-based approach	– Test cases are required to map the features to the source code – Maintainers need basic understanding of the program features so that the test cases can be exercised. – The accuracy of the extracted features depend on how accurate the test cases are – If the test case is limited the maintainer may miss important program behavior.
RIPPLES	Abstract System Dependence Graph (ASDG)	– The ASDG complement the maintenance task, but does not replace them – It is the task of the programmer, not of the tool, to acquire all necessary domain and programming knowledge. – The programmers make all decisions: choose the starting component, make changes, determine the end of the process etc. Hence, it is not a fully automatic technique
Recon2	Test-Case-based approach	– Test cases are required to map the features to the source code. The accuracy of the extracted features depends on how accurate the test cases are.
Bauhaus tool	Resource-graph-based approach	– The Bauhaus tool is relatively slow, which can cause noticeable waiting periods for large graphs [Cze00] – Graphs with more than 500 nodes cannot be automatically laid out due to limitations of the external layouter graphlet[Cze00].

Table 3 Limitations of current feature extraction tools

3 Source code-based feature extraction approach

As discussed earlier in Section 2, feature extraction from source code can aid in reverse engineering activities such as program comprehension [Eis01], software maintenance and reuse [Wil94, Tur99a], evolving legacy systems [Meh01b, Meh01c, Meh02],etc. Typically, externally visible software requirements or functionalities are mapped to the software components in order to extract features using several techniques such as software reconnaissance, testing, concept analysis, etc. In this section, we will define our research objective, outline our contributions and describe in detail the approach used to extract features from the source code.

3.1 Motivation

Most of the existing techniques for feature extraction such as software reconnaissance [Wil95], FODA [Kan90], concept analysis [Sti97], etc., are based on the assumption that some type of domain knowledge of the system is available. A feature in terms of the problem domain is acquired at first with the aid of test cases, scenarios and domain analysis (or a combination of them). Then, this information is mapped to the source code and extracted.

Figure 9 illustrates this typical scenario. Features are extracted by utilizing all three resources, domain knowledge, test cases and the associated execution traces and the source code. However, often the only available and reliable source for the potential feature extraction might be the source code and test cases. In addition, up-to-date documentation or domain knowledge might not exist for a system.

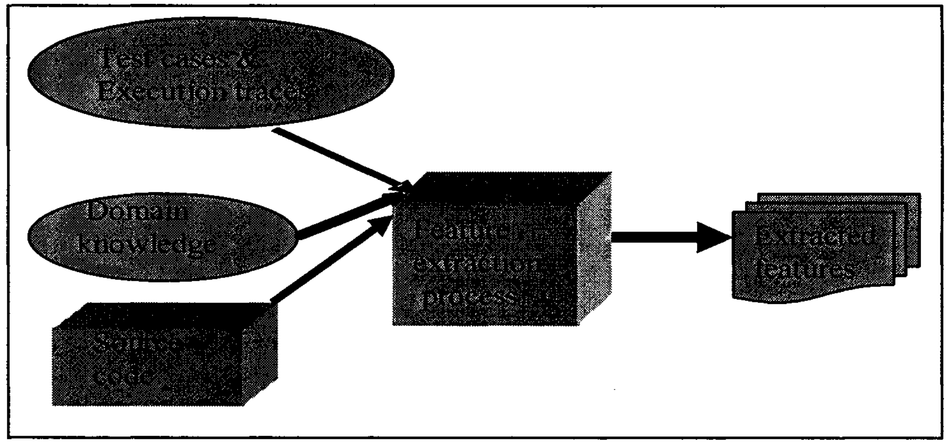


Figure 9 Traditional approaches to feature extraction

The motivation for this research is to overcome some of the limitations of existing feature extraction approaches. This is accomplished by developing a semi-automatic source code based feature extraction approach based on program slicing of the source code. The resulting source code based features can then be applied towards various application domains. These include program comprehension, software reuse, software evolution, creation of test cases, feature-based testing, etc. [Eis01, Wil94, Tur99a, Meh01b]. The feature extraction approach presented in this thesis (shown in **Figure 10**) extracts feature from the source code, based on test cases and the execution traces obtained from executing these test cases. The major advantages of the presented approach are as follows:

- There is no need for domain knowledge from sources like documents or domain experts.
- The feature extraction approach is a semi-automatic approach that only requires source code and test cases to extract features.
- No user interaction is required during the extraction process itself.
- The extracted features correspond to functional features.

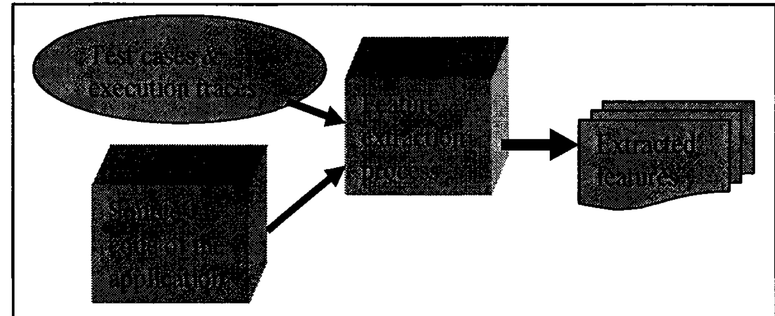


Figure 10 Feature extraction process based on our research

In what follows we extend the previous feature definitions presented by [Tur99a, Eis01] to provide a more user perspective oriented view of source code features. In [Turn99], user perspective is described as the following:

User perspective is concerned with the problem domain. Users interact with the system and are directly concerned with its functionality. Users think of the system in terms of the features provided by the system.

A feature is an identifiable bundle of system functionalities that help characterize the system from the user perspective [Tur99a]. We define a feature as follows:

Definition

A feature is a group of statements which constitutes an executable or non-executable program, based on functional requirements corresponding to system inputs and outputs.In other words, a feature implements functional requirements from a user's perspective.Therefore a feature can be seen as a group of statements which constitutes an executable or non-executable program that perform some actions based on user's input or system's output.

In the next section, we provide a categorization of features based on different criteria that can be applied to extract features (either executable or non-executable).

3.2 Feature categorization

There exist several properties one can use to categorize features. In what follows we present a more detailed discussion on our feature categorization which is based on the following categories.

- Execute-ability
- Extraction Technique
- Feature Type

Executable versus non-executable features

A feature might represent an executable or non-executable program. An executable feature can be described as a group of statements that constitute an executable subprogram that preserves the semantically correct behavior of the program. An executable feature facilitates the program understanding process and can be applied for testing, reuse, and debugging. For instance, in debugging, one often is interested in a specific execution of program that exhibits irregular behavior. Hence, an executable feature can be used to detect the fault in a program feature. However, it should be noted that one of the major disadvantages of an executable feature is the feature size. In order to compute an executable feature, it has to be a semantically correct program, which requires the inclusion of statements that might not be directly related to the feature itself. On the other hand, a non executable feature corresponds to a set of statements that cannot be executed independently, but contain only statements that are relevant for the particular feature and therefore might result in a smaller feature size (compared to the executable feature). This size reduction of the non-executable feature is a result of not having to

guarantee the semantic correctness of the program and therefore statements like, variable definitions or method headers, etc are not included in the feature. As a result, the application of the non-executable feature is somewhat limited. This is because it provides developer with a more general comprehension support by identifying these parts of a program that are directly influencing the feature and its implementation. Hence, the presented research is focusing on the computation of executable features.

Dynamic versus static feature extraction

For the second feature category, we distinguish between static and dynamic analysis. The static computation on the source code is parsed and analyzed. The dynamic computation on the other hand utilizes execution traces that are generated by recording program executions for specific input(s).

Static feature

For a static feature, information is parsed from the source code only. **Figure 11** provides an example of a static feature where the "if" condition affects the execution of two separate statements *S1* and *S2* respectively. When the value of input variable *c* is true, statement *S1* should be executed, and *S2* should be executed otherwise. In **Figure 11,** when computing a static feature, *S1* and *S2* will be executed, although the execution of *S1* or *S2* corresponds to a specific input of variable *c*. For the static feature extraction, no dynamic (run-time) information is available to determine the run-time behavior (program path that might be selected). A static feature therefore has to consider all possible program inputs (execution paths) during the computation process.

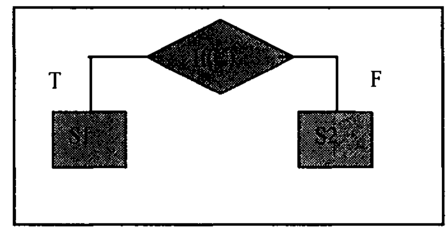


Figure 11 Example of a static feature

Dynamic feature

A dynamic feature computation is based on a particular program input resulting in a specific program execution. The dynamic feature extraction is based on run-time information in the form of recorded execution traces. Utilizing these execution traces allows determining the execution path that will be taken by the program for the specific input (execution).

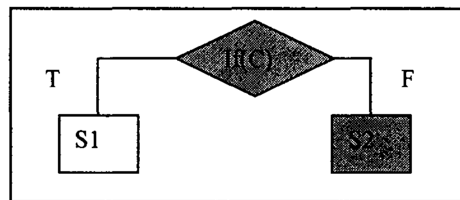


Figure 12 Example of a dynamic feature

Figure 12 shows an example of a dynamic feature where for a particular input of C (in this case the condition evaluates to false). In this case only the If statement and statement S2 would be executed and therefore considered in the dynamic feature computation.

This research work is focusing on dynamic feature analysis. The reason is that a dynamic feature is smaller in size (compared to a static feature) as it does not need to consider all possible program inputs (execution paths) during the computation process. However, the computation process of a dynamic feature is semi automatic because execution traces need to be recorded based on run time information. On the other hand, a static feature can

be computed solely based on the source code information which causes the feature extraction process to be fully automatic. However, one of the major disadvantages of a static feature is its size. Since all the possible execution paths needed to be considered, it might require inclusion of statements that might not be related for the computation of a particular feature. Consequently, the feature computation process might be slower compared to a dynamic feature.

Functional versus Informational Features

Features can be computed based on different output properties of a program. One can distinguish between output information that is used to display information in order to provide the user with information not directly relevant to the program functionalities (e.g. a welcome screen, system status information, etc).

```
1. int x;
2. x = x+10;
3. System.out.println("Welcome");
```

Figure 13 Example of an informational feature

For instance, statement 3 in **Figure 13** is used to provide the user with a welcome message rather than providing any system functionalities. On the other hand, there are output statements that are an essential part of the program or are a direct result of a program's functional requirements.

```
Double balance;
...
1. Account a = new Account();
2. a.deposit(20); //current balance is balance= balance+20
3. System.out.println("The current balance is "+a.currentbalance());
...
```

Figure 14 Example of a functional feature

For instance, **Figure 14** illustrates the output statement statement3 is calling a method currentbalance() in order to show the user the current balance of the system after 20 dollars is deposited in the user's account. Hence, statement 3 reflects the current balance of the system where the current balance is directly influenced by the deposit Functionality of the system.

This research is focusing on extracting the functional features rather than the informational features because our goal is to extract the source code of the program that represents the functionalities of the system.

3.3 Research approach

Source code and existing test cases based on execution traces are required for the feature computation in this presented approach. Executable dynamic features are extracted by deleting the irrelevant statements from the program for that specific feature. The extracted executable dynamic feature should conform to the original functionality with respect to the given feature criteria based on program slicing [Wei82]. As there are several program slicing techniques available, this presented work used backward and forward program slicing for extracting features. For backward slicing, the definition is taken based on [Hor90], where a backward slice is defined as a set consisting of all statements and control predicates that affect the computation of the slicing criterion. On the other hand, forward slicing [Hor90, Har01] is represented as a set that consists of all statements and predicates which are affected by the slicing criterion. In the presented feature extraction approach, the feature criteria corresponds to input and output statements.

Feature extraction method

Input and output statements of the program are represented as the feature criteria for feature extraction since they work as the interface for providing interaction between the system and the user [Tan99]. Hence, features are further categorized into input and output features. An output feature is computed based on an output statement in which a program computation is displayed. On the other hand, an input feature is computed based on a statement in which an input statement modifies the value of a program variable. In what follows, we describe in detail the input and output feature extraction techniques applied in our approach.

Techniques for the extraction of output feature

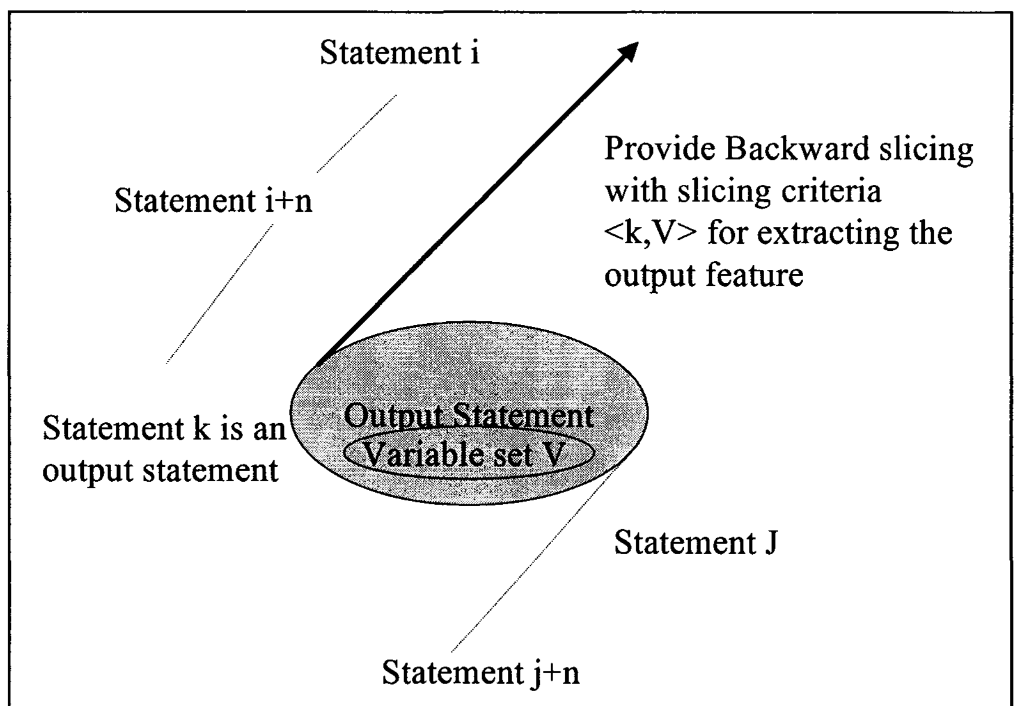


Figure 15 Method for extracting output feature using backward slicing

For extracting an output feature, we are interested to identify which statements affect the value of the variable used in an output statement. Suppose k is an output statement where variable v is used, backward slicing with slicing criterion <k,v> has been applied to extract the statements that lead to the computation of variable *v*. The described technique is illustrated in **Figure 15**. The up arrow from *k* was used to show that backward slicing has been applied from the output statement *k*.

Figure 16 provides an example of computation for extracting output features using the backward approach. The process can be divided into 4 major steps. In step 1 the sample source code of a small program is shown and the output feature criterion (statement 6) is identified. In step 2 the direction of the algorithm to be applied has to be determined. For output features the backward slicing technique has to be applied. In step 3 the program dependence graph (PDG) is derived. Step 4 shows the extracted feature based on traversing backward the incoming edges from the feature criterion at statement 6.

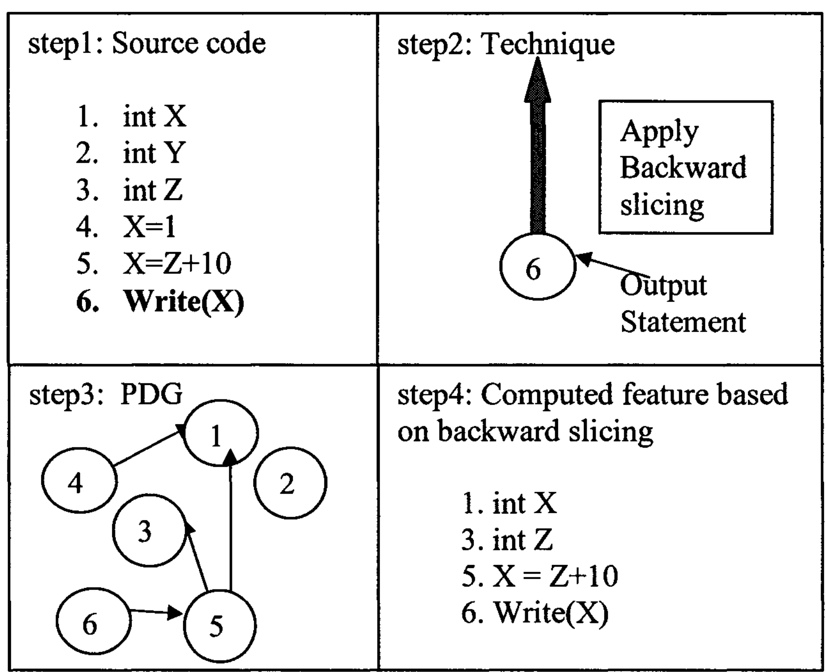


Figure 16 An example of the output feature extraction technique

Techniques for extracting an input feature

The input feature computation identifies the statements that are influenced by the value of the variable used in an input statement. Therefore, the forward slicing approach has been applied to identify these statements that will be influenced by the input statement. For the computation of an executable feature, it is necessary to include variables definitions and other necessary statements. Therefore, the forward algorithm has to be combined with the backward algorithm to compute an executable feature. In **Figure 17**, an input statement *I* uses the variable set *V*. The forward slicing (indicated by the down arrow) is used to identify the statements that are being used by *V*. The backward slicing algorithm is applied in an iterative process to identify and include all relevant control and data dependencies. This is done to make the feature a syntactical and semantically correct program.

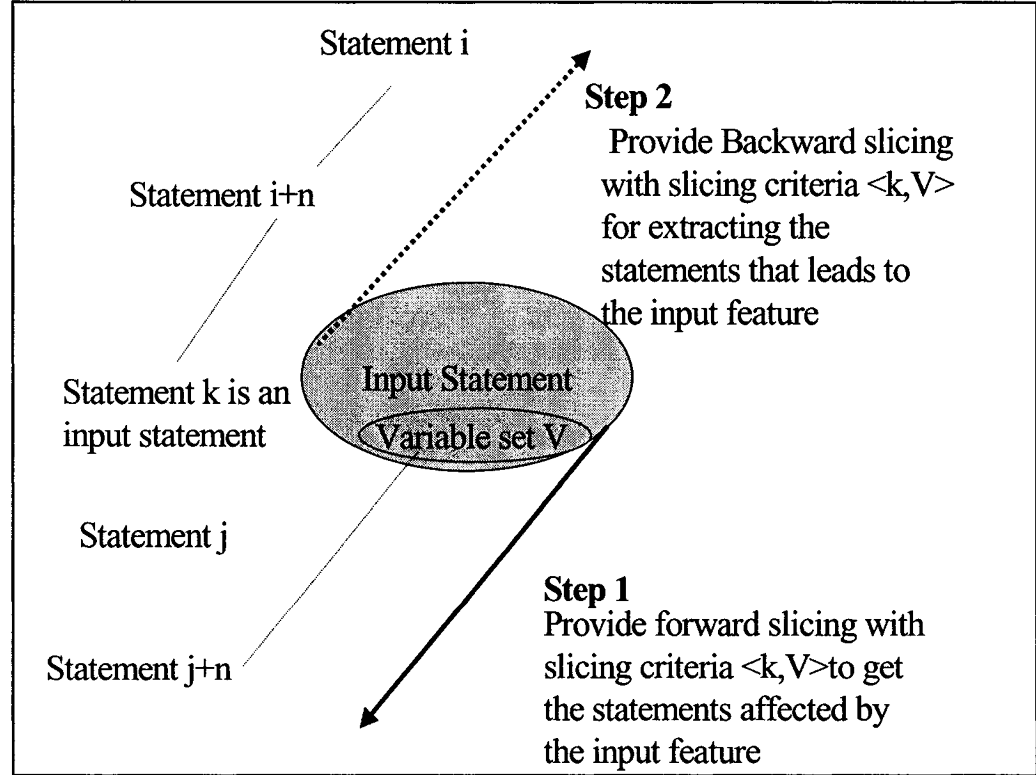


Figure 17 Provides an overview of the feature extraction approach for input features.

In **Figure 18**, a short example is used to illustrate the application of our feature extraction algorithm for input statements. Step1 shows the source code of a program with statement number 4 being the input feature criteria. In the next step, forward slicing is applied to identify all statements that are affected by the input criteria through data dependencies. Step 3 shows the computed slice using the forward slicing approach. From step 3, we can see that statement 6 is not part of the slice as it does not have any data and control influence from statement 4. Next, in order to make the feature executable, it is necessary to apply the backward slicing algorithm for every used variable in the forward slice. This is done to guarantee that the behavior of these variables corresponds to the original behavior. Applying the backward slicing algorithm will identify all statements that influence the computation of these variables up to the specified position in the source code. The resulting feature is shown in step 4.

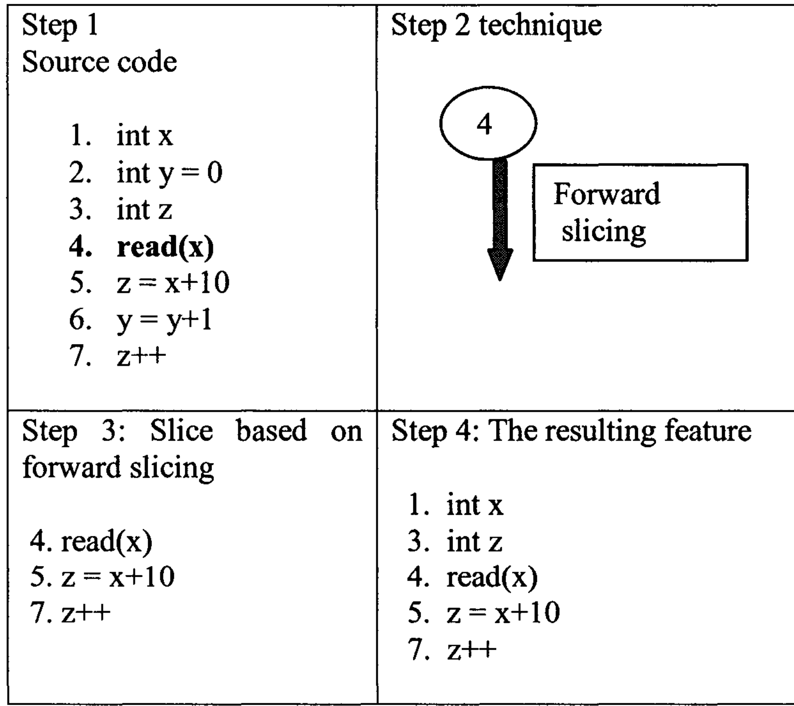


Figure 18 Input feature extraction techniques using the combination of backward and forward slicing

3.4 Feature extraction algorithm

The slicing based feature extraction algorithm used in this thesis applied dynamic backward slicing and dynamic forward slicing algorithm. Hence, the presented approach used methods and terminology from the dynamic slicing techniques. The basic terminologies are as follows:

Execution Trace

An execution trace *Tx* is an abstract list or sequence whose values are accessed by position in it [Ril98]. The execution trace, as used in this research, records the executed position and line number for a particular input. To create the necessary execution trace, monitoring statements are created in the source code. The node x at position k in *Tx* will be written as x^{k}, and will be referred to as an action.

Removable blocks

Removable Block as defined by Korel is "the smallest part of program text that can be removed during since computation without violating the syntactical correctness of the program" [Kor97A]. Input/output and assignment statements are examples of removable blocks. The conditional expressions or control statements (if-else, while, for, do blocks) are not removable, so these statements are not considered as removable blocks. Each block *B* has 1) a regular entry and exit referred to as its r-entry and 2) a regular exit called an r-exit.

Block Traces

S(*B*, *k1*, *k2*) denotes a block trace, which is the part of the execution trace corresponding to the execution of block *B*. In other words, a block trace is a "sub trace" of execution trace *Tx* where *k1* and *k2* are the position of the r-entry and r-exit of *B* respectively.

Moreover, the execution does not exit from block B through its r-exit between $k1$ and $k2$-1 [Kor97A, Cha04].

Description of the Feature Extraction algorithm from source code

This following section describes in detail the feature extraction algorithm presented in this thesis. In the first step (line 1) of the algorithm, as shown in **Figure 19**, a program p is executed on input x and the execution is recorded up to the execution position q where q contains the last executed statement of the program. Set Rc contains initially a set of all blocks in the program. In step 3, based on the information obtained by the parser, and from the execution trace, executed input and output statements are identified. Each input statement is added to the Inp_s as an action, and each output statement is added to the Out_s. Initially, all the actions in Inp_s and Out_s are marked as not visited.

The algorithm iterates in the repeat loop 4-19 until all actions in the Inp_s and Out_s are marked as visited. There are two major steps inside the loop. The user has the option to chose whether the current interest is to compute an input or output feature. In step 6 if the interest is to compute an input feature, then, if there exists any not visited action x^k, procedure "compute feature" is called with the action x^k as the parameter. This step (procedure) is presented in more detail in lines 20-24 of **Figure 19**. Step 20 shows the declaration of the procedure "Compute input feature" for the given action x^k. Initially, the action x^k in Inp_s is marked as visited. Next, the variables v used in the input statement are selected. If v is not empty, that means the slicing position includes at least one variable. The major component of this step is calling the forward slicing algorithm in line 23 to get

the statements influenced by action x^k. The forward slicing algorithm is explained in detail in section 3.6.

Step 12 shows if the user chose to compute an output feature, the procedure compute output feature is called when a not visited and marked action exists. This step (procedure) is presented in more detail in lines 25-30 of **Figure 19**. The procedure mark the action x^k as visited. Next, for the variables read in $x^{k,}$ backward slicing algorithm is called to get the statements that modified the value of action x^k. The backward slicing algorithm is explained in detail in section 3.5. Step 25 shows the procedure compute output feature for the given position x from feature extraction algorithm. Step 25 shows the variables used in the given position are stored in v. If v is not empty, that means the slicing position includes at least one variable. Hence, the backward slicing algorithm is called with slicing criterion $<x,v>$ when the feature candidate is an output feature. Finally, step 18 shows that after a feature is computed (either input or output feature), it is displayed to the user.

Input: Source code of program *P*

Output: Extracted input and output features

Tx: Execution trace of the program *P* on input *x*
Qc: A set of block traces
Rc: A set of blocks
Out_s : Output statements of the program
Inp_s : Input statements of the program

1. Execute project *P* on input x and record execution trace *Tx* up to position *q* (the last executed statement of program *P*)
2. Initialize *Rc* to all blocks in project *P*
3. Based on the static and dynamic information identify executed input and output statements and store the execution positions into Out_s and Inp_s respectively

4. **Repeat**
5 Select choice
6 **if** choice is input feature
7 **if** there exists a marked and not visited action x^k in Inp_s **do**
8 Select a marked and not visited action in Inp_s
9 Compute input feature for x^K
10 **end if**
11 **end if**
12 **else** **if** choice is output feature
13 **if** there exists a marked and not visited action in Out_s **do**
14 Select a marked and not visited action in Out_s
15 Compute output feature
16 **end if**
17 **end if**
18 Show the computed feature
19 **until** there exist a marked and not visited action

20 **Procedure** Compute input feature
21 mark x^k as visited
22 **if there exist** variable $v \in U(x^k)$
23 Call forward slicing algorithm with slicing criteria <*x*,*v*>
24 **end** Compute input feature

25 **Procedure** Compute output feature
26 mark xk as visited
27 **if** there exist variable v ∈ U(xk)
28 Call the backward slicing algorithm with slicing criteria <x,v>
30 **end** Compute output feature

Figure 19 Algorithm for computation of input and output feature

3.5 Description of the backward slicing algorithm

The dynamic backward program slicing algorithm [Kor97, Ril98] identifies the actions in the execution trace *Tx* which contribute to the computation of a slicing criterion $C = (x, y^q)$. This is obtained by deriving the data and control dependencies. However, it is also important to identify the actions which do not contribute to the computation of variable y^q. The reason is that the more non-contributing actions are identified, the smaller may be the resulting slice. The data dependencies are used to identify the contributing actions and the removable blocks, and the non-contributing ones. Naturally, a block can be removed from a program if its removal does not interrupt the flow of execution for input *x*. Let *B1*, *B2*, and *B3* be a sequence of three blocks. Block *B2* can be removed if during the execution of the program for input *x* the execution exits from block *B2* thorough its r-exit, and enters block B2 through its r-entry, leaves *B2* through its r-exit, enters block *B3* through its r-entry. The final condition is that none of the executed actions within *B2* contribute to the computation of y^q. If *B2* is removed and the program is executed for the same input *x*, then after leaving *B1* through its r-exit, the execution will enter *B3* through its r-entry. This removal will not affect the flow of execution or the computation of variable y^q.

The backward algorithm using Korel's approach [Kor97A, Ril98] is shown in the **Appendix 2**, and the algorithm can bee summarized as follows:

- Initially the program is executed and its execution is recorded up to execution position *q*. The actions in the execution trace can be in contributing, non-contributing or neutral state.

- Initially, all the actions in the execution trace are marked as neutral and not visited, and *Nb* (a set of non-contributing blocks) is initialized of all blocks in the program.
- Step 2 in **Appendix 2** shows the last definition of y^q is identified, and is marked as contributing and not visited.
- Next, the algorithm starts a repeat loop. The loop iterated until all the actions are marked as either contributing or non-contributing.
- Inside the repeat loop, the procedure find contributing actions is called to identify actions that contribute to the computation of the y^q. When there exist a contributing and not visited action, the contributing action is marked as visited. Subsequently, in a repeat loop the last definitions of all variables used in action X^k (contributing action) is identified. Next, all blocks that contain node X are removed from *Nb*. The procedure continues until all contributing actions in the execution trace are visited.
- For the given set of contributing actions, the algorithm identifies non-contributing actions using the procedure "find non-contributing actions". This procedure finds non-contributing actions in a repeat loop by finding a set of block traces for the set of blocks *Nb*. The procedure initially marks all actions as neutral if they are not already marked as contributing actions. The procedure explores the execution trace from the beginning looking for actions that are marked as neutral. If such an action is found, then for this action the procedure tries to identify block trace $S(B,p,p1)$of block B with block entry at p and block exit $p1$. If all the actions between p and $p1$ are not marked at contributing, then all the actions between p and $p1$ are marked as non-contributing and the algorithm continues searching for non-contributing actions from position $p1$. Otherwise, the algorithm tries to find the next neutral actions

starting at position p+1. The details of how to compute the non-contributing actions are given in step 17 to 31 in **Appendix 2**.

- Actions which are not identified as contributing or non-contributing actions are marked as contributing actions.
- Finally, the dynamic slice that is constructed from P by removing all blocks from Rc that belong to non-contributing blocks N_b.

3.6 Description of the forward slicing algorithm

The forward algorithm used in this thesis is a modified version of the backward slicing algorithm with removable blocks proposed by Korel [Kor97, Ril98]. However, the forward slicing algorithm varies from the backward algorithm in several ways.

First of all, the dynamic forward program slicing algorithm identifies the actions in the execution trace Tx which are influenced or "affected" by the computation of a slicing criterion $C = (x, y^q)$. In contrast, the backward algorithm identifies the statements that affect the slicing criteria.

Last definition versus first usage reference

The forward algorithm varies from the backward algorithm in step 2 of **Appendix 2** in that instead of calling the last definition of the variable, get first usage reference is called to get the first place where the variable v is used.

```
....
1. int x
2 .y = x + 1
3 System.out.println(y);
....
```

```
1. int z = 10 ;
2 .int y = 5 ;
3.Strnum n = in.readLine(_)
4. z = y+1
5. y = n+3
```

Figure 20 a) Last definition example b) First usage reference example

For instance, in **Figure 20** (a), if the slicing criterion is statement number 3 for variable y, then last definition of y would be in statement number 2 because the value of y is assigned in that statement. On the other hand, in **Figure 20** (b) if the slicing criterion is statement 3 for variable "in", the value of in.readline() is assigned to "in". Next, the value of n is used in statement 5. As a result, the first usage reference method for statement 3 would return statement 5 as the position because statement 5 is influenced by the value of strnum at a forward direction, and the value of strnum is affected by the value of variable in.

Another difference is given in the find contributing actions method. In step 13 of the find contributing actions method rather than getting the last definition of v, the first usage reference is marked as a contributing action in order to get the influenced statements by v in the forward direction. The highlighted statement 13 in **Figure 21** shows the modified step from the backward slicing algorithm in order to compute the slice using the forward slicing algorithm.

procedure Find contributing actions
9 while there exists a contributing and not visited action in Tx do
10 Select a contributing and not visited action Xk in Tx
11 Mark X^k as a visited action ($I_C := I_C \cup \{X^k\}$)
12 **for** all variables $v \in U(X^k)$ **do**
13 **Find and mark as a contributing action the first usage reference of v**
endfor
14 **for** all blocks $B \in R_c$ **do**
15 **if** $X \in N(B)$ **then** $N_B := N_B - \{B\}$
endfor
16 **endwhile**
end Find contributing actions

Figure 21 Modified method from backward slicing algorithm

Since the dynamic slicing algorithm introduced by Korel was designed for the procedural version of the Pascal programming language, some modifications were needed to be done in order to compute the slices for object oriented programs written in Java.

A method called "mark action as contributing" was added to the backward algorithm by Charland [Cha04] so that the algorithm can handle constructor or method calls. In the backward algorithm, if an action is marked as contributing and this action contains a method call with a return type other than void, then the return statement of the method body is also marked as contributing. However, in the case of an action with a constructor or method call with a void return type, all actions within the body of the constructor or method are also marked as contributing. On the other hand, in the forward algorithm when an action is marked as contributing which contains a method call or a constructor call, all actions within the body of the constructor or method are also marked as contributing action as shown in **Figure 22**. If a method or constructor is called, all the executed statements in the method or constructor are directly affected by the method call.

procedure Mark action as contributing
1. Let X^p be an action at position p in Tx
2. Let S($B, p, p1$) be a block trace with an r-entry at position p
3. Mark x^p as a contributing action
4. For all method calls and contructor calls at x^p do
5. Mark action as contributing all the actions which belong to the body of the constructor or method between p and $p1$

   End for

end Mark action as contributing

Figure 22 Modified procedure added to the algorithm

For example, **Figure 24** shows the execution trace of the sample program of **Figure 23**. If the first action (which is a constructor call) in the execution trace is marked as contributing, then all actions from position 2 to 3 will also be marked as contributing.

```
...
10 Account A = new Account(100);
...
15 Public Account(double amount){
16 Balance = amount;
}
```

Figure 23 Sample program with a constructor call

Execution Trace	Action State
...	
10^1 Account A = new Account(100)	Contributing
15^2 Public Account(double amount)	Contributing
16^3 Balance = amount	Contributing

Figure 24 Execution Trace of the sample program of Figure 23

Incorporate backward slicing with the forward algorithm

Next, in order to make the slice executable after the slice is computed based on the above algorithm, the backward slicing algorithm is called for the given feature criteria. When the backward slice is computed, it is incorporated with the forward slice. In that case, the contributing blocks from the backward slice also have to be added to the contributing blocks of the forward slice. This step is shown in detail in **Figure 25**. This can be done as the following:

The noncontributing block that belong to the forward slice, but is not in the non contributing blocks of backward slice, remove the block from forward slice. In other words, if any block is not in the non contributing blocks of backward slicing, this block is a contributing block for the backward slicing. Also, the contributing actions of the backward slice are added to the contributing actions of the forward slice. In **Figure 17**,

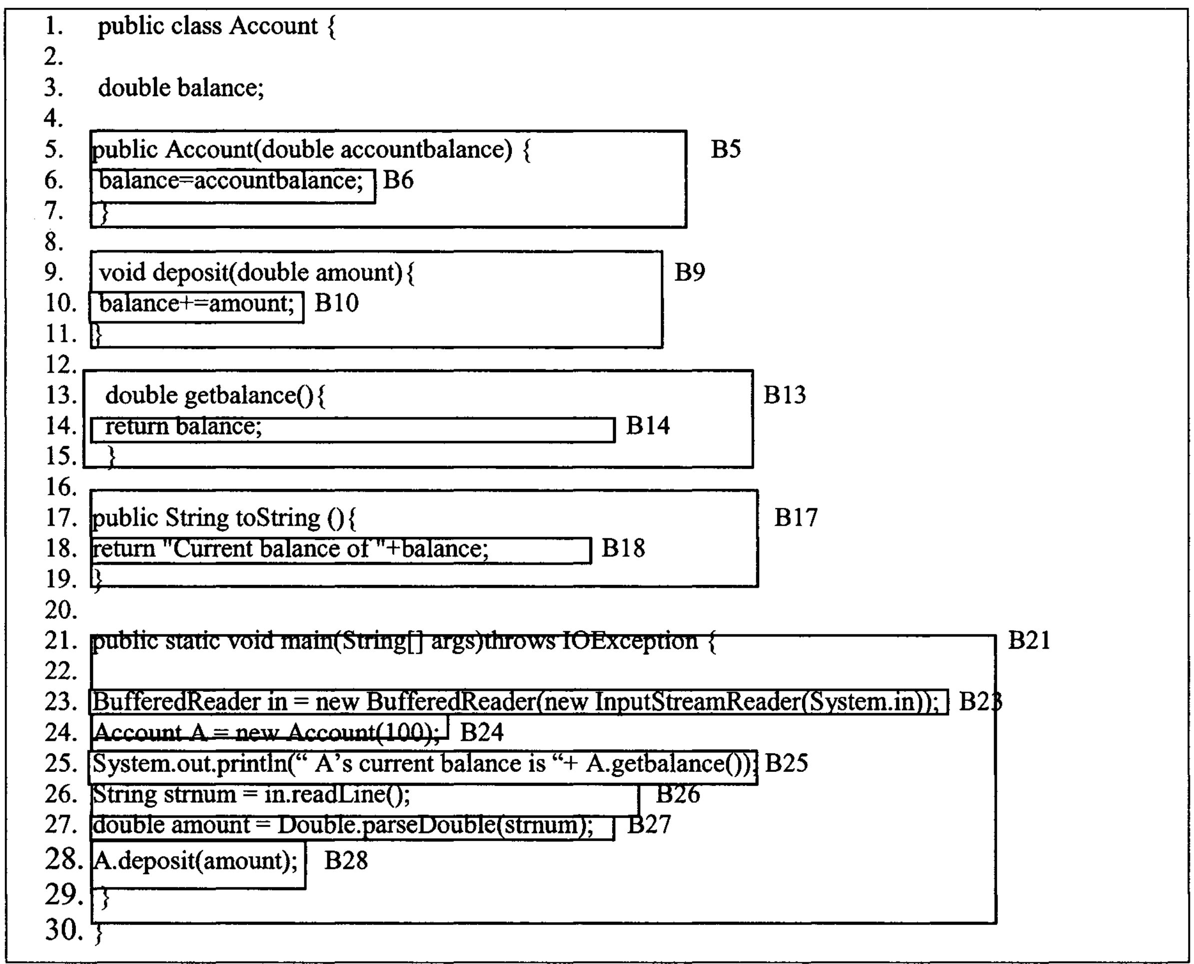
step 30 shows the backward slicing is called for the input feature criteria, after the computation of slice is done in the forward direction for from the backward slicing is assigned to *Ic(backward)* and *Nb(backward)* in step 32 and 33 respectively. Step 33-36 shows that when there exist any non contributing block in forward slice which is not an element of the non contributing block of the backward slice, the block is deleted from the non contributing blocks of the forward slice. Finally, step 37 shows that the forward algorithm returns all the contributing blocks including the contributing blocks from the backward slice as well.

1. I_c = Contributing actions of forward algorithm
2 N_b = Noncontributing blocks of forward algorithm
3 I_c*(backward)* = Contributing actions of backward algorithm
4 Nb(backward) = non contributing blocks of backward algorithm
……
10 The forward algorithm before calling the backward slicing
………..
30 Call the backward slicing for $\{C = x, y^p\}$ (returns the contributing blocks)
31 *Ic*(backward) = Contributing Actions of backward algorithm
32 *Nb(backward)* = Non contributing blocks *(backward)* of backward algorithm
33 Repeat
34 if (there exist a block *B* in *Nb, but it does not exist in Nb(backward)*
35 Remove block B from Nb
36 *Until all the blocks of Nb are traversed*
37 *Return (all blocks of the program – Nb)*

Figure 25 Integrate backward slicing with forward slicing algorithm

A sample program has been to taken to show the above described feature extraction algorithm technique. The sample program Account is shown in **Appendix 1**. It has an instance variable called balance, and has methods such as deposit that deposits the amount the user wants. Also, the method getbalance returns the current balance of the system. The toString method returns the account object in a string repreesntation. The constructor Account() create a new account object with the amount specified. In order to

extract feature, initially removable blocks of the program is stored in *Rc* which is shown in **Figure 26** as described in the algorithm. Next, the execution trace and block traces of the Account program is need to be recorded which is illustrated in **Figure 27**. The input and output statements are needed to be identified. **Figure 19** shows the highlighted input and output statements. In this example, there is only one input statement action 26^9 which is added to Inp_s, and the only output statement action 25^6 is added to Out_s.



```
1.  public class Account {
2.
3.  double balance;
4.
5.  public Account(double accountbalance) {                B5
6.  balance=accountbalance; B6
7.  }
8.
9.  void deposit(double amount){                           B9
10. balance+=amount; B10
11. }
12.
13.  double getbalance(){                                  B13
14.  return balance;                          B14
15.  }
16.
17. public String toString (){                             B17
18. return "Current balance of "+balance;     B18
19. }
20.
21. public static void main(String[] args)throws IOException {          B21
22.
23. BufferedReader in = new BufferedReader(new InputStreamReader(System.in)); B23
24. Account A = new Account(100); B24
25. System.out.println(" A's current balance is "+ A.getbalance()); B25
26. String strnum = in.readLine();          B26
27. double amount = Double.parseDouble(strnum);  B27
28. A.deposit(amount);  B28
29. }
30. }
```

Figure 26 Removable blocks of the sample program

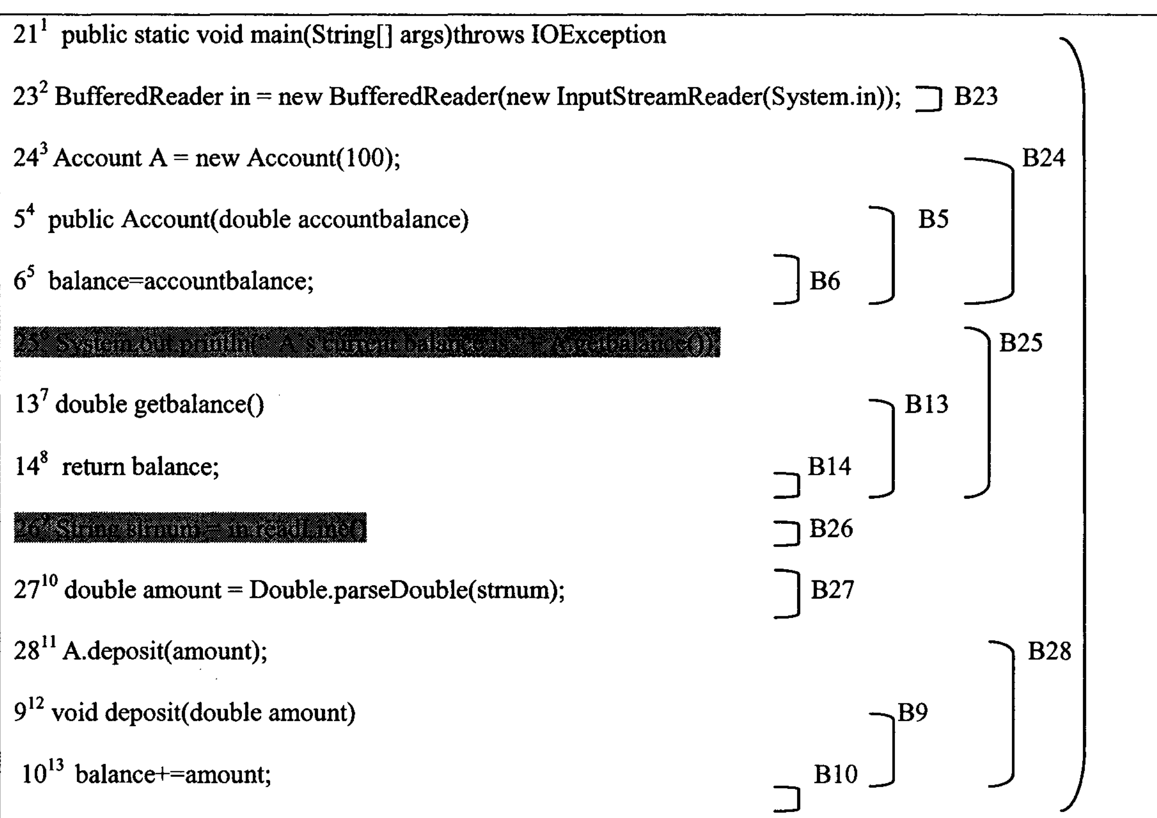


Figure 27 Execution trace and blocks traces of the sample program

Based on the above information

$Inp_s = \{26^9\}$

$Out_s = \{25^6\}$

If the choice is to compute an output feature then

$Out_s = \{25^6\}$ is taken as the feature criteria

Variable used in action 25^6 is A

The backward slicing algorithm is called with the slicing criterion (25, A)

In the first iteration of the backward slicing algorithm

$Ic = \{25^6, 24^3, 5^4, 6^5, 13^7, 14^8\}$

Rc = {B23,B26,B27,B28,B9,B10,B21,B17,B18}

Qc = {S(B23,2,3), S(B26,9,10), S(B27,10,11),S(B28,11,13)}

21^1 is taken as the neutral action because it is neither in the Contributing actions, nor it is in the block traces.

In the second iteration

Ic = $\{25^6, 24^3, 5^4, 6^5, 13^7, 14^8, 21^1\}$

Rc = {B23,B26,B27,B28,B9,B10,B17,B18}

Qc = {Ø}

The extracted output feature based on the backward slicing algorithm is highlighted in **Figure 28.**

```
1.  public class Account {
2.
3.   double balance;
4.
5.  public Account(double accountbalance) {
6.   balance=accountbalance;
7.  }
8.
9.   void deposit(double amount){
10.  balance+=amount;
11. }
12.
13.  double getbalance(){
14.  return balance;
15. }
16.
17. public String toString (){
18. return "Current balance of "+balance;
19. }
20.
21. public static void main(String[] args)throws IOException {
22.
23. BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
24. Account A = new Account(100);
25. System.out.println(" A's current balance is "+A.getbalance());
26. String strnum = in.readLine();
27. double amount = Double.parseDouble(strnum);
28. A.deposit(amount);
29. }
}
```

Figure 28 Highlighted output feature of the account program

The output feature shows that the programmers don't need to understand all the methods of the program for understanding the feature A.getbalance(). The methods that are used for the particular feature representation is the constructor of the program, and the getbalance() method to return the balance.

If the choice is to compute an input feature,

$Inp_s = \{26^9\}$

The variable used in action 26^9 is " in"

After the first iteration

$Ic = \{ 26^{9,} 27^{10}, 28^{11}, 9^{12}, 10^{13} \}$

Rc = { B23, B24, B5, B6, B25,B13,B14, B17,B18,B21}

Qc ={ S(B23,2,3) , S(B24,2,5). S(B25,6,8),S(B13,7,9)}

Action 21^1 is a neutral action because it is not marked as contributing, neither it is in the block traces. Hence, the neutral action is marked as contributing

After the second iteration

$Ic = \{ 26^{9,} 27^{10}, 28^{11}, 9^{12}, 10^{13}, 21^1 \}$

Rc = {B23, B24, B5, B6, B25,B13, B14, B17,B18}

Where Rc represents the noncontributing blocks of the program

Qc = { Ø }

Next the backward slicing is called for the position 26^9

After the backward slicing based on position 26^9 is called

Ic from backward is $Ic = \{23^2, 21^1, 26^9\}$

Rc = {B24, B5, B6, B25, B13, B14, B27, B28, B9, B10}

Qc = { Ø }

Next, the slice from the backward algorithm is needed to be integrated with the forward slice.

The block that is not in the Rc of backward but is in the Rc of forward is deleted from the forward Rc.

Hence, the Ic = $\{26^{9,} 27^{10}, 28^{11}, 9^{12}, 10^{13}, 21^{1}, 23^{2}\}$

Rc = {B24, B5, B6, B25, B13, B14, B17, B18} // noncontributing blocks

The declarations of the contributing actions are also needed to be added to the input feature to make the feature executable and if the declaration contains a constructor call or a method call, the constructor and the method call are added to the slice.

Action 28^{11} is a contributing action included in the slice. The declaration of variable A of action 28^{11} is found in the action 24^{3}. Since action 24^{3} contains a call, the constructor body is also taken as contributing. Hence, the contributing actions for getting the declarations of the contributing actions are as follows:

Ic = $\{28^{11}, 24^{3}, 5^{4}, 6^{5}\}$

The blocks corresponding to the contributing actions that should be removed from the noncontributing blocks are

B28 B5 B6 B24

Hence, finally the forward slicing algorithm gives

Ic = Result of forward integrated with backward + declarations of the contributing actions

= $\{ 26^{9,} 27^{10}, 28^{11}, 9^{12}, 10^{13}, 21^{1}, 23^{2}\} + \{28^{11}, 24^{3}, 5^{4}, 6^{5}\}$

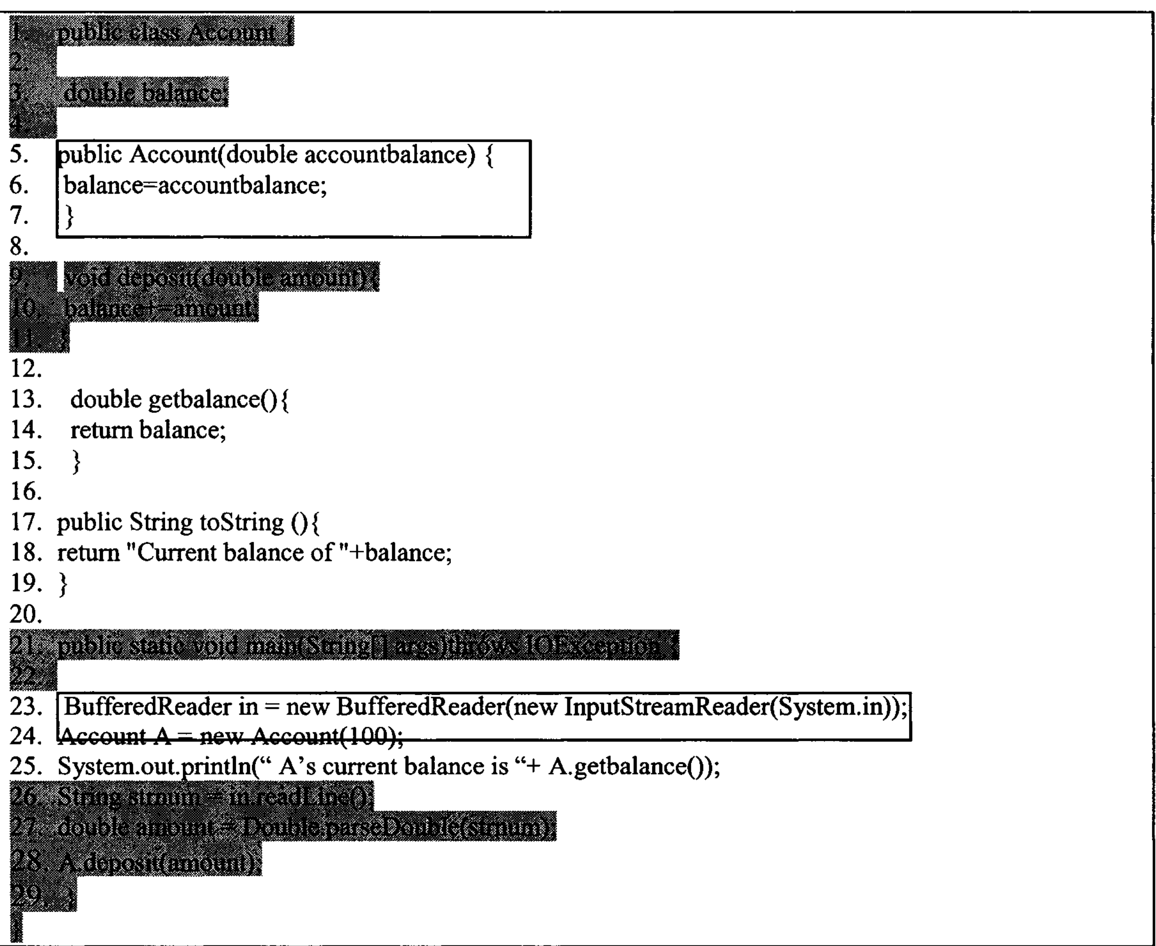
= $\{ 26^{9,} 27^{10}, 28^{11}, 9^{12}, 10^{13}, 21^{1}, 23^{2}, 24^{3}, 5^{4}, 6^{5} \}$

Forward slice = All blocks of the program - (non contributing blocks from forward integrated with backward slicing - contributing blocks for the declarations of contributing actions)

= All blocks of the program – ({B24, B5, B6, B25, B13, B14, B17, B18 }– B28, B5,B6,B24}

= All blocks of the program – { B25,13,14,B17,B18}

The computed input feature using the above approach is shown in **Figure 29**. The extracted statement based on the computation before calling the backward algorithm is shown with the highlighted statements. Next, the result based on the backward slicing for making the slice executable is shown in the rectangular boxes.



```
1.  public class Account {
2.
3.  double balance;
4.
5.  public Account(double accountbalance) {
6.  balance=accountbalance;
7.  }
8.
9.  void deposit(double amount){
10. balance+=amount;
11. }
12.
13.  double getbalance(){
14.  return balance;
15.  }
16.
17. public String toString (){
18. return "Current balance of "+balance;
19. }
20.
21. public static void main(String[] args)throws IOException {
22.
23. BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
24. Account A = new Account(100);
25. System.out.println(" A's current balance is "+ A.getbalance());
26. String strnum = in.readLine();
27. double amount = Double.parseDouble(strnum);
28. A.deposit(amount);
29. }
```

Figure 29 The extracted input feature

4 Implementation

The feature extraction algorithm presented in the present research is integrated within the CONCEPT project. In what follows, we will introduce the CONCEPT project and its system architecture. Later, we will provide the design and implementation details of our Feature Extraction algorithm approach. Finally, the experimental result based on JUnit is provided.

4.1 Concept system architecture

The CONCEPT (Comprehension Of Net-CEntered Programs and Techniques) [Ril01] is a reverse engineering tool that aids in program comprehension of large and distributed systems. The major goal of the CONCEPT project is to provide novel program comprehension techniques and to assist programmers during the creation of mental models while comprehending software systems. These program understanding techniques are based on a variety of source code analysis, visualization, and application approaches. Currently, the CONCEPT project explores new program slicing algorithms such as hybrid slicing and forward slicing, and investigates their application in different software engineering sub-areas areas (e.g., software measurement, design pattern recovery, software visualization, feature analysis and architectural recovery, etc).

Figure 30 shows an architectural overview of the CONCEPT project where the Feature Extraction algorithm is integrated. The CONCEPT framework is built as a layered architecture with Postgres SQL database being the underlying repository. The database stores both static and dynamic source code information that is needed for the feature extraction.

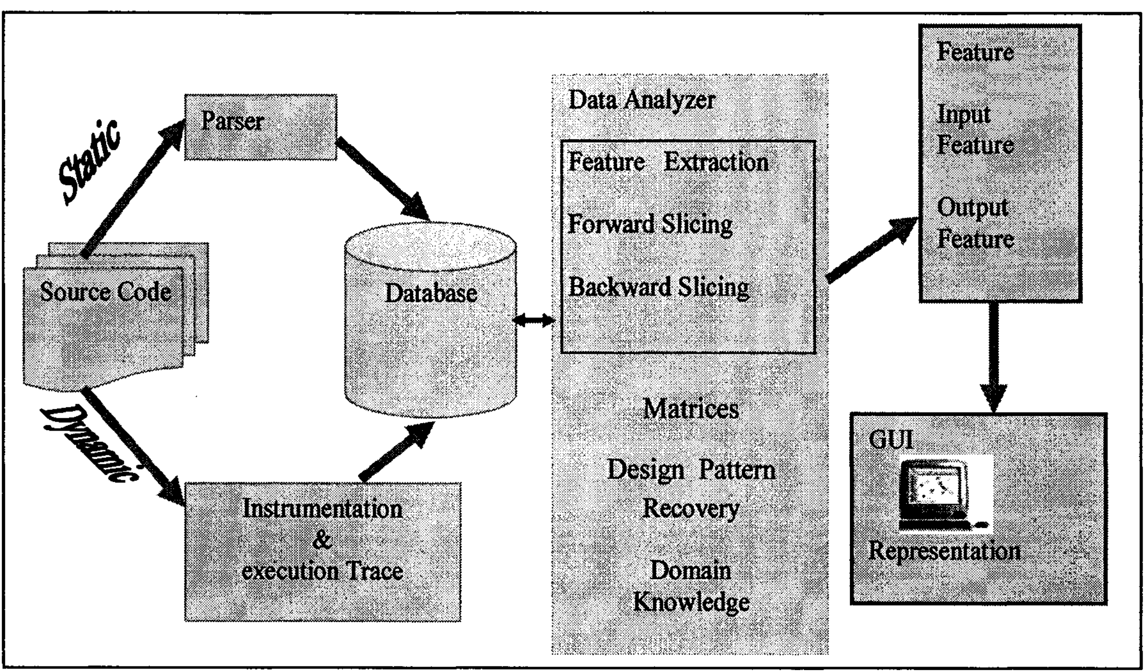


Figure 30 System architecture of the CONCEPT framework

Initially the source code is statically analyzed using a parser. The parser extracts required information such as the set of elements used in the program (e.g., field, class, variables, methods information), a set of relationships between the elements (e.g., file p contains class a, class a contains methods a,b,c, etc) variables used in the program (e.g. variable c is an instance variable, variable c is used and defined in method b), the relationships between method calls (e.g. method a calls method b n times) etc. The static information is represented in the form of an abstract Syntax Tree (AST) that is stored in the Postgres knowledge base. The AST provides detailed static information about structural and logical dependencies in the source code. The AST is stored in the PostgresSQL database for late retrieval. In a next step, dynamic information is collected through a source code instrumenter that modifies the existing source code to include monitoring statements. The

execution trace is created by executing the instrumenting source code and recording the run-time information associated with the particular program execution. The resulting execution trace is stored in the database.

During the data analysis phase, data is retrieved from the repository through database interface (DBI). These obtained data facilitate the computation of different kinds of slicing algorithms, software coupling and cohesion measurement, design recovery; etc .Our feature extraction algorithm uses the forward slicing algorithm and backward slicing algorithm implemented in the data analysis phase. Next, feature extraction algorithm creates the input and output features.

4.2 Feature extraction algorithm implementation architecture

The presented algorithm introduced in Section 3 was implemented using Java program language with version 1.4. *BackwardSlicingAlgorithm*, *ForwardSlicingAlgorihtm*, and *FeatureExtractionAlgorithm* were used as the main basis for computing features.

As illustrated in **Figure 31**, initially the backward slicing algorithm [Kor95, Ril98] was implemented by Charland [Cha04]. The existing backward slicing implementation [cha04] can compute features only when the slicing criterion is a variable. However, slicing computation needs to be done for a method call when the method call contains an output/input criterion. The modified backward slicing algorithm can compute feature when the feature candidate includes a method call which consists of an output or input criteria. The modified *DynamicBackwardAlgorithm* class is used as a data member in *FeatureExtractionAlgorithm*. Another major private data member used in this class is the *DynamicForwardgAlgorithm*. Forward slicing has been implemented with the extension to backward slicing except that forward slicing identifies the statements that are

potentially affected by the slicing criterion instead of the statements that lead to the current position.

The *DynamicForwardAlgorithm* and the *DynamicBackwardAlgorithm* are inherited from the class *SlicingAlgorithm*. Since input and output statements have been used as the basis for the slicing criteria for extracting feature, initially the input and output statements are also stored using the aid of class's *inputstatements* and *outputstatements*. The input and output criteria are stored in two different files. When traversing the execution trace, the method calls are compared with file input or output criteria to see whether they match. If any position is equivalent to the input or output criteria of the file, then the output or input statement is stored in a map with the file id number and the statement number corresponding to the given file. Next, the *DynamicBackwardAlgorithm* class is called in the case of an output statement, and the class *DynamicForwardAlgorithm* class is called when the statement is an input statement. Finally, the *FeatureExtractionAlgorithm* class is called by the *FeatureGui* class in order to provide a graphical user interface representation for the user.

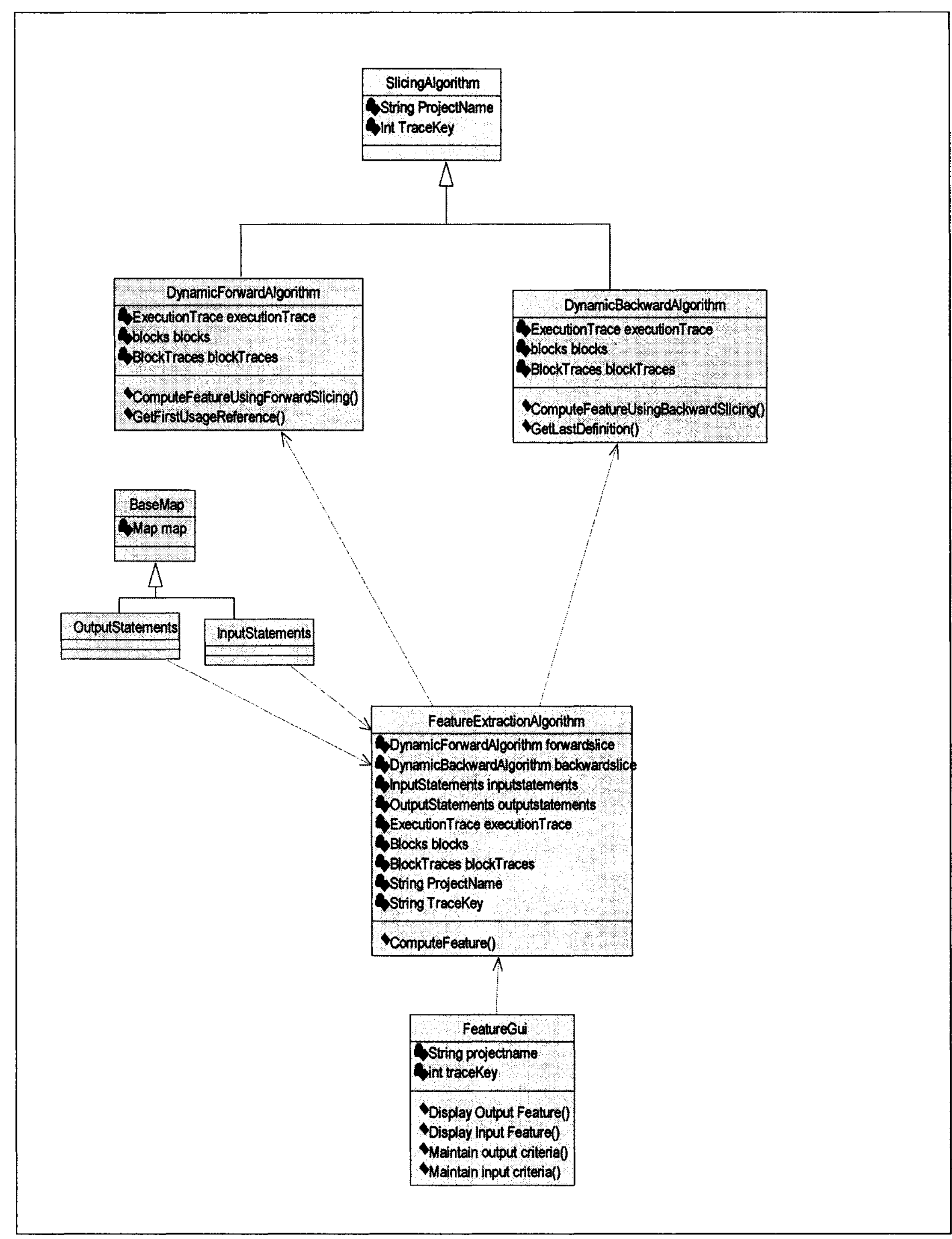


Figure 31 High level view of the feature extraction algorithm implementation

Figure 32 shows the screen capture of the displayed computed features using our Feature Extraction Algorithm on the GUI.

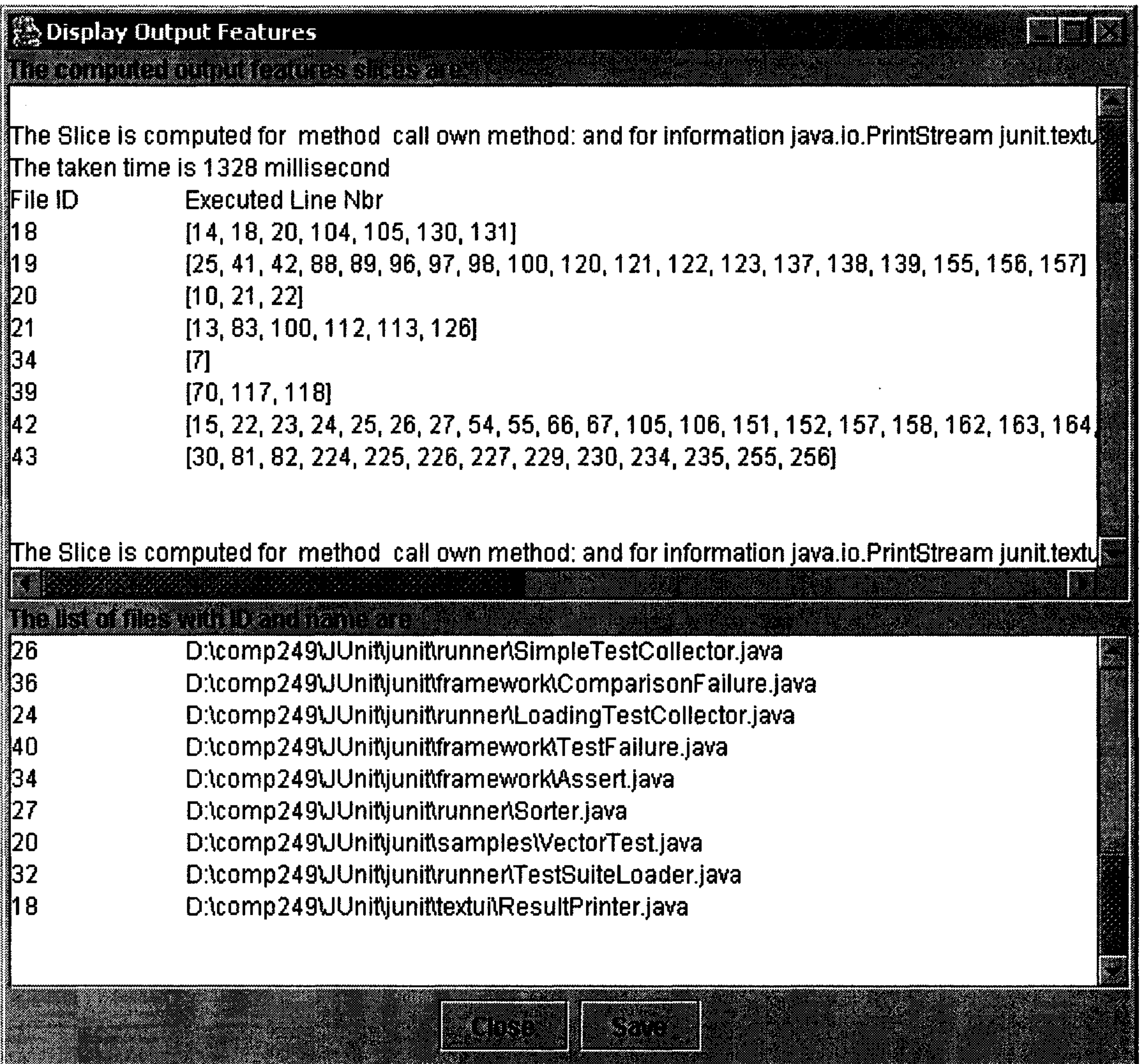


Figure 32 A screen capture of the part of the output features from JUnit

4.3 Case study and experimental result

In this section a case study is presented which was applied to provide an initial empirical validation of the proposed approach. JUnit [Jun01] an open source project testing framework started by Erich Gamma and Kent Beck and developed under the Open Source Initiative (OSI) [Bec98]. JUnit is a testing framework to write repeatable tests. It allows developer to easily create tests for Java code, and to run them more easily and quickly. Furthermore, the result is formatted in a consistent fashion. JUnit has become the

standard testing framework for Java Development. Using JUnit, test cases can be cheaply and incrementally built which helps the programmer to measure their progress, spot unintended side effects, and focus on development effort rather than spending time in debugging the code each time for detecting defects.

A modified version of JUnit 3.8.1[Jun01, Cha04] was used in this experiment which includes 2K of code. JUnit was used in this research because it is a standard open source project, and its source code is available. Moreover, this particular framework has reached a relatively mature development level. In addition, using JUnit it is possible to create test cases of one's own choice, and then to test the result by running the test case class. Its feature consist of assertions for testing expected results, test fixtures for sharing common test data, test suites for easily organizing and running tests, and graphical and textual test runners.

Project Name	**JUnit**
Lines of Code	2358
Total number of executed statements	568
Number of packages	4
Total number of classes	27
Number of executed output statements	**7**
Number of executed input statements	**0**

Table 4 An overview of the JUnit project

Table 4 presents some information about the project size, and the number of executed statements for the particular test case. The initial Vector Test class executes a total of 568 statements, including 7 output statements and 0 input statements.

There are three different test cases that were being used for the experiment to get output feature, and input features. For the initial experiment, the first test case being used with the JUnit project in this research is called the VectorTest class, which is a rather small program. This class was used as a test case for our experiment because it comes with the JUnit project, and it is a simple program which aids to test our implemented feature extraction algorithm. This specific class has two vectors called fEmpty, and fFull as private data member. Next, in the set up method, the fEmpty and fFull vectors are initialized to empty. Subsequently, three elements are added to the fFull vector. The testClone method tests with the assert method whether the fFull vector's size is same as the number of elements that are added to the vector. Also, another assert method tests whether the first element such as 1 added to the fFull vector corresponds to the number 1. If not, an assert failure is returned. Otherwise, the assert method returns true. The VectorTest class is shown in **Figure 33**.

In JUnit the ResultPrinter class displays the result generated by the TestRunner when a given test case class is tested. The class ResultPrinter contains the instance variable fWriter, which is of type PrintStream. It is through this instance variable that the TestRunner class outputs the results of the test cases. The class TestRunner is a command line based tool used to run tests. The text based test runner outputs its results on the console. Hence, if JUnit is executed with the given test case Vector test used in this experiment, the following result is displayed as illustrated in **Figure 34**.

```
1.  package junit.samples;

2.  import junit.framework.*;
3.  import java.util.Vector;

4.  /**
5.  A sample test case, testing <code>java.util.Vector</code>.
6.  *
7.  */
8.  public class VectorTest extends TestCase {
9.  protected Vector fEmpty;
10. protected Vector fFull;

11. protected void setUp() {
12. fEmpty= new Vector();
13. fFull= new Vector();
14. fFull.addElement(new Integer(1));
15. fFull.addElement(new Integer(2));
16. fFull.addElement(new Integer(3));
17. }
18. public static Test suite() {
19. return new TestSuite(VectorTest.class);
20. }
21. public void testClone() {
22. Vector clone= (Vector)fFull.clone();
23. assertTrue(clone.size() == fFull.size());
24. assertTrue(clone.contains(new Integer(1)));
25. }
26. }
```

Figure 33 VectorTest class

The output screen shows that when the VectorTest class is executed, a result is displayed mentioning that 1 test case was executed and the test case successfully passes the validation. Also, the time for computing the result is shown.

```
D:\WINNT\System32\cmd.exe
D:\junitframework>javac junit\runner\*.java

D:\junitframework>javac junit\samples\*.java

D:\junitframework>javac junit\textui\*.java

D:\junitframework>java junit.textui.TestRunner junit.samples.VectorTest

Time: 0

OK (1 test)

D:\junitframework>_
```

Figure 34 Test results generated by JUnit

4.3.1 Experimental results

In this section we present some experimental results from our feature extraction algorithm when applied on the JUnit project. The results shown in **Table 5** are based on the Vector Test class of **Figure 33**.

The first column in **Table 5** corresponds to the identified features, the second column contains the class name and the third column shows the executed line numbers. **Figure 35** shows the file name, and the associated file id for each test case that was obtained after executing each test case separately. Since three test cases were used for 3 different experiments, the file name and id representation is shown for each of the execution. For instance, the File id 18 is the representation of file ResultPrinter.java of JUnit project (execution trace #1 obtained from executing the first test case).

To aid in understanding, **Appendix 3** shows the identified output statements by this given algorithm when executing the JUnit project using the test case Vector Test for our first experiment. The class ResultPrinter contains the executed output statements.

Packages	Class name	Execution 1 File id	Execution 2 File id	Execution 3 File id
Framework	Assert.java AssertionFailedError ComparisonFailure.java Protectable Test TestCase TestFailure TestListener TestResult TestSuite	34 35 36 37 38 39 40 41 42 43	136 137 138 143 144 139 140 145 141 142	44 45 46 47 48 49 50 51 52 53
Runner	BaseTestRunner ClassPathTestCollector FailureDetailView LoadingTestCollector ReloadingTestSuiteLoader SimpleTestCollector Sorter StandardTestSuiteLoader TestCaseClassLoader TestCollector TestRunListener TestSuiteLoader Version	21 22 23 24 25 26 27 28 29 30 31 32 33	123 124 132 125 126 127 128 129 130 133 134 135 131	54 55 56 57 58 59 60 61 62 63 64 65 66
Samples	VectorTest.java AllTest.java SampleTest.java	20	146	67 68 69
Textui	ResultPrinter.java TestRunner.java	18 19	122 121	70 71

Figure 35 Executed junit and file id for each execution that was used in 3 separate experiments

Next, in **Table 5**, the executed line Numbers are shown for each file, indicating which statements were executed for the each identified feature (F1, ...F7) . From **Table 5** we can also see that, each feature includes statements from several files. For instance Feature

F1 contains executed statements from 7 files with file id 18, 19, 20, 21, 34, 39, 43 and 43. We can also see that the features are of different sizes, and some files are necessary for computing all the features such as files with id 18, 19, 41, 42. The number of files included in a feature is often different for computing the features. For instance, Feature F1 contains 7 files, but Feature F2 comprises of 4 files.

Feature	**File id**	**Executed line number**
F1	18	[14, 18, 20, 104, 105, 130, 131]
	19	[25, 41, 42, 88, 89, 96, 97, 98, 100, 120, 121, 122, 123, 137, 138, 139, 155, 156, 157]
	20	[10, 21, 22]
	21	[13, 83, 100, 112, 113, 126]
	34	[7]
	39	[70, 117, 118]
	42	[15, 22, 23, 24, 25, 26, 27, 54, 55, 66, 67, 105, 106, 151, 152, 157, 158, 162, 163, 164, 180]
	43	[30, 81, 82, 224, 225, 226, 227, 229, 230, 234, 235, 255, 256]
F2	18	[14, 18, 20, 26, 27, 41, 42, 104, 105]
	19	[25, 41, 42, 88, 89, 96, 97, 98, 99, 100, 101, 102, 103, 120, 121, 122, 123, 137, 138, 139, 155, 156, 157]
	21	[13, 83, 100, 112, 113, 126]
	42	[15, 22, 23, 24, 25, 26, 27, 54, 55, 180]
F3	18	[14, 18, 20, 26, 27, 41, 43, 100, 101, 104, 105]
	19	[25, 41, 42, 88, 89, 96, 97, 98, 99, 100, 101, 102, 103, 120, 121, 122, 123, 137, 138, 139, 155, 156, 157]
	21	[13, 83, 100, 112, 113, 126]
	42	[15, 22, 23, 24, 25, 26, 27, 54, 55, 180]
F4	18	[14, 18, 20, 26, 28, 29, 30, 46, 47, 50, 51, 54, 55, 56, 81, 82, 83, 104, 105]
	19	[25, 41, 42, 88, 89, 96, 97, 98, 99, 100, 101, 102, 103, 120, 121, 122, 123, 137, 138, 139, 155, 156, 157]
	21	[13, 83, 100, 112, 113, 126]
	42	[15, 22, 23, 24, 25, 26, 27, 54, 55, 81, 82, 87, 88, 93, 94, 99, 100, 180]

Table 5 The computed features [Continued]

Feature	File id	Executed line number
F5	18	[14, 18, 20, 26, 28, 29, 30, 46, 47, 50, 51, 54, 55, 56, 81, 82, 84, 104, 105]
	19	[25, 41, 42, 88, 89, 96, 97, 98, 99, 100, 101, 102, 103, 120, 121, 122, 123, 137, 138, 139, 155, 156, 157]
	21	[13, 83, 100, 112, 113, 126]
	42	[15, 22, 23, 24, 25, 26, 27, 54, 55, 81, 82, 87, 88, 93, 94, 99, 100, 180]
F6	18	[14, 18, 20, 26, 28, 29, 30, 46, 47, 50, 51, 54, 55, 56, 81, 82, 85, 92, 104, 105]
	19	[25, 41, 42, 88, 89, 96, 97, 98, 99, 100, 101, 102, 103, 120, 121, 122, 123, 137, 138, 139, 155, 156, 157]
	20	[10, 21, 22]
	21	[13, 83, 100, 112, 113, 126]
	34	[7]
	39	[70, 117, 118]
	42	[15, 22, 23, 24, 25, 26, 27, 54, 55, 81, 82, 87, 88, 93, 94, 99, 100, 105, 106, 128, 129, 151, 152, 157, 158, 159, 160, 176, 177, 180]
	43	[30, 81, 82, 224, 225, 226, 227, 229, 230, 234, 235, 255, 256]
F7	18	[14, 18, 20, 26, 28, 29, 30, 46, 47, 50, 51, 54, 55, 56, 81, 82, 85, 92, 104, 105]
	19	[25, 41, 42, 88, 89, 96, 97, 98, 99, 100, 101, 102, 103, 120, 121, 122, 123, 137, 138, 139, 155, 156, 157]
	21	[13, 83, 100, 112, 113, 126]
	42	[15, 22, 23, 24, 25, 26, 27, 54, 55, 81, 82, 87, 88, 93, 94, 99, 100, 176, 177, 180]

Table 5 The computed features

Execution time

The initial experiment shows that there exist seven executed output statements in the project JUnit. Also, seven output features were identified (indicated by F1..F7). Since there is no input statement, no input feature could be extracted for that case study. The following table shows the computation time to compute the output feature slices. **Table 6** contains columns such as feature, slice type, slicing criterion and so on. Here the "Feature" column represents the name assigned to each computed feature. The feature

type column indicates whether the feature is an input or output feature. The slicing criterion column indicates whether the feature is computed based on a variable or method call as the slicing criterion. In addition, each input or output statement may or may not contain any variable. When there is an input or output statement, variable defined in the statement as the slicing criterion for calling the backward or forward algorithm from the feature extraction algorithm. However, when an I/O statement does not contain any variable, but includes a method call to perform some computation, when calling the backward or forward algorithm from the feature extraction algorithm, the method call position is given as the slicing criterion. Next, the data and control dependencies that lead to the method call are computed.

Feature	**Feature type**	**Slicing criterion**	**Code size (Line of code)**	**Execution Position in the execution trace**	**Number of classes included in the feature**	**Feature computation time in second**
F1	Output feature	Method call	73	440	8	1.344
F2	Output feature	Method call	48	503	4	1.235
F3	Output feature	Variable	50	506	4	1.282
F4	Output Feature	Method call	66	540	4	1.750
F5	Output Feature	Method call	66	543	4	1.469
F6	Output Feature	Variable	99	546	8	1.906
F7	Output Feature	Method call	69	553	4	1.781

Table 6 The result obtained from the JUnit when applying feature extraction algorithm

One observation from the results in **Table 6** is that there seems that no direct relationship between the computation time and the slice size. For instance, although feature F1

consists of 73 lines of code, the computation time is shorter compared to the other slices of similar size. After a closer analysis, one can identify that there exist several factors which may contribute to the computation time:

- The slice size and the number of files in which these statements are included. The number of files in general influences the computation time because often a file corresponds to a separate method/class and therefore to separate function calls. The underlying slicing algorithm requires some additional computational overhead for the computation of slices for function calls. For instance, in the method "markActionAsContributing" of the dynamic backward slicing algorithm, the given position is marked as contributing at first. Next, a check is performed to see whether the statement is a function call such as a constructor or method call. If it is, then when a statement includes a method call, the statement is marked as contributing. Next, in the case of a method call, the return statement from the called method has to be marked as contributing too. Moreover, when the slice is in a separate file, and if the file includes a class definition, then the class name, and the constructor needs to be added in the slice to make the slice executable.

  – The position in the execution trace of the particular slicing criteria has a direct impact on the slice computation time. For instance, the backward slicing algorithm starts from a given position in the execution trace and identifies the last redefinition of the slicing criteria in the execution by traversing the trace backwards. Therefore, the more statements the algorithm has to consider in the evaluation and computation process, the longer the computation time. Hence, if the execution position is at the end of the execution trace, the

algorithm has to consider all the statements of the trace in the computation process of the slice. It also has a direct effect on the slicing computation time.

- From Table 6, we can see that feature F1 is computed with the slicing criterion of execution position 440. The computation time for the slice is 1.344 sec, although the time to compute the slice should be comparably longer because the slice included a large number of statements (73 statements) from 8 files of the project. On the other hand, the next feature is executed after 63 positions in the execution trace. Hence, although feature F2 does not have many files to access as it contains only 4 files and the slice size is rather small containing 48 statements, the computation time des not differ greatly from that of Feature 1. However, we can still see that Feature 2 has less computation time 1.235 sec compared to 1.34 sec for Feature 1.

Second Test Case

The illustrated VectorTest class shown in **Figure 33** returns a true result by the ResultPrinter class of JUnit because the vector represented in this class contains the values 1,2,3, and when there was a test to see whether the vector contains 1, it returns true. In order to see the result from a class that returns a failure result, the VectorTest class has been changed so that now the test clone methods test whether integer 4 is an element of the vector as illustrated in **Figure 36**. A test failure is produced by the TestRunner class because Integer 4 is not included in the vector as shown in **Figure 37.**

```
.....
......
18. public static Test suite() {
19. return new TestSuite(VectorTest.class);
20. }
21. public void testClone() {
22. Vector clone= (Vector)fFull.clone();
23. assertTrue(clone.size() == fFull.size());
24. assertTrue(clone.contains(new Integer(4)));
25. }
26. }
```

Figure 36 Partial listing of modified VectorTest class

Figure 37 Output produced by the modified Vector Test class

This example does not have any input statement so only output features were computed. The numbers of executed output features are 19. From the identified output features some of these are only informational features so we did not compute them.

```
The Slice is computed for method call own method: and for information
java.io.PrintStream junit.textui.ResultPrinter.getWriter() at
D:\junitwithfailurefeature\junit\textui\ResultPrinter.java L:42,O:1063 in file 122 in
position 42
The taken time is 2234 millisecond
File ID Executed Line Nbr
121   [25, 41, 42, 88, 89, 96, 97, 98, 99, 100, 101, 102, 103, 120, 121, 122, 123, 124,
137, 138, 139, 155, 156, 157]
122   [14, 18, 20, 26, 27, 28, 29, 30, 41, 42, 46, 47, 50, 51, 54, 55, 56, 57, 58, 61, 62, 66,
67, 71, 74, 81, 82, 104, 105]
123   [13]
136   [7, 12, 13, 206, 207, 213, 214]
139   [70, 117, 118, 124, 125, 126, 127, 137, 138, 139, 140, 145, 149, 153, 154, 194,
195]
140   [12, 20, 21, 22, 27, 28]
141   [15, 22, 23, 24, 25, 26, 27, 44, 45, 54, 55, 66, 67, 81, 82, 87, 88, 93, 94, 99, 100,
105, 106, 119, 121, 134, 135, 136, 138, 139, 157, 158, 162, 163, 164, 176, 177, 180, 181,
182]
142   [30, 81, 84, 85, 96, 97, 98, 99, 100, 101, 103, 130, 131, 132, 134, 135, 137, 140,
147, 148, 155, 156, 157, 158, 159, 200, 201, 202, 203, 207, 210, 211, 214, 215, 216, 217,
218]
146   [10, 14, 15, 16, 17, 18, 19, 21, 22]
```

Figure 38 A feature extracted by the sample program

The result from this test case show that among 27 classes, only 12 classes were used for computing the feature for method getwriter().The name of the classes which were executed are TestRunner, ResultPrinter, BaseTestRunner, Assert, TestCase. TestFailure,, TestResult,TestSuite,VectorTest. The TestFailure is executed because the given test case fails because of not having 4 in the vector. The TestRunner class is the tool for running the JUnit, and the class ResultPrinter prints the result to the user.

Third TestCase

Since both of the above examples did not have any input feature, another test case was added to see the applicability of the extracted input features. In the new test case, 2 input statements are provided as shown in **Figure 39**. The output displayed in the console

based on the AllTest test cases is shown in **Figure 40.** **Figure 41** shows a computed input feature from the given test case.

```
public class AllTest  extends TestCase {

  protected static String input;
  protected static String inputSampleTest;

  public static Test suite() {

    TestSuite suite = new TestSuite();
    System.out.println("Do you want to test the VectorTest class? Enter Y or y to accept ");
    try{
    BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
    input = in.readLine();
    }
    catch(IOException exception){
      System.out.println(exception);
    }
    if(input.equalsIgnoreCase("Y"))
     suite.addTestSuite(VectorTest.class);

    System.out.println("Do you want to test the SampleTest class? Enter Y or y to accept ");

    try{
    BufferedReader readinput = new BufferedReader(new InputStreamReader(System.in));
    inputSampleTest = readinput.readLine();
    }
    catch(IOException exception){
      System.out.println(exception);
    }
    if(inputSampleTest.equalsIgnoreCase("y"))
    suite.addTestSuite(SampleTest.class);
    return suite;
  }
}
```

Figure 39 Test case with input feature

D:\WINNT\System32\cmd.exe

Figure 40 Output from the AllTest class

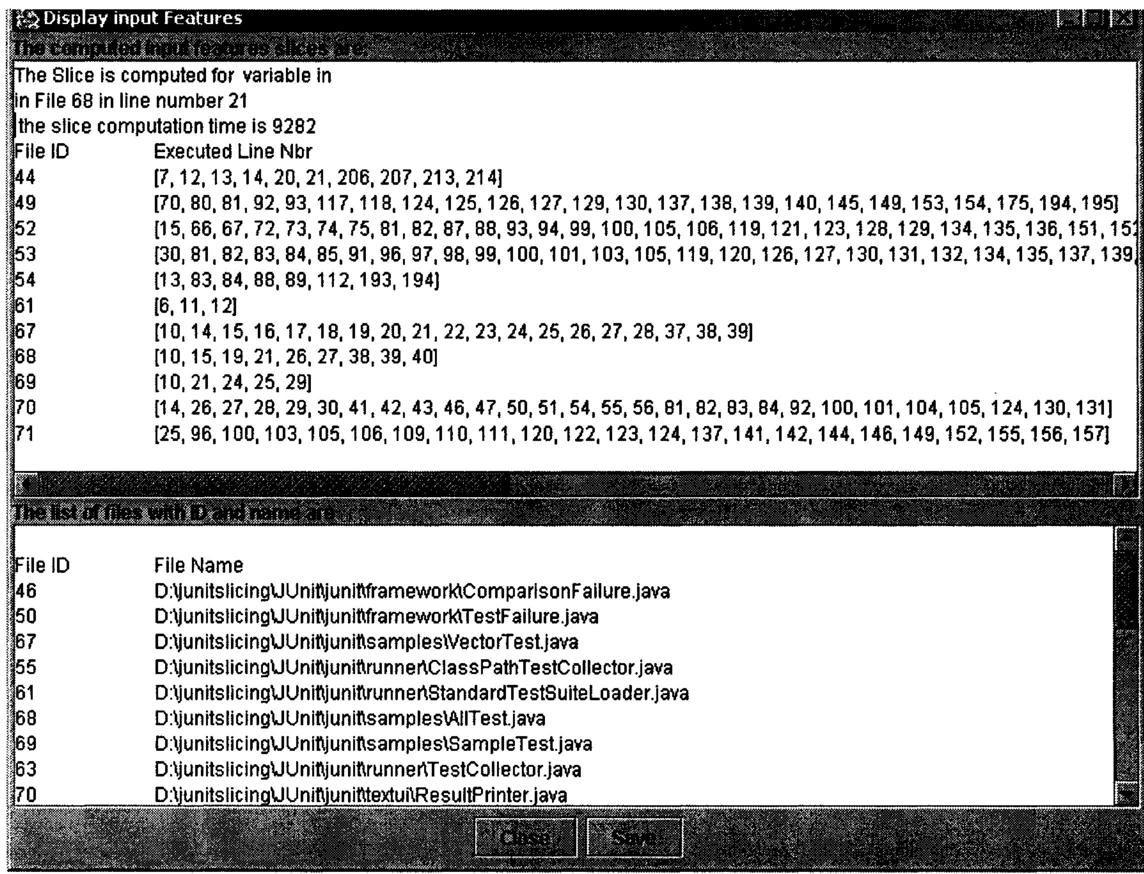


Figure 41 An example of an input feature

The classes executed for this feature are Assert, TestCase, TestResult, TestSuite, BaseTestRunner, StandardTestSuiteLoader, VectorTest,AllTest, SampleTesResultPrinter and TestRunner. Hence, this input feature provides us the information that all these classes are affected by the execution of this input statement, and computations such as displaying the test result is performed by the other executed classes.

4.4 Application of extracted features

As customer requirements are changing, software systems need to evolve and features in software systems may need to be deleted, added, modified or extended [Mul01, Eis01].

Our semi automatic feature extraction approach can provide users with additional insights about a system by extracting functional features from the source code. Identifying these features can be an important aspect of source code comprehension [Kan90]. This is because it may identify clusters of source code relevant to certain domain aspects where all the features of the system are not known.

Other applications of the proposed feature extraction algorithm can be found, for example, in debugging. Often a bug can be associated with a particular program feature. By extracting features and removing these parts of the program that are not relevant for the implementation of a particular feature, a user can focus on those parts of the program which most likely will contain the fault.

Feature extraction can also be used for measuring the interaction (coupling) among different features. There exist several definitions that introduce the notion of coupling. For instance, Stevens et al. [Ste74] defined coupling as "the measure of the strength of association established by a connection of one module to another." Next, Constantine and Yourdon [You79] defined coupling based on the relationship of subroutines as measurements for procedural systems. The commonality among these traditional coupling definitions is that they are applied on the function level. However, we have applied an extended definition of coupling introduced in [Ril04] as the source code dependencies between features. This approach can be seen as an extension of the traditional measurements by including not only the call dependencies at the function or class level but also on the feature level.

When a feature is implemented by many software functions the resulting code is often highly coupled [Meh01c]. Ideally, software should follow certain design standards that

require a high degree of cohesion and a low degree of coupling amongst its various components. In practice, however, this case rarely happens. In fact, one may find that in a large, complicated software system, a program feature is spread across a number of components. Code used to implement a feature may be found in components which are seemingly unrelated to each other. If a single feature is implemented within a single object or method, it corresponds to high cohesion. If two features overlap, the program has high coupling. In addition the intersected parts are a crucial part of the program as it is used by multiple functional requirements of the system. If the intersected part can be separated, this part could be reused as a separate entity for the purpose of being used by the system. **Figure 42** shows an example of the use of features in measuring the source code coupling between features. Here, the two features, *F1* and *F2,* have overlapping in their code. Both features *F1* and *F2* are sharing some functions or classes. If these functions or classes are spread over several components, then the same code segment that is shared between multiple components should be extracted as a new class or function for reuse. The common part between *F1* and *F2* represents high coupling between features.

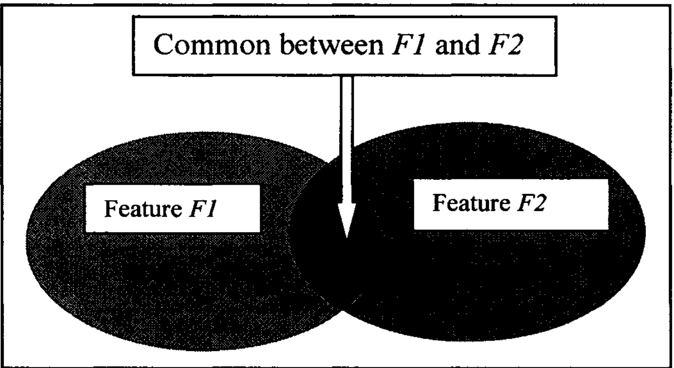


Figure 42 Measuring coupling and cohesion of the system

The following **Table 7** shows the identified features and their overlap for JUnit with the first test example. This was the test case with the Vector Test class. The rows show the feature (identified by a feature number), the number of statements and percentage of overlap among the specific features with other identified features in JUnit. For instance, the row F1, and column F2 includes the value 39 and 53.4. This means that feature F1 overlaps with feature F2 for 53.4% of F1's statements.

From Table 7, it can be observed that the identified features within the JUnit have some commonalities with each other. In what follows we discuss some of the observations and their potential impact on the comprehension process.

1) The common files in the features can be used to provide some ranking of the source code to identify these parts of the system that are commonly used by most of the features and therefore most likely provide some core functionality. For instance the classes VectorTest.java,TestRunner.java, BaseTestRunner.java and ResultPrinter class are used by all identified features.
2) The feature overlapping shows the dependencies (coupling) among different features and the intention of good system design should be to reduce coupling between features so that any feature can be modified without having any side effect on other features. From our case study, Feature2 is sharing most of the statements of Feature1, causing high coupling between Feature1 and Feature2. If there exist more than one feature to be evolved, then it is important to evaluate the relationship between them.

Feature		F1	F2	F3	F4	F5	F6	F7
F1	Overlap size	73	39	41	40	40	66	40
	%	**100**	**53.4**	**56.2**	**54.8**	**54.8**	**90.4**	**54.8**
F2	Overlap size	39	48	46	46	46	45	45
	%	**81.3**	**100**	**95.8**	**95.8**	**95.8**	**93.8**	**93.8**
F3	Overlap size	41	46	50	43	43	45	45
	%	**82**	**92**	**100**	**86**	**86**	**90**	**90**
F4	Overlap size	40	46	43	66	57	65	66
	%	**60.6**	**69.7**	**65.2**	**100**	**86.4**	**98.5**	**100**
F5	Overlap size	40	46	43	57	66	64	65
	%	**60.6**	**69.7**	**65.2**	**86.4**	**100**	**97**	**98.5**
F6	Overlap size	66	45	45	65	64	99	69
	%	**66.7**	**45.5**	**45.5**	**65.7**	**64.7**	**100**	**69.7**
F7	Overlap Size	40	45	45	66	65	69	69
		58	**65.2**	**65.2**	**95.7**	**94.2**	**100**	**100**

Table 7 Features overlapping percentage from JUnit

Table 8 shows the set of statements (identified by a file-id and statement number) that correspond to Feature1 and Feature 2, and their common statements (indicated in bold).

Feature 1 (F1) Feature slice size is 73

File ID	Executed Line Nbr
18	[**14,18, 20, 104, 105**, 130, 131]
19	**[25, 41, 42, 88, 89, 96, 97, 98, 100, 120, 121, 122, 123, 137, 138, 139, 155, 156, 157]**
20	[10, 21, 22]
21	**[13, 83, 100, 112, 113, 126]**
34	[7]
39	[70, 117, 118]
42	[**15, 22, 23, 24, 25, 26, 27, 54, 55**, 66, 67, 105, 106, 151, 152, 157, 158, 162, 163, 164, **180**]
43	[30, 81, 82, 224, 225, 226, 227, 229, 230, 23 , 255, 256]

Feature 2 (F2) Feature slice size is 48

File ID	Executed Line Nbr
18	[**14, 18, 20**, 26, 27, 41, 42, **104, 105**]
19	**[25, 41, 42, 88, 89, 96, 97, 98, 99, 100, 101, 102, 103, 120, 121, 122, 123, 137, 139, 155, 156, 157]**
21	**[13, 83, 100, 112, 113, 126]**
42	**[15, 22, 23, 24, 25, 26, 27, 54, 55, 180]**

Overlapped statements between F1 and F2 Slice size is 39

File ID	Executed Line Nbr
18	**[14, 18, 20, 104, 105]**
19	**[25, 41, 42, 88, 89, 96, 97, 98, 99, 100, 101, 102, 103, 120, 121, 122, 123, 137, 139, 155, 156, 157]**
21	**[13, 83, 100, 112, 113, 126]**
42	**[15, 22, 23, 24, 25, 26, 27, 54, 55, 180]**

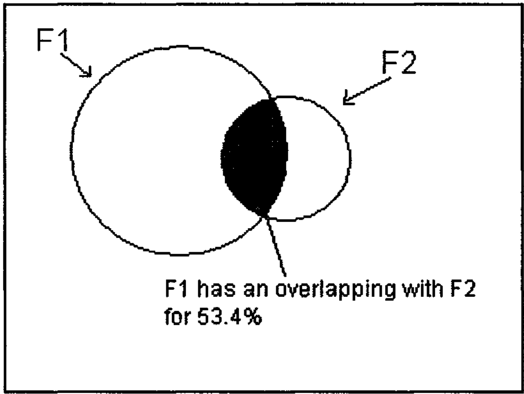
Table 8 The result of Feature 1 and Feature 2 overlapping

The overlap among features can be computed as follows:

The formula used is: ***(overlapped slice size/original slice size) *100***
Example:

Feature1 has original slice size of 73
Feature2 has original slice size of 48
Feature1 overlaps with Feature2 for 39 statements
Feature1 overlaps with Feature2 for (39/73)*100 =53.4%
Feature2 overlaps with Feature1 for 39 statements
Feature2 overlaps with Feature1 for (39/48)*100 = 81.3%

The overlap among features can also graphically be represented in this example; the feature overlap would be the intersection among the two features (indicated in grey)



Overlapping between F1 and F2

Figure 43 The overlapping computation formula and overlapping example

Another application of the feature extraction technique is in testing and creation of test cases. Each executable feature can be tested separately as they can be executed as a separate program.

A product normally evolves during different releases by adding, modifying or changing features from one release to another.

The feature extraction can provide some additional guidance to identify, compare changes that were made to a system with respect to its features. **Figure 44** shows a summary of the various applications of the feature extraction technique used in this research.

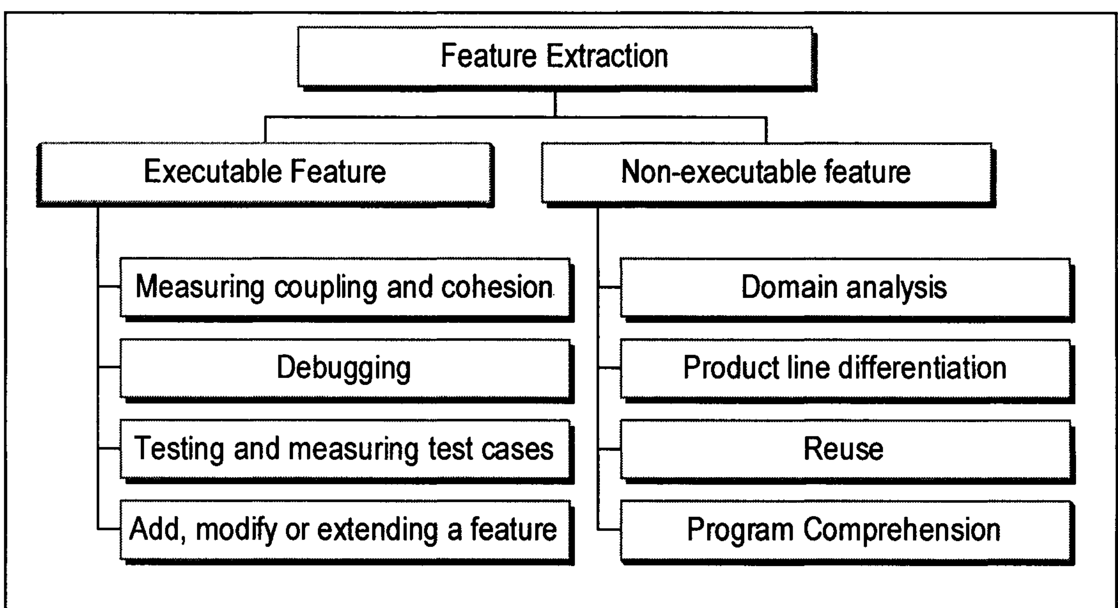


Figure 44 Applications of feature extraction technique

Limitations and assumptions of the presented approach

The following is a summary describing the limitations of the presented feature extraction approach:

- The algorithm extracts only the statement numbers which corresponds to a feature, rather than the associated source code. The feature extraction algorithm uses the instr_trace program for creating execution traces. However, the dynamic information provided by the instr_trace program is minimal as for each executed source code line, the output consist only the file name and the line number by the

instr_trace. Hence, the feature extraction algorithm has to depend on the static information generated by the parser although it uses the "dynamic" program slicing algorithms.

- Output statements that are only displaying a string literal are considered as an informational feature. In addition, the algorithm does not include these information features in the feature extraction.
- The dynamic slicing algorithms are conservative, by computing slices larger than necessary. This is caused in situations when the algorithm identifies an action as neutral (the execution of the statement can not be identified at that point of time as being contributing or non-contributing to the computation of the slice). Based on the conservatism, the algorithm considers all of these neutral actions automatically as contributing actions (includes them in the slice).
- Problem occurs when several contributing actions contain the same method call. In the given algorithm, if a contributing action contains a method call, the return statement from the method call is also marked as a contributing action. The rule for computing non-contributing action is when there is any action that is not marked as contributing, find all the actions between S *(B, p, p1)* where *p* is the block-entry of block *B*, and *p1* is the block exit of block *B* that contains no contributing actions. Finally, the actions that are not marked as contributing or noncontributing, are being marked as neutral. All the neutral actions become contributing in the next iteration. Hence, when several procedures are calling the same method call, the return statement of the method is already marked as contributing by the first action that calls it. The reason is each action containing

the same method call will have the same block exit. Hence, for example, if there is a method call such as *f()* in any contributing action x^k, the return statement of *f()* is being marked as the last definition of x^k. When the non-contributing actions method is called, if action x^n is not marked as contributing yet and it contains the same method call *m( ),* it will not be marked as non-contributing. The reason is the block exit of x^n is in the return statement of *m()* which is already marked as contributing by the computation of x^k. Hence, the action x^n becomes neutral although it may not contribute to the computation. As a result, the action x^n is becoming a contributing action. This situation is shown in **Figure 45**. In this example statement 92 corresponds to feature F6, and statement 85 corresponds to feature F6. From this figure we can see that, feature F7 should not have statement 92 because statement 92 does not affect the value of statement 85. However, the feature includes statement 92 because statement 85 and statement 92 has the same method call "getWriter". When statement 85 is a contributing action, the return statement of "getWriter", which is statement 105 is also marked. When the method non-contributing action method is called, the action with statement 92 is not marked as contributing because the return statement or the block exit of 92 is already marked as contributing as the block exit of statement 92 is statement 105 again. Hence, statement 92 becomes a neutral action which is ultimately a contributing action in the repeat loop. From **Table 7** we can see that Feature 7 overlaps with Feature 6 100%, however it has to be noted that some statements are included in the slice because of the limitations of our slicing algorithm and therefore causes an imprecision in the interpretation of the results.

```
81. protected void printFooter(TestResult result) {
82.       if (result.wasSuccessful()) {
83.          getWriter().println();
84.          getWriter().print("OK");
85.          getWriter().println(" (" + result.runCount() + " test" + (result.runCount() == 1 ? "": "s") +
    ")"); //F7
86.
87.       } else {
88.          getWriter().println();
89.          getWriter().println("FAILURES!!!");
90.          getWriter().println("Tests run: " + result.runCount() + ",  Failures: " + result.failureCount()
    + ",  Errors: " + result.errorCount());
91.       }
92.       getWriter().println(); // F6
93.    }
94.
95.
96.    /**
97.     * Returns the formatted string of the elapsed time.
98.     * Duplicated from BaseTestRunner. Fix it.
99.     */
100.   protected String elapsedTimeAsString(long runTime) {
101.      return NumberFormat.getInstance().format((double)runTime/1000);
102.   }
103.
104.   public PrintStream getWriter() {
105.      return fWriter;
106.}
```

Figure 45 Partial listing of ResultPrinter

- The parser has several limitations with respect to its ability to parse and extract static information. The limitations include:
- The parser is using javac which can extract information from Java programs only. More specifically, the current implementation will not be able to handle the new features and additions included in the upcoming Java version 1.5.
- The input and output statements are not parsed initially by the parser. The AST created by the parser does not identify input and output statements. Rather, input and output statements are later stored in a data structure component using the

inputStatements and outputStatements classes by identifying the usage references which corresponds to the java.io package.

- Another limitation of the parser identified when implementing feature extraction algorithm involves objects. When a method is invoked on an object, the parser cannot determine whether or not new values are defined for its instance variables [Cha04]. The only information it provides is that the object is used. Hence, when the backward slicing algorithm is called for computing an output feature, the computed feature might contain some irrelevant statements that did not modify the program. For instance, in **Figure 46** if the last definition of action 26^6 has to be found then action 24^4 is going to be part of the slice even though this action is not modifying the value of the balance, and it is just printing the instance variable balance.

Execution Trace	Action State
...	
21^1 Account a = new Account();	Contributing
24^4 a.printbalance();	Contributing
26^6 a.getbalance();	Contributing

Figure 46 Example of the limitation with the invoking objects

- Currently, the input and output criteria are only based on the java.io package. The input and output criteria do not include any method call of the javax.swing package. This limitation is caused by the inability of the parser to parse the GUI libraries of Java (e.g. AWT, Swing).

4.5 Related work

The existing non source code based techniques [Amy00, Kan90, and Won99] for feature extraction do not extract the source code that implements functional system requirement (feature). For example, the approach presented in [Amy00] uses Use Case Maps [Kan90] uses feature oriented domain analysis and use cases to provide a high level view of a system's overall architecture and its features. The approach does not consider implementation details. In comparison, our approach focuses on the feature extraction at the implementation (source code) level. Both approaches (source code and non-source code based techniques) have their advantages and disadvantages and different application areas.

- Extracting features using the non source code based techniques has the advantage that they provide a better high-level understanding of the system functionality. Hence, for identifying some of the core functionalities of a system these approaches are particularly useful. There also has been work on requirements gathering techniques such as from requirement and specification document [Won99], which would appear to lend itself to the identification of features with requirements specifications. But they do not address the question of how the features would be reflected in life-cycle artifacts other than requirement specifications and in a restricted form of design prototypes. Another disadvantage is that these documents might not always be complete. Moreover, the implementation of the system might not conform to the original document of the system. Making any changes to a feature solely based on these documented methods may corrupt the original source code as the documents may not reflect

the changes which occurred due to maintenance. Our feature extraction tried to avoid these problems of documentation based methods by locating the features on the source code level.

- Use cases capture functionalities of a system according to user requirements. Use cases focus on how the system is perceived from the outside by the user and therefore use cases are the most useful approach in discussions with end users to make sure that the requirement of the system will meet the end users demand. One of their shortcomings is that both developers and maintainers will have to identify and locate the statements manually that implement these use cases.
- During domain analysis the users' requirements of the system are captured. However, when a feature needs to be modified, even if the domain level information is provided, one still has to identify the source code that implements the feature. The feature oriented domain analysis affects the maintainability, understanding ability, and reusability characteristics of a system or family of systems. Nonetheless, it does not address the issue about what percentage of future changes need to be done if the software needs to be evolved [Meh01b].

In contrast, our feature extraction technique identifies features on the source code level and therefore provides for a more detailed analysis of the features (with respect to their implementation in the source code). However, the disadvantage of our approach is that modifications based mainly on solution domain might vary from the end users needs. The reason is that implementation usually includes the technical details of the system, and evolving the system solely based on the technical merits may miss the connection with the end user [Meh01c]. Since we are only doing the source code analysis, the upgrade of

the system may differ from what the customers wanted. This is because there is no communication between the customer and the developer anymore that may cause changes unacceptable to the customer.

Source code based feature extraction

The existing source code based feature extraction techniques [Meh01a, Eis01a, and Wil95] include concept analysis, test cases, software reconnaissance etc. These techniques compute a single feature with either the aid of test cases, or with the static and dynamic analysis, and by creating scenarios for extracting features.

We need some program execution which corresponds to the input to execute the program. Later input and output statements in the same execution have to be identified to be able to extract features. The reason we are using dynamic slicing is that the computed slice is relatively small compared to the slice computed by static analysis. The existing source code based techniques, and our feature extraction techniques both have advantage and disadvantage. These are illustrated in the following:

- Wilde and Scully [Wil95] used software reconnaissance for analyzing features. The advantage of the extracted features using this method is that there is a mapping between the user requirements according to the problem domain, and the corresponding source code in the program. If proper test cases are given, and the implementation can be identified correctly, there is an assurance that the user requirement conform to the implementation of the user requirement. However the disadvantage of this method is that test cases might not be

complete. It is not always possible to know what group of test cases will exercise a given feature [Meh01].

Directories	VectorTest true result (TestCase 1)	Vector Test False Result(Test Case 2)
Samples	VectorTest	VectorTest
Framework	Assert TestCase TestResult TestSuite	Assert TestCase TestResult TestSuite TestFailure
Runner	BaseTestRunner	BaseTestRunner
Textui	TestRunner ResultPrinter	TestRunner ResultPrinter

Figure 47 Differences between two features identified with different test cases

Figure 47 shows that we have used two Vector test classes one for testing the true result produced by the test case and one for testing the failure result. The class level information from the computed features using these two test cases shows that the true and false feature includes the same classes with the exception of the TestFailure class that is used to compute the failure feature. Hence, by looking at the difference one can identify that this TestFailure class is specifically computing a feature that corresponds to the failure result from the JUnit. Hence, our method can provide an insight for the similar functionality as the software reconnaissance method if different test cases are provided, even though our computation technique is different than the software reconnaissance method.

Concept analysis was used by Eisenbarth [Eis01] for identifying the dependencies among features. The approach requires dynamic and static analysis and the creation of scenarios

based on test cases to get the binary relationship between the scenarios and components. The disadvantage is in the interpretation of the identified concepts. They are mainly identified by common usage rather then identifying data and control flow that corresponds to feature. Also the notion of a feature differs in the context that concept analysis interprets a feature as some execution sequence that is shared among other parts of a program.

- The feature extraction techniques using test cases [Meh01a, Meh02], and software reconnaissance [Wil95] did not discuss about the software evolution problem. In contrast, from the case study and as discussed previously, we also see that our feature extraction technique can aid in software evolution by analyzing the relationships between features. For instance, if several features are sharing some common statements, then when enhancing one feature, another identified feature can be taken into account so that the other feature does not get any undesirable affect. However, our feature extraction technique needs to create the execution trace. Also, input and output statements are needed to be identified.
- Wilde and Scully [wil95] focus on localizing required components for one specific feature rather than analyzing commonalities and variability of related features. Conversely, by measuring the overlapping between feature features, we can identify what percentage of a feature is common to the other feature. The derived commonalities and variability are important information to a maintainer who needs to understand the system.

The presented feature extraction technique bridges the gap between the problem and the solution domain by mapping the features that the end user sees using the input and output

statements used in the program, and the functions in the source code that a developer sees. Moreover, in our feature extraction algorithm, we are identifying the source code associated with a feature. We are concentrating in the software implementation of the system. The extracted code can be analyzed, and if there is dependency with any other feature, the other feature should be analyzed. This reduces the task of searching the whole program for the associated source code of the program. However, the disadvantage is that the source code might not corresponds to exactly what the user wanted initially, and that any extension based on the current implementation might not reflect to user needs. Moreover, features might be computed based on function calls too instead of an input and output statements only.

5 Conclusions and future work

The main motivation of this research work is to support programmers performing typical program comprehension and software maintenance tasks. During the maintenance phase of a software product, many of the required changes are performed on a feature level, with features either being added, modified or deleted, in order to respond to market needs. Programmers tend to focus on only those parts of the system to save time when creating a mental model of the system. It is almost infeasible to enhance or add a software feature without comprehending the source code of the application [Mul01].

The majority of existing techniques [Wil95, Eis01a, Eis01b, Buh95, Meh02, and Meh01a] are based on the assumption that several resources are available for the feature extraction, including source code, domain experts and/or test cases. In this research, a survey and categorization of these existing techniques for feature extraction was provided. Within the context of this research a refined feature definition based on user perspective was introduced. A source code based feature extraction algorithm based slicing of input and an output statement of a program was presented and a Java implementation that was integrated within the CONCEPT environment provided. We conducted a case study for the JUnit, an open source testing framework for which several output features were identified and computed. From the experimental result, we observed the relationships between features such as data dependencies or method dependencies.

Contribution

The contribution of this thesis can be summarized as follows:

- A refined definition of the notion of a feature was provided.

- A feature categorized based on their properties was introduced
- Input statements and output statements at the source code level were identified
- A modified version [Cha04] of Korel's [Kor94] dynamic backward slicing algorithm was utilized as the general algorithm to compute program slices.
- A forward slicing algorithm has been implemented by extending the backward slicing algorithm [Ril98, Cha04, Kor88]. It combines computing slices using forward and backward directions of data and control dependencies to identify the statements that will be affected by the slicing criteria. The forward slicing algorithm also allows making the extracted features executable. When computing slices using the forward algorithm, initially the slices are computed by identifying statements that are affected by the slicing criterion in the forward direction. Next, in order to make the slice executable necessary statements are included in the slice by computing the statements from backward direction. This leads to the statements which affects the value of the variable of the slicing criterion.
- The feature extraction algorithm has been implemented within the CONCEPT (Comprehension Of Net-Centred Programs and Techniques) framework [Ril01]. As discussed previously, Concept is a reverse engineering environment which stores both static and dynamic information into the database. The information is later used in different applications for comprehending a system. For our feature extraction algorithm, static information is obtained by the parser. Next, the execution trace is created using instrumentation and the execution trace is sussequently stored in the database.

Future work

However, several main challenges remain that go beyond just extracting features. They are included in the following:

- The slicing algorithm should be less conservative to improve the precision of the algorithm (slice size).
- Additional experiments with larger projects that also contain different type of input/output statements are needed.
- Another improvement would be to extract the source code associated with a feature rather than only the statement number and extract automatically executable features from the source code.
- Static algorithm for feature extraction should be implemented which will allow for an automatic feature extraction. However, static slicing algorithms are required for static feature extraction technique including all advantage and disadvantages associated with static slicing. The advantage of static slicing is that it would be cheaper, but the disadvantage is that the slicing technique will produce less precise result.
- The parsing environment will have to be extended to included the ability to support GUI components

Bibliography

[Agr90] H. Agrawal and J. Horgan, "Dynamic program slicing", In *Proceedings of the ACM SIGPLAN '90 Conference*, 1990.

[Agr94] H. Agrawal, "On Slicing programs with jump statements", In *proceedings of the ACM SIGPLAN'94 Conference on Programming Language Design and Implementation,* pp. 112-135, 1994.

[Agr98] H. Agrawal, J. L. Alberi, J. R. Horgan, J. J. Li, S. London, W. E. Wong, S. Ghosh and N. Wilde, "Mining system tests to aid software maintenance", *IEEE Computer Society*, pp. 64-73, July 1998.

[Amy00a] D. Amyot, "Use Case Maps as a Feature Description Notation", In *FIREworks Feature Constructs Workshop*, Glasgow, Scotland, UK, May 2000.

[Amy00b] D. Amyot, L. Charfi, N. Gorse, T. Gray, Logrippo, J. Sincennes, B. Stepien and T. Ware, "Feature Description and Feature Interaction Analysis with Use Case Maps and LOTOS", In *Sixth International Workshop on Feature Interactions in Telecommunications and Software Systems (FIW'00)*, Glasgow, Scotland, UK, May 2000.

[Amy02] D. Amyot, G. Mussbacher and N. Mansurov, "Understanding Existing Software with Use Case Map Scenarios", In *3rd SDL and MSC Workshop (SAM'02)*, Aberystwyth, U.K., June 2002.

[Bir40] G. Birkoff. "Lattice Theory", Americal Mathematical Society, 1940.

[Boe81] W.B. Boehm, "Software Engineering Economics", Prentice Hall, Englewood Cliffs, NJ, 1981.

[Bro83] R. Brooks,"Towards a theory of the comprehension of computer programs", *International Journal of Man-Machine Studies*, pp.543-554, 1983.

[Big93] T. J. Biggerstaff, B. Mitbander, and D. Webster, "The concept assignmet problem in program understanding", *In 15th International Conference on Software Engineering*, Baltimore, Maryland, May1993. IEEE Computer Society Press, Los Alamitos, California, USA.

[Boh96] S. Bohner and R. Arnold, "An Introduction to Software Change Impact Analysis", *Software Change Impact Analysis, IEEE Computer Society*, 1996.

[Buh96] R.J.A. Buhr and R.S. Casselman, "Use Case Maps for Object-Oriented Systems", Prentice Hall, 1996.

[But97] G. Butler, P. Grogono and F. Khendek, "A Z Specification of Use Cases", In *Proc. of the Asia-Pacific Software Engineering Conference and International Computer Science Conference,*. IEEE Computer Society Press, pp. 505-506, 1997.

[Buh98] R.J.A. Buhr, "Use Case Maps as Architectural Entities for Complex Systems", *IEEE Transactions on Software Engineering*, Archive Volume 24, Issue 12, pp. 1131-1155, December 1998.

[Chi90] E. J Chikofsky & J. H. Cross. "Reverse engineering and design recovery: A taxonomy." *IEEE Software, v.7 n.1*, pp. 13-17, January 1990.

[Cho94] J. D. Choi and J. Ferrante, "Static slicing in the presence of goto statements", *ACM Transactions on Programming Languages and Systems*, vol. 16, no. 4, pp. 1097-1113, 1994.

[Coh98] S. Cohen, and L. Narthrop, "Object-Oriented Technology and Domain alalysis", Proceedings of IEEE ICSR5, Victoria, June 1998.

[Che00] K. Chen and V. Rajlich, "Case Study of Feature Location Using Dependence Graph", In *Proc. of the 8th Int. Workshop on Program Comprehension*, pp. 241-249, June 2000.

[Cze00] J. Czeranski, T. Eisenbarth, H. Kienle, R. Koschke and D. Simon, "Analyzing xfig Using the Bauhaus Tool", *Working Conference on Reverse Engineering*, Brisbane, Australia, IEEE Computer Society Press, pp. 197-199, November 2000.

[Che01] K. Chen and V. Rajlich, "RIPPLES: Tool for Change in Legacy Software", *IEEE International Conference on Software Maintenance*, pp.230-239, 2001.

[Cha04] Phillipe Charland, "Enhancing Traditional Behavioural Testing through Program Slicing", *Master Thesis*, Department of computer Science, Concordia University, Montreal, Canada, September 2004.

[Dav82] A. M. Davis, "The design of a family of application oriented requirements languages" *Computer 15(5),* pp. 21-28, 1982.

[Dav93] A.M. Davis, "Software Requirements", Prentice Hall, Englewood Cliffs, New Jersey, 1993.

[Doi98] A. Doinisi, N. Argentieri, "FODAcom: An Experience with Domain Analysis in the Italian Telecom Industry", *In: Proceedings of Fifth International Conference on Software Reuse*, pp.166-175, 1998

[Eis01a] T. Eisenbarth, R. Koschke, D. Simon, "Aiding Program Comprehension by Static and Dynamic Feature Analysis", *IEEE International Conference on Software Maintenance (ICSM'01), Florence, Italy*, pp.602, November 2001.

[Eis01b] T. Eisenbarth, R. Koschke, and D. Simon, " Derivation of Feature Components Maps by Means of Cocept Analysis, *CSMR*, PP. 176-179, 2001.

[Gal89] K.B. Gallegher and J.R. Lyle, "Using Program slicing in Software Maintenance", *IEEE Transactions on Software Engineering, Vol. 17, Issue 8,* pp. 751 – 761, August 1991.

[Gop91] R. Gopal, "Dynamic program slicing based on dependence relations", In *Proceedings of the Conference on Software Maintenance,* pp. 191-200, 1991.

[Gup92] R. Gupta, M. Harrold and M. Soffa, "An approach to regression testing using slicing", *In Proceedings of the Conference on Software Maintenance*, pp. 299-306, 1992.

[Gri98] M. Griss, J Favaro, M. Alessandro, "Integrating Feature Modeling with the RSEB", *In Proceedings of the Fifth International Conference on Software Reuse*, Canada, 1998.

[Hor90] S. Horwitz, T. Reps and D. Binkley, "Interprocedoral slicing using dependence graphs", *ACM transaction on Programming Languages and Systems*, vol. 12, pp. 26-60, 1990.

[Hor92] S. Horwitz and T. Reps, "The use of Program Dependence Graphs in Software Engineering", *Proceedings of the 14th International Conference on Software Engineering*, May 1992.

[Har95] S. Harman, S. Danicic and Y. Sjvaguranathan, "A parallel algorithm for static program slicing", *Information Processing Letters*, 56960, pp. 307-313, December1995.

[Har01] M. Harman and R. M. Hierons, "An overview of program slicing", *Software Focus 2*, 3 (2001), pp. 85-92, 2001.

[IEE 90] Institute of Electrical and Electronics Engineers, *IEEE Standard Computer Dictionary: A Compilation of IEEE Standard Computer Glossaries,* New York, 1990.

[IEE94] ANSI/IEEE Std 830- 1994,"IEEE Guide to Software Requirements Specifications", *IEEE Computer Society Press*, 1994.

[Idr00] H. Idris and P. Colin, "Studying the Evolution and Enhancement of Software Features", *International Conference on Software Maintenance (ICSM'00)*, San Jose, California, October 2000.

[Jac97] I. Jacobson, M. Griss and P. Jonsson, "Software Reuse: Architecture, Process and Organisation for Business Success", *Addison-Wesley and ACM Press, Reading MA,*pp. 66,1997.

[Jun01] "JUnit", *JUnit.org*, April2001. http://www.JUnit.org/index.htm

[Kor88] B. Korel and J. Laski, "Dynamic program slicing", *Information Processing Letters*, 29(3):155-163, Oct. 1988

[Kan90] C. Kang, S.G. Cohen, J.A. Hess, W.E. Novak and A.S. Peterson, "Feature-Oriented Domain analysis (FODA) Feasibility Study", *Technical Report CMU/SEI-90-TR-21*, ESD-90-TR-222, November 1990.

[Kru93] J.R.W. Krut, "Integrating 001 Tool Support into the Feature-Oriented Domain Analysis Methodology", *Technical Report SEI-93-TR-11*, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA., July 1993.

[Kor94] B. Korel and S. Yalamanchili, "Forward Computation of Dynamic Program Slices", *Proceedings of the 1994 ACM SIGSOFT international symposium on Software testing and analysis*, pp. 66-79, 1994.

[Kor94] B. Korel and S. Yalamanchili, "Forward Derivation of Dynamic Slices", *Proceedings of the International Symposium on Software Testing and Analysis*, pp. 66-79, 1994.

[Kor97] Korel, B., " Computation of Dynamic Program Slices for Unstructured Programs," *IEEE Transactions on Software Engineering, vol. 23*, no. 1, pp. 17-34, January 1997.

[Kir97] T. G. Kirner, J. C. Abib, "*Inspection of software requirements specification documents: a pilot study*" *Proceedings of the 15th annual international conference on Computer documentation, ACM Special Interest Group for Design of Communications,* October 1997.

[Kan98] K.C. Kang, S.J. Kim, J.J. Lee, K.J. Kim and E. Shin, "FORM: A Feature-Oriented Reuse Method with Domain-Specific Reference Architectures", *To appear in Annuals of Software Engineering*, Vol. 5, 1998.

[Kul00] D. Kulak and E. Guiney with illustrations by E. Lavkulich, "Use Cases Requirements in Context", *ACM Press*, Addison-Wesley, May 2000.

[Kos04] R. Koschke, "Bauhaus Library", February 2004.

http://www.iste.uni-stuttgart.de/ps/bauhaus/papers/

[Leh80] M. Lehman. "Programs, life cycles and laws of software evolution", *Proceedings of IEEE Special Issue on Software Engineering*, 68(9), pp. 1060-1076, September 1980.

[Lyl93] J.R. Lyle and M. Weiser, " Automatic program bug location by program slicing", *Proceedings of 2nd International Conference on Computers and Applications*, Peking, China, pp.877-882, 1987.

[Liv94] P.E. Livadas and A. Rosenstein, "Slicing in the presence of pointer variables", *Technical Report SERC-TR-74-F*, Computer Science and Information Services Department, University of Florida, Gainesville, FL, June 1994.

[Li01] X. Li, Z. Liu and J. He, "Formal and Use-Case Driven Requirement Analysis in UML", *UNU/IIST Report*, No. 230, March 2001.

[McC92] C.L. McClure, "The Three Rs of Software Automation: re-engineering, repository, reusability", *N.J. Englewood Cliffs*, Prentice Hall, 1992.

[Mul94] H.A. Müller, K. Wong and S.R. Tilley, "Understanding Software Systems Using Reverse Engineering Technology", In *The 62nd Congress of L'Association Canadienne Francaise pour l'Avancement des Sciences Proceedings (ACFAS)*, 1994.

[May95] A.V. Mayrhauser and A.M. Vans, "Program comprehension during software maintenance and evolution", *IEEE Journal*, Computer, Vol. 28, Issue: 8, pp. 44-55, August 1995.

[Mal99] R. Malan and D. Bredemeyer, "Functional Requirements and Use Cases", The Architecture Discipline, June 1999. http:// www.bredemeyer.com/use_cases.htm

[Mul00] H.A. Müller, J.H. Jahnke, D.B. Smith, M.A. Storey, S.R. Tilley and K. Wong, "Reverse Engineering: A Roadmap", *In Proceedings of Future*

of Software Engineering, Limerick, Ireland, June 2000.

[Meh01a] A. Mehta, "Evolving Legacy Systems Using Feature Engineering and CBSE", *Proceedings of the 23rd international conference on Software engineering*, ICSE, pp. 797-798, 2001.

[Meh01b] A. Mehta and G.T. Heineman, "Evolving Legacy System Features using Regression Test Cases and Components",

Proceedings of the 4th international workshop on Principles of software evolution, ICSE, pp. 190-193, 2001.

[Meh01c] A. Mehta and G.T. Heineman, "Evolving Leagcy Systems by Locating System Features Using Regression Test Cases", *CiteSeer.IST Scientific Literature Digital Library,* June 2001. http://citeseer.ist.psu.edu/441595.html

[Mur01] G.C. Murphy, A. Lai, J. Rober, Walker, and Martin P. Robillard, "Separating Features in Source Code: An Explanatory Study", *Proceedings of the 23rd International Conference on '01*, ICSE, May 2001.

[Meh02] A. Mehta and T. Heineman, "Evolving Legacy System Features into Fine-Grained Components", *Proceedings of the 24th International Conference on Software Engineering*, Orlando, Florida, May 2002.

[Nel96] M.L. Nelson, "A Survey of Reverse Engineering and Program Comprehension", *CiteSeer.IST Scientific Literature Digital Library*, 1996. http:// www.citeseer.ist.psu.edu/rugaber95program.html

[Ott84] K.J. Ottenstein and L.M. Ottenstein, "The Program Dependence Graph in a Software Development Environment", *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pp. 177-184, 1984.

[Ric90] C. Rich ,L. Wills. "*The Programmer's Apprentice". ACM Press*, 1990.

[Rug92] S. Rugaber," Program Comprehension for Reverse Engineering" *AAAI Workshop on AI and Automated Program Comprehension* , San Jose, California,1992.

[Rug94] S. Rugaber, "*White Paper on Reverse Engineering",* Georgia Institute of Technology, 1994.

[Rug95] S. Rugaber, K. Stirewalt, and L. Wills, " *The Interleaving Problem in Program Understanding*", In: Working Conference on Reverse Engineering, pp. 166-175, 1995.

[Raj97] V. Rajlich, "A model for Change Propogation based on Graph Rewriting", *International Conference on Software Maintenance (ICSM'97)*, Bari, Italy, Oct. 1997.

[Ril98] J. Rilling, " Investigation of Program Slicing and its Applications in Program Comprehension of Large Software Systems,", *Ph.D. Thesis*, Illinois Institute of Technology, Chicago, Illinois, 1998.

[Ril01] J. Rilling, "Concept", 2001. http://www.cs.concordia.ca/CONCEPT/

[Raj02] V. Rajlich, N. Wilde, "The Role of Concepts in Program Comprehension", *Proceedings of the 10th International Workshop on Program Comprehension,* IEEE Computer Society , Washington, DC, USA, 2002.

[Ril04] J. Rilling,W. Meng, and O. Ormandjeva, "Context Driven Slicing Based Coupling Measures", *20 th IEEE International conference on Software Maintenance*, IEEE Computer Society, Chicago, Illinois, 11-14 September,2004.

[Ste74] W. Stevens, G.J. Myers, and L.L. Constantine, "Structured design", IBM Systems Journal 13(2), pp.115-- 139, 1974.

[Sne94] G. Snelting, "Reengineering Class Hierarchies Using Concept Analysis', Proc. of the ACM SIGSOFT Symposium on the Foundations of Software Engineering, pp. 99-110, November,1994.

[Sne97] G. Snelting, F. Tip "Reengineering of configurations Based on Mathematical Concept Analysis", ACM Transactions on Software Engineering and Methodology 5,2, pp. 146-189, April 1997.

[Sti97] M. Stiff, "Identifying Modules Via Concept Analysis", *Proc. of. The International Conference on Software Maintenance*, 1997.

[Son99] Y. Song and D. Huynh, "Forward Dynamic Object-Oriented Program Slicing", *Application-Specific Systems and Software Engineering and Technology (ASSET '99),* IEEE, pp. 230-237, 1999.

[Suc00] G. Succi, A. Valerio, T. Vernzza, M. Fenaroli and P. Predenzani, "Framework extraction with domain analysis", *ACM Computing Surveys*, Vol.32, No. 1, March 2000.

[Ton96] P. Tonella, R. Fiutem, G. Antoniol, and E. Merlo, "Augment-ing Pattem-Based Architectural Recovery with Flow Anal-ysis: Mosaic - A Case Study", *In: Working Conference on Reverse Engineering*, pp. 198-207, 1996.

[Tan99] H.B.K. Tan and J.T. Kow, "Extracting Code Fragment that Implements Functionality", *Sixth Asia Pacific Software Engineering Conference*, IEEE, pp. 351-354, December 1999.

[Tib99] G. Tibor, B. Árpád and F. István, "An Efficient Relvevant Slicing Method for Debugging", *Software Engineering Notes*, Software Engineering –ESEC/FSE'99 Spinger ACM SIGFT, pp. 303-321, 1999.

[Tur99a] C.R. Turner, "Feature Engineering of Software Systems", *Ph. D. Thesis*, Department of computer Science, University of Colorado, Boulder, December 1999.

[Tur99b] C.R. Turner, A. Fuggetta, L. Lavazza and A. L. Wolf, "A conceptual basis for feature engineering", *Journal of Systems and Software*, 49(1), pp. 3-15, December 1999.

[Wei79] M. Weiser, "Program slices: formal, psychological and practical investigations of an automatic program abstraction method", *Ph.D. thesis*, University of Michigan, Ann Arbor, 1979.

[Wei84] M. Weiser, "Program slicing", *IEEE Transactions on software engineering*, Vol. SE-10, No. 7, pp. 352-357, July 1984.

[Wil94] N. Wilde, "Faster Reuse and Maintenance Using Software Reconnaissance", *Software Engineering Research Center (SERC)*, CSE-301, University of Florida, Gainesville, FL 32611, July 1994.

[Wil95] N. Wilde and M.C. Scully, "Software Reconnaissance: Mapping Program features to code", *Journal of Software Maintenance: Research and Practice*, Vol. 7, No. 1, pp. 49-62, January-February 1995.

[Wan96] Y. Wang, W.T. Tsai, X. Chen and S. Rayadurgam, "The Role of Program Slicing in Ripple Effect Analysis", *The 8th International Conference on Software Engineering and Knowledge Engineering (SEKE)*, Knowledge Systems Institute, Lake Tahoe, Nevada, USA, June 1996.

[Won99] W.E. Wong, S.S. Gokhale, K.S. Trivedi and J.R. Horgan, "Locating Program Features Using Execution Slices," In *Proc. of Application Specific Software Engineering and Technology (ASSET 99)*, pp. 194-203, Dallas, TX, March 1999.

[Whi01] L.J. White and N. Wilde, " Dynamic Analysis for Locating Product Features in Ada Code", *Proceedings of the 2001 annual ACM SIGAda international conference on Ada,* ACM SIGAda ada letters vol. 21, Issue 4, September 2001.

[Wil02] N. Wilde, "Recon Tool for C programmers", *Recon2*, March 2002. http:// www.cs.uwf.edu/~recon/

[You79] E. Yourdon and L.L. Constantine, "Structured Design", Prentice-Hall, Englewood, New Jersey, 1979.

```
1.  public class Account {
2.
3.   double balance;
4.
5.  public Account(double accountbalance) {
6.   balance=accountbalance;
7.   }
8.
9.   void deposit(double amount){
10.  balance+=amount;
11. }
12.
13.  double getbalance(){
14.  return balance;
15.  }
16.
17. public String toString (){
18. return "Current balance of "+balance;
19. }
20.
21. public static void main(String[] args)throws IOException {
22.
23. BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
24. Account A = new Account(100);
25. System.out.println(" A's current balance is "+ A.getbalance());
26. String strnum = in.readLine();
27. double amount = Double.parseDouble(strnum);
28. A.deposit(amount);
29.  }
30. }
```

Appendix 1 The Account.java program

Input: a slicing criterion $C=(x,y^q)$
Output: a dynamic slice for C

Tx: execution trace upto position q
Φ_C: a set of block traces
R_c: a set of blocks
I_C: a set of contributing actions (actions set as visited)
In: a set of noncontributing actions(actions set as visited)
Nb: *a set of noncontributing blocks*

1 Initiaize Ic as empty, In to empty and *N*b to all blocks of P
2 Find and mark as contributing the last definition of y^q
3 **repeat**
4 Find contributing actions
5 Find non-contributing actions
6 Mark all neutral actions as contributing
7 **until** all actions are marked as contributing or non-contributing in T_x up to position q
8 Show a dynamic slice that is constructed from P by removing all blocks that belong to Nb.

procedure Find contributing actions
9 **while** there exists a contributing and not visited action in T_x **do**
10 Select a contributing and not visited action X^k in T_x
11 Mark X^k as a visited action ($I_C := I_C \cup \{X^k\}$)
12 **for** all variables $v \in U(X^k)$ **do**
13 Find and mark as a contributing action the last definition of v
endfor
14 **for** all blocks $B \in R_c$ **do**
15 **if** $X \in N(B)$ **then** $N_B := N_B - \{B\}$
endfor
16 **endwhile**
end Find contributing actions

Appendix 2 An example of the computation of dynamic backward algorithm [Kor95]

```
procedure Find non-contributing actions

17 Mark as neutral all actions that are not marked as contributing
18 p:=1
19 repeat
20     Let X^p be an action at position p in T_x
21     if X^p is not a contributing action (X^p ∉ I_C) then
22      Let B be a block which has an r-entry at position p
23      if B ∈ Nb then
24        if there exists an r-exit from block B at position p1 such that
25          (1) p < p1 < q
26          (2) all actions between p and p1 are not marked as contributing
27        then
28          Mark all actions between p and p1 as non-contributing (Φ_C := Φ_C
        ∪{S(B,p,p1)})
29          p := p1
30        endif
31     endif
32     p := p + 1
33 until p ≥ q

end non- contributing actions
```

Appendix 2 [Continued]

```
........
........   // Represent other statements
.........
41.        protected void printHeader(long runTime) {
42.        [illegible]
43.        [illegible]
   elapsedTime[illegible] // F3

44.        }
45.
.......    // Other statements of the program
........
81.        protected void printFooter(TestResult result) {
82.        if (result.wasSuccessful()) {
83.        [illegible]
84.        [illegible]
85.        [illegible] test[illegible]
   + (resu[illegible]
      86.
87.        } else {
88.        getWriter().println();
89.        getWriter().println("FAILURES!!!");
90.        getWriter().println("Tests run: " + result.runCount() +
   ", Failures: " + result.failureCount() + ", Errors: " +
   result.errorCount());
91.        }
92.        getWriter().print[illegible] //F[illegible]
93.        }

......
......
130.       public void startTest(Test test) {
131.       getWriter[illegible] //F[illegible]
132.       if (fColumn++ >= 40) {
133.       getWriter().println();
134.       fColumn= 0;
135.       }
.........
.........
```

Appendix 3 Identified output statements and output feature criteria from the ResultPrinter class for test case 1