Supporting UCM Requirements Evolution by Means of Formal Concept Analysis

Maryam Shiri

A Thesis
in
The Department
of
Computer Science and Software Engineering

Presented in Partial Fulfillment of the Requirements
for the Degree of Master of Applied Science (Software Engineering) at
Concordia University
Montreal, Quebec, Canada

February 2008

Your file Votre référence
ISBN: 978-0-494-40934-3
Our file Notre référence
ISBN: 978-0-494-40934-3

Abstract

Supporting UCM Requirements Evolution by Means of Formal Concept Analysis

Maryam Shiri

Software evolution refers to the process of adapting software systems to modification request caused by requirements and technology changes that occur after the initial system delivery. Prior to implementing a requirements change, decision-makers must determine if a modification request is feasible and the cost associated with it. Traditional modification analysis approaches have focused mainly on the analysis of source code to determine the potential impact of a change.

In this research, we present a novel approach to support modification analysis early in the software evolution lifecycle thus allowing management, who are not necessarily familiar with the detailed system implementation, to determine the impact associated with a modification request. In our research we develop several analysis techniques to support the system evolution at the requirements level, namely impact analysis at the requirements level, prediction of regression testing effort, feature interaction analysis including detecting potential bad smells of the requirements, in order to provide support for improving the comprehensibility and maintainability of requirements. In our approach, we combine Use Case Maps, as a notation to model requirements, with Formal Concept Analysis in order to explore the applicability of Formal Concept Analysis during modification analysis at the requirements level. We implemented a proof of concept tool and several case studies are presented to demonstrate the applicability of our methodology.

Acknowledgements

I am most grateful to my supervisor, Dr. Rilling, for his encouragement, support, and patience throughout this research. Without his help, advice and positive attitude, my studies in this area would not have been possible.

I would also like to thank, my family for their great love. Their support was a major factor in the success of my work.

I would also like to extend my appreciation to Jameleddine Hassine for his helpful comments and suggestions.

Dedication

To my parents, for their unconditional love and who are the inspiration of my life.

Table of Contents

List of Figures

List of Tables

1. Introduction

Requirement specifications, regardless of how they are articulated, represent a way to transform vague ideas and goals into concrete and comprehendible documents for the stakeholders and engineers during the process of requirement engineering, and later in the maintenance process. Yet it is impossible to always construct requirement specifications right the first time without the necessity for change. These imprecise requirements often play a major and costly role in project failures [1]. A requirements change is typically referred to as either the modification or deletion of an existing requirement, or the addition of a new requirement. Even though requirement changes are not the sole reason for software evolution, they are responsible for 80% of all necessary software maintenance activities [65]. In [21], Brooks states, "the hardest single part of building a software system is deciding precisely what to build. No other part of the conceptual work is as difficult as establishing the detailed technical requirements. No other part of the work so cripples the resulting system if done wrong. No other part is as difficult to rectify later." This emphasizes the importance of analyzing requirements, their overlaps and conflicts, and managing their modifications [110].

While developing software systems, it is rare that an initial system design will correspond completely with the final design or implementation of a system [70]. Ever changing customer needs, such as changing or adding new features, lead to requirement modifications (perfective maintenance) [17], while repairing faults (corrective maintenance) and adapting to new environment changes [87] (adaptive maintenance) are other possible reasons for evolution in requirements [57].

The efficient management and execution of these changes are critical to software quality and the evolution of software systems [17]. Maintenance processes (e.g., [57]) have been established to guide both management and maintainers during typical requirement maintenance and evolution tasks.

Common to these process models is that they require some type of modification analysis (MA) prior to committing to a change. The initial MA is typically performed at a higher non source code-based abstraction level (e.g. requirements or design level). During a MA, management, who may not necessarily be familiar with the detailed implementation of a system, have to determine the feasibility, impact, and cost associated with such a change request.

1.1 Contributions

In this research, we propose a novel framework for supporting an approach which we refer to as UCM Requirements Evolution Analysis Framework (UREAF). The goal of this framework is to identify potential change impact and re-testing efforts at the requirements level, without the need for implementation-specific details. Our UREAF approach also supports the identification of possible feature interactions within existing system requirements and allows for the detection of potential bad smells in the requirements, to improve future evolvability of these requirements. We illustrate our new approach using Use Case Maps (UCM) [117], an existing requirement modeling technique that is being translated into a formal semantics representation [50] to make the requirements model executable. Traces are created from this model by executing UCM

[117] scenarios that are then collected and further analyzed using Formal Concept Analysis (FCA) [41].

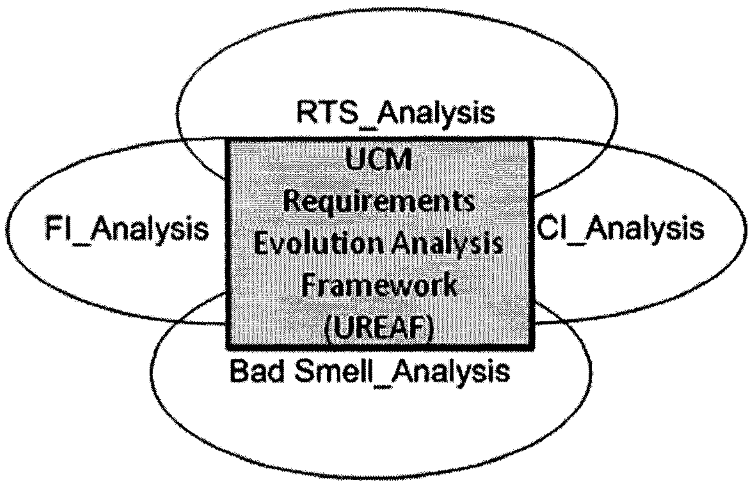


Figure 1-1: UCM Evolution Framework

Within the software engineering community, FCA has various applications in program comprehension and maintenance applications. However, to the best of our knowledge there exists no previous work utilizing FCA for evolution analysis at the requirements level.

We introduce four typical evolution analysis tasks (Figure 1-1) supported by our implemented UREAF framework, which we will be revisited throughout the thesis to illustrate the applicability of the presented approach. The four analysis task examples are:

1. *Determine the impact of a modification request on traces*
2. *Estimate the test cases that have to be re-tested as part of a modification request*
3. *Identify Use case scenarios which contain feature interactions*
4. *Suggest some of the potential bad smells within the UCM system*

The major contributions of this thesis can be summarized as follows:

(1) explored the use of Formal Concept Analysis in combination with formalized UCM traces to support evolution analysis;

(2) introduced different analysis techniques, including change impact, regression testing, feature interaction and bad smells detection at the requirement specification level;

(3) implemented a proof of concept tool, UREAF_tool, which implements the introduced methodologies and automates the analysis process. We have shown that this tool can be used to extract dependency and perform different types of evolution analysis at the UCM level.

1.2 Publications

Research results presented in this thesis have been published in proceedings of three recognized international conferences and workshops including:

- Proceeding of the Ninth international Workshop on Principles of Software Evolution in Conjunction with the 6th ESEC/FSE Joint Meeting (IWPSE 2007) where we presented an FCA-based approach for determining change impact analysis and regressing testing at the UCM level [106].
- In Proceedings of the Twenty-Second IEEE/ACM international Conference on Automated Software Engineering (ASE'07) where we proposed the possibility of detecting feature interaction analysis by means of FCA [107] in terms of a short paper.

- In proceeding of the Third International IEEE Workshop on Software Evolvability where we introduced an evolution analysis framework at the requirement specification level using FCA [105].

Parts of this research are a joint effort with Jameleddine Hassine, a PhD student at Concordia University. His contribution mainly focused on providing the formal semantics and verification of Use Case Maps by means of Abstract State Machines Languages (AsmL) in order to generate dynamic UCM traces which are used as inputs to the research presented in this thesis.

The remainder of the thesis is organized as follows: Section 2 introduces Use Case Maps, Formal Concept Analysis, and some background relevant to change impact analysis, regression testing, and feature interaction. Section 3 introduces our modification change impact analysis and selective regression testing approach based on UCM and FCA. An initial case study is demonstrated in Section 4, followed by related work and discussions in Section 5. Section 6 presents conclusions and future work.

2. Background

In an attempt to make this thesis self-contained, we review some of the relevant background including FCA, Use Case Maps, feature interaction, and impact analysis.

2.1 UCM

Use Case Maps (UCM) [117] are a modeling technique, originally introduced by Buhr [23], applied to capture functional requirements in terms of causal scenarios. UCMs can represent behavioral aspects at a higher level of abstraction than, for example, UML diagrams. UCM can visualize an entire system to provide a better understanding of the evolving behavior of complex and dynamic systems [117] at the requirements and specification level. The Use Case Maps notation [58] is a high-level scenario-based modeling technique used to specify functional requirements and high-level designs for various reactive and distributed systems. Furthermore, UCMs can provide stakeholders with guidance and reasoning about system-wide functionalities and behavior. UCMs were originally introduced to model the behavior of telecommunication systems. In recent years, UCMs have also been applied in other application domains, such as web services, airline reservations, object-oriented frameworks and many more [4, 120, 121].

A UCM model (Figure 2-1) depicts scenarios as causal flows of responsibilities (e.g. operation, action, task, function, etc.) that can be superimposed on the underlying component structures. Components are generic and can represent software entities (objects, processes, databases, servers, etc.) as well as non-software entities (e.g. actors or hardware). These relationships are said to be causal because they involve concurrency, partial ordering of activities, and they link causes (e.g., preconditions and triggering events) to effects (e.g. post-conditions and resulting events).

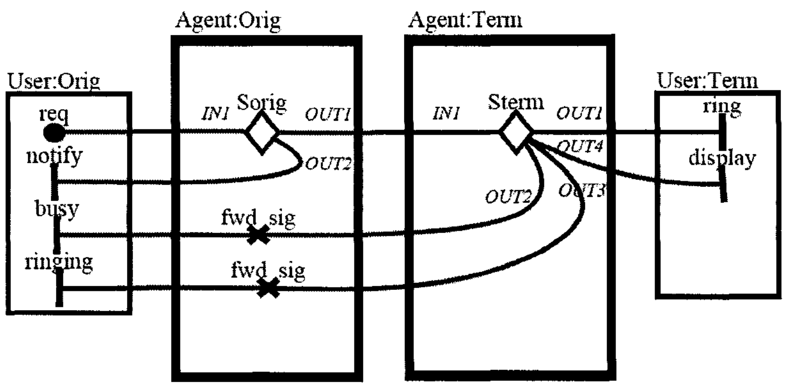


Figure 2-1: A Simple Telephony System (UCM root map) [81]

The UCM notation expresses scenarios above the level of messages exchanged between components, hence, they are not necessarily bound to a specific underlying structure (such UCMs are called Unbound UCMs). Path details can be hidden in sub-diagrams, called plug-ins, contained in stubs (containers) on a path. A stub can be either static (represented as plain diamond) containing only one plug-in, or dynamic (represented as dashed diamonds), which may contain several plug-ins. Dynamic plug-ins are selected at run-time according to a selection policy. Strengths of UCM is its ability to integrate a number of scenarios together (in a map-like diagram), as well as the ability to interpret the architecture and its behavior over a set of scenarios. UCM is not intended to replace UML, but rather complement it and bridge the gap between requirements (use cases) and design (system components and behavior). In comparison with UML, UCMs provide a complementary fit between use cases and behavioral diagrams. Use cases often describe the system according to its external behavior (black-box view), whereas UML class diagrams are used to describe how the system is constructed (glass-box), but do not describe how it works. UCM is an attempt to close the conceptual gap that typically exists between requirements and design [10]. Use Case Maps are part of a new proposal to ITU-T for a User Requirements Notation (URN) [56]. UCMs have been useful in a number of areas such as design and validation of telecommunication and distributed systems [6, 7], detection and avoidance of undesirable feature interactions [25, 26], evaluation of architectural alternatives [81], and performance evaluation [89].

2.1.1 Basic UCM Notation

As shown in the simple telephony example, Figure 2-1, and continued in Figure 2-4, the basic UCM contains the following constructs:

Start points (represented as a filled circle): The execution of a scenario path begins at a start point (e.g *req* in Figure 2-1),

Responsibilities (shown as crosses): Responsibilities are abstract activities that can be described in terms of functions, tasks, procedures, events (e.g. *fwd_sig* [forwarding signal in Figure.2-1]).

End points (represented as vertical bars): The execution of a path terminates at an end point (e.g. busy in Figure 2-1).

Components (represented as simple boxes): UCM components are abstract enough to correspond to software entities (e.g. object, process, etc), as well as non-software entities (e.g. actors, hardware, *user: Orig* in Figure 2-1).

UCMs also support structuring and integrating scenario sequences by using alternatives (with OR-forks/joins shown in Figure 2-2(a)) or concurrent paths (with AND-forks/joins Figure 2-2 (b)).

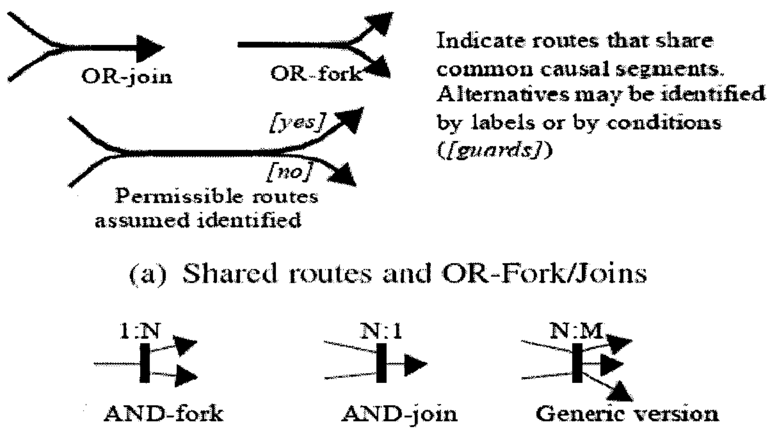

OR-join
OR-fork
Indicate routes that share common causal segments. Alternatives may be identified by labels or by conditions ([guards])
[yes]
[no]
Permissible routes assumed identified
(a) Shared routes and OR-Fork/Joins
1:N
N:1
N:M
AND-fork
AND-join
Generic version
(b) Concurrent routes with AND-Fork/Joins

Figure 2-2: Structuring Scenarios [52]

OR-Joins: Capture the merging of two or more independent scenario paths.

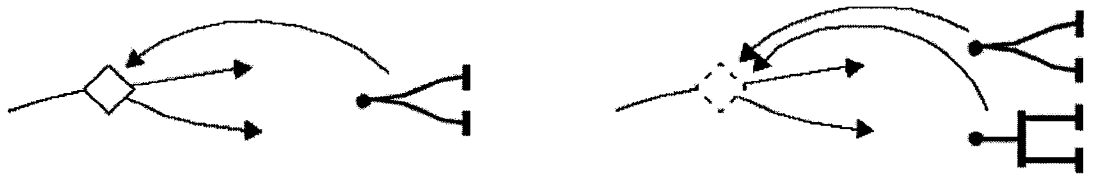
OR-Fork: Split a scenario path into two or more alternative paths, generally determined by Boolean conditions (guards).

AND-Forks: Split a single control into two or more concurrent control.

AND-Joins: Capture the synchronization of two or more concurrent scenario paths.

Scenario variables (global variables) are introduced to model system states, and to control alternative scenario paths, and selection policies [81].

When maps become too complex to be represented as one single UCM diagram, a mechanism for structuring sub-maps becomes necessary. UCM provides the notion of stub concepts, allowing for hierarchical decomposition of complex maps. UCM path details can be hidden in separate sub-diagrams, referred to as plug-ins, contained in stubs (diamonds) on a given path. These plug-ins are reusable UCMs that can be used (plugged in) in many stubs (Figure 2-3).



(a) Static stubs have only one plug-in (b) Dynamic stubs may have multiple plug-ins

Figure 2-3: Stubs and Plug-ins [52]

The two types of Stubs are:

Static stubs (represented as plain diamonds): They contain only one plug-in.

Dynamic stubs (shown as dashed diamonds): They may contain several plug-ins, whose selection is determined at run-time according to a selection policy (often described with preconditions).

2.1.2 UCM Example – A Simple Telephony System

Figure 2-1 shows the UCM root map of a telephone example that was originally introduced in [81] and referenced in [52]. The example illustrates the connection request phase in an agent-based telephony system with user-subscribed features. The case study contains four components (originating/terminating users and their agents) and two static stubs. Upon the request of an originating user (*req*), the originating agent selects the appropriate user feature (in stub *Sorig*) which could result in some feedback (*notify*). This may also cause the terminating agent to select another feature (in stub *Sterm*), which in turn, can cause different results in the originating and terminating users. Stub *Sorig* contains the *Originating* plug-in, whereas stub *Sterm* contains the *Terminating* plug-in. These sub-UCMs have their own stubs, whose plug-ins correspond to user-subscribed features.

In stub *Sscreen* we have the following plug-ins:

- **OCS (Originating Call Screening)**: This plug-in, blocks calls to people on the OCS filtering list. It checks whether the call should be denied or allowed (chk). When denied, an appropriate event occurs at the originator side (notify).
- **Default**: This is used when the user is not subscribed to any other originating feature.

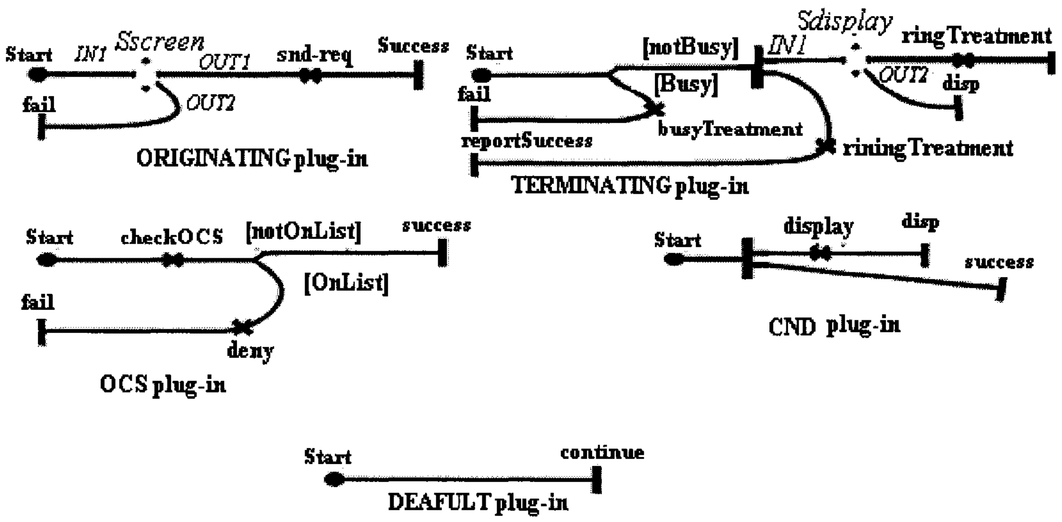


Figure 2-4: Plug-ins for Simple Telephony Features [81].

The plug-ins in *Sdisplay* are:

- **CND (Call Number Delivery)**: This plug-in displays the caller's number on the callee's device (display) concurrently with the rest of the scenario (update and ringing).
- **Default**: This plug-in is used when the user is not subscribed to any other terminating feature.

As part of this UCM a set of global variables are defined: *Busy* (the callee is busy), *OnOCSList* (the callee on OCS list), *subCND* (the callee is subscribed to CND), *subOCS* (the caller is subscribed to OCS). Figure 2-4 illustrates the corresponding UCM that was generated with UCM Navigator. Each plug-in is bound to its parent stub, that is, stub input/output segments (IN1, OUT1, etc.) are connected to the plug-ins' start/end points. A more detailed discussion and description of the case study can be found in [53].

2.1.3 UCM Tools

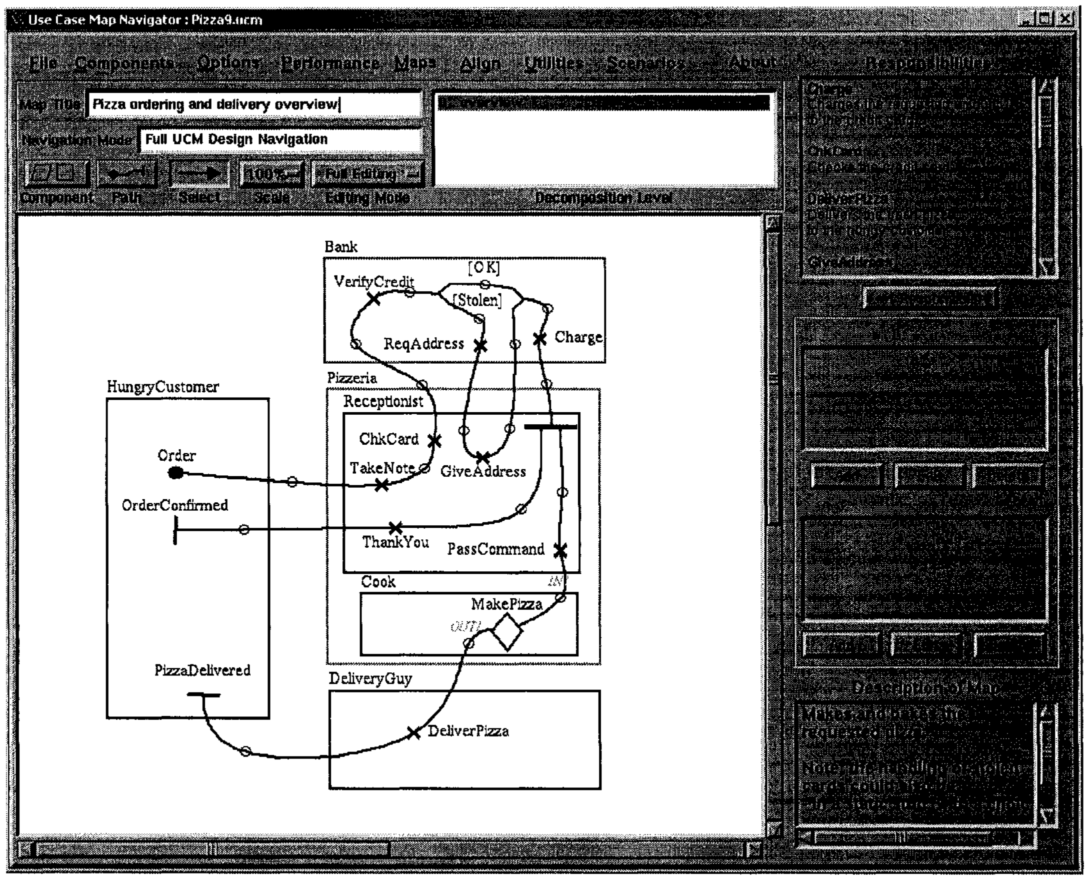
In an effort to support the use of UCMs, there are two freely available editing tools: UCMNav (UCM Navigator) [84] and jUCMNav [59]. In this thesis, creation and maintenance of UCM models were supported by both tools.

UCM Navigator

UCMNav [118], developed by Andrew Miga [84] at Carleton University, provides features for creating syntactically correct UCMs manipulated with respect to a defined UCM Document Type Definition (DTD) that is available at [118]. UCMNav uses this DTD to ensure that syntactic and static semantic rules are satisfied [81].

In addition, UCMNav maintains various kinds of bindings (plug-ins to stubs, responsibilities to components, subcomponents to components, etc.) and allows users to edit the plug-ins of each stub at all levels and allows for the generation of comments and descriptions for the UCM specification. UCMNav also supports the generation of XML descriptions and scenario definitions, enabling the highlights of scenario paths in a UCM specification, and providing the ability to export UCMs in different formats (e.g. Encapsulated Postcript(EPS), Computer Graphics Metafile (CGM), Maker Interchange Format(MIF), etc) [4]. The tool also supports the converting of created (or imported) UCMs to other models, such as the performance model, Layered Queuing Networks

(LQN) and message sequence charts (more details on these transformations can be found in [89] and [81] respectively). UCMNav is supported by multiple platforms such as: Solaris, Linux (Intel and Sparc),HP/UX, and Windows (95, 98, 2000, XP and NT).


Use Case Map Navigator : Pizza9.ucm

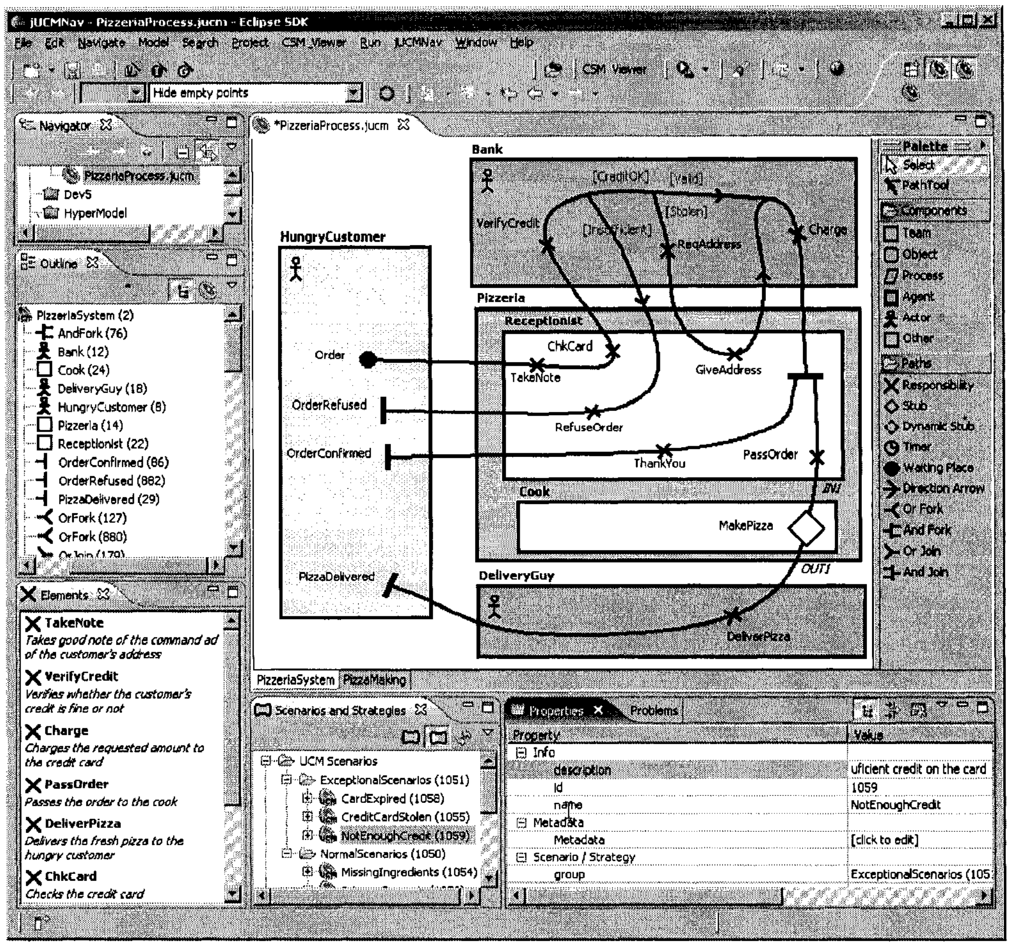
Map Title: Pizza ordering and delivery overview

Navigation Mode: Full UCM Design Navigation

Figure 2-5: UCMNav GUI

jUCMNav

jUCMNav [59] is an open-source tool distributed as an Eclipse plug-in, supporting the editing and analyzing of URN models. jUCMNav [100] originally supported only the development of Use Case Maps scenario models. In its most recent version, it was extended to support GRL [57] and provide complete URN coverage. This integration of UCM and GRL views in the same tool allows for the creation of various types of traceability links among these notations. In the meantime, jUCMNav was further extended to support scenario definitions.



Elements

TakeNote
Takes good note of the command ad of the customer's address

VerifyCredit
Verifies whether the customer's credit is fine or not

Charge
Charges the requested amount to the credit card

PassOrder
Passes the order to the cook

DeliverPizza
Delivers the fresh pizza to the hungry customer

ChkCard
Checks the credit card

Property	Value
Info	
description	uficient credit on the card
Id	1059
name	NotEnoughCredit
Metadata	
Metadata	[click to edit]
Scenario / Strategy	
group	ExceptionalScenarios (105

Figure 2-6: jUCMNav GUI

2.1.4 UCM Formalization

In [51], an operational semantics for the UCM language, based on Multi-Agent Abstract State Machines, was proposed. The authors claim that their technique in comparison to the one given in [5] using LOTOS, provides a more abstract and flexible formalization approach. Also, authors in [51] state that their ASM rules can be easily modified to accommodate language evolution and reflect new design choices without the need to change the original specifications. In [5], one is required to re-design the mapping between UCM to LOTOS.

Furthermore, Hassine et al. [51], state their ASM-UCM simulation engine may support different concurrency semantics at minimal cost. Agents may behave either in an interleaving semantics with atomic actions (i.e. comparable to LOTOS processes) or in a true concurrency model. The choice of a suitable alternative depends on the application domain and design choice.

The ASM model provides a concise semantics of UCM functional constructs and describes precisely the control semantics. The resulting operational semantics are embedded in an ASM-UCM simulation engine and are expressed in AsmL [82], an advanced ASM-based executable specification language; an overview of the system is shown in Figure 2-7.

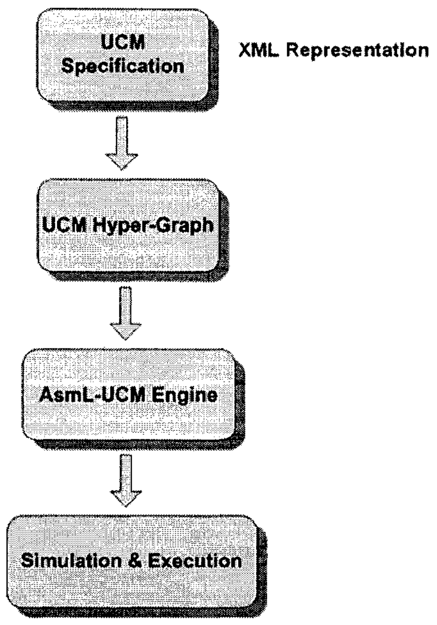


Figure 2-7: UCM_AsmL Overview [51]

The proposed AsmL-UCM approach taken from [51] (Figure 2-8) provides an environment for executing and simulating UCM specifications and creating the simulation traces. In order to apply ASM rules, the UCM specification (originally described in XML format) should be translated into a hyper graph format according to the syntax defined below.

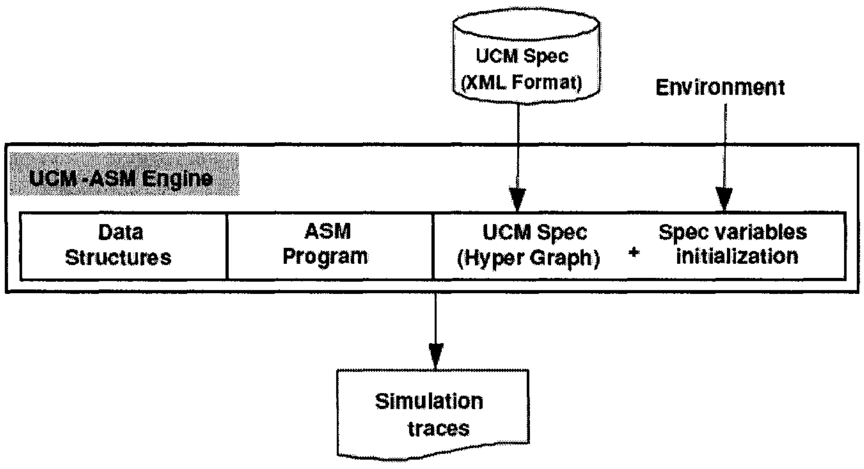


Figure 2-8: ASM-UCM Simulation Engine Architecture [51]

The definition of the ASM formal semantics consists of associating each UCM construct with an ASM rule (a hyper graph format) to model its behavior [51].

A UCM specification is defined as a Hyper-Graph Spec=(C, H, λ)

Where:

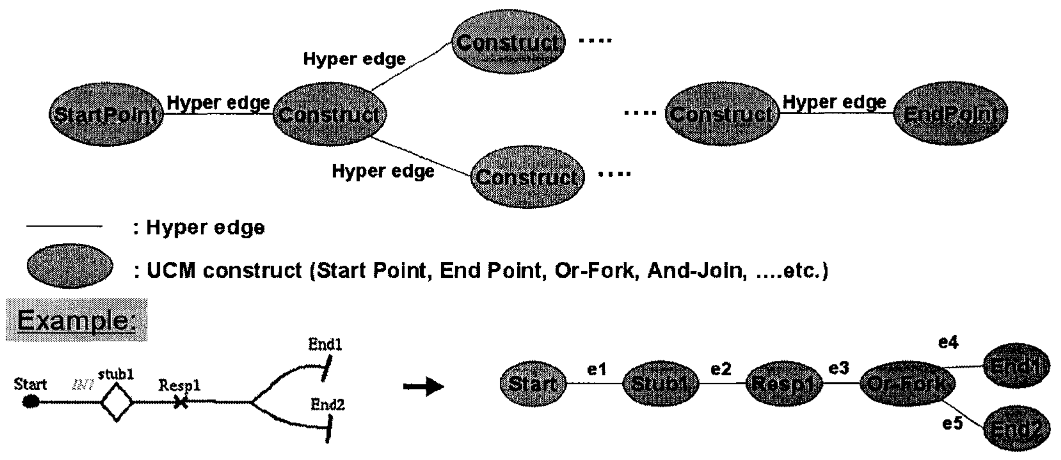
- C is the set of UCM constructs composed of sets of typed constructs.

  $C = R \cup SP \cup EP \cup AF \cup AJ \cup OF \cup OJ \cup AF \cup ST \cup Tm \cup ST \cup$...etc

  where R: Responsibilities, SP: Start Points; EP: End points; AF: AND-fork; AJ: AND-join; OF:OR-fork; OJ : OR-Join; AF: AND-fork; ST: Stubs...etc.

- H is the set of hyper-edges

- λ is a transition relation (path connection) defined as: $\lambda = C \times H \times C$



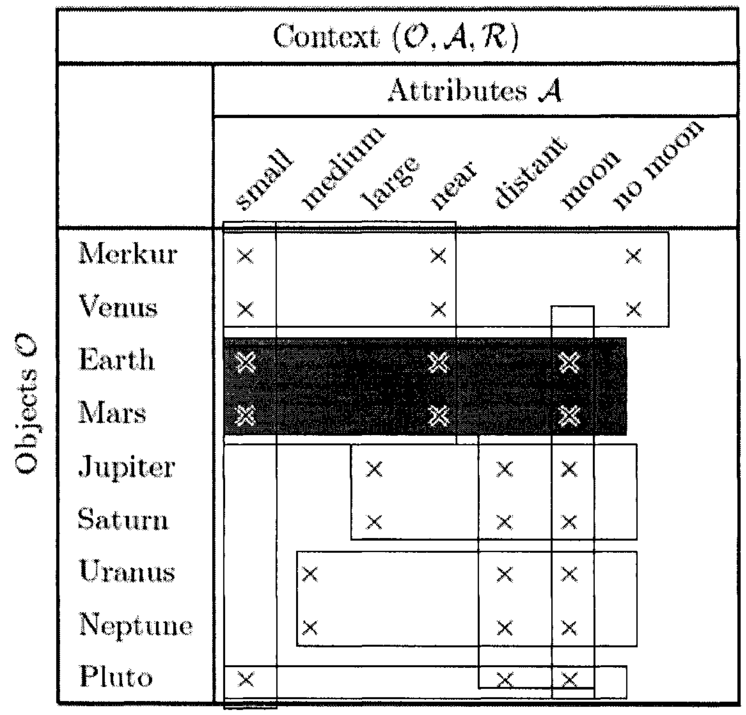
UCM= {(Start,e1,Stub1), (Stub1,e2,Resp1), (Resp,e3,Or-Fork), (Or-Fork,e4,End1), (Or-Fork,e4,End2)}

Figure 2-9: UCM Hyper_ Graph [51]

In Figure 2.9, a simple UCM example and its representation as a Hyper –Graph [51] are presented.

2.2 FCA

Formal Concept Analysis (FCA) [41] is a mathematical approach that dates back to Birkoff in 1940 [14]. FCA is commonly used for representing and analyzing information by performing logical grouping of objects with common attributes. An FCA *context* is a triple C= (O, A, R) where *O* represents a set of objects and *A*, a set of attributes, with *R* $\subseteq$O x A being a relation among them [71]. A context in FCA is normally represented as a relation or a context table, where rows represent objects and columns their attributes. In the context table (Figure 2-10); cells marked with an "x" in each row indicate attributes associated with a particular object.



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

Figure 2-10: **Context Table [71]**

In a relation *R*, the *concept (O, A)* corresponds to the maximal set of objects (extent) that share a set of attributes (intent). For example, in Figure2-10 there exists a concept with *Earth* and *Mars* as its objects and *small*, *near,* and *moon* as its attributes (shown highlighted). The goal of FCA is to group concepts in such a way that no other object, outside of an identified concept, contains the same attributes and no other external attribute can be ascribed to other objects of that group.

Table 2.1 lists all the concepts from the context table. Each concept is a set of objects with common attributes. For example, Concept 4 is a pair of the object set, {*Merkur, Venus*}, and the attribute set containing {*small, near, nomoon* }. It can be interpreted that *small, near*, and *nomoonl* are all members of the attribute sets of *Merkurs* and *Venus.*

Likewise, Concept 8, ({*Earth, Mars, Pluto*}, {*moon, small*}) indicates that in the attribute sets of *Earth, Mars,* and *Pluto,* all contain *moon* and *small.*

One of the major advantages of FCA is its ability to visualize relationships between sets of objects and their common attributes, i.e. "concepts" as a hierarchical graph or "concept lattice". These lattices contain the complete information of the concept structure. In the concept lattice (Figure 2-11); the italic names represent attributes, while the others represent objects. Concept lattices are typically represented with either redundant or non-redundant (sparse or reduced) labeling [71, 42].

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

Table 2-1: Table of Calculated Concepts Corresponding Figure 2-10

Concept lattices using redundant-labeling display the complete list of objects and attributes for each concept. Even though redundant labeling ensures all objects and attributes of each concept are shown, it can cause information-overloading, resulting in lattices that are too difficult to comprehend. Lattices with non redundant-labeling address this problem by presenting each object and each attribute only once in the lattice [42] (shown in Figure 2-11).

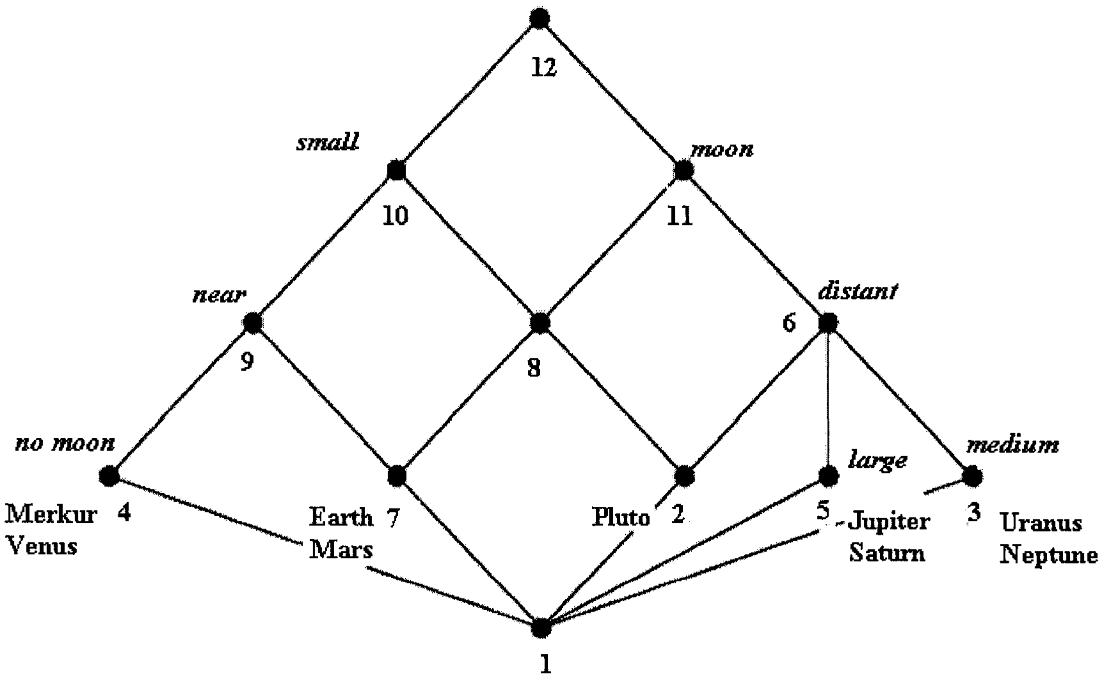


Figure 2-11: Resulting Concept Lattice [71]

When interpreting a concept lattice, all attributes are passed down and objects are passed up [71]. For example, concept #7 can be obtained by passing down the attributes small, near and moon, and passing up the objects Earth, and Mars, therefore being written as 7: ({Earth, Mars}, {small, near, *moon*}). The bottom concept in a concept lattice - in this case (#1) - contains those objects with all the common attributes, while the top concept (#12) contains those attributes shared by all objects.

Formal Concept Analysis has gained popularity in recent years due to: (1) its programming language and application domain independence which allows users to easily define different views (by varying the object, attribute combinations), (2) its availability of tool support to automatically generate context tables and lattices, and (3) the fact that it is a relatively inexpensive analysis, particularly when compared to other more traditional dynamic dependency and trace analysis techniques. However, it must be noted that FCA does not consider semantic information during the analysis step thus limiting its applicability for certain analysis tasks. Furthermore, its flexibility in selecting different attribute-object pairs can result in difficulties selecting an appropriate context for a particular analysis task.

FCA has been applied in various domains such as medicine, linguistics, psychology, mathematics, industrial engineering and computer science. More specifically, FCA-based applications are used for conceptual clustering, ontology engineering, information retrieval, and software engineering [75, 114].

Within the software engineering community, FCA has various applications in program comprehension and maintenance applications [113]. Among these applications are re-engineering of class hierarchies [109], re-engineering of configurations [108], software component retrieval [71], identifying modules [94] and objects from legacy code[73], model restructuring [115], and suggesting refactoring to correct certain design defects[85].

2.3 Analysis Techniques

In this section, we review existing analysis techniques supporting software evolution related to our research. We focus mainly on impact analysis, regression testing, feature interaction and bad smells detection at different levels of abstraction.

2.3.1 Impact Analysis

When a change occurs in a system, regardless whether it is at the requirement, design or code level, such a change will introduce potential defects in a system [2]. Impact analysis focuses on identifying the parts of a system that are (potentially) affected by a proposed modification request. Software Change Impact Analysis (CIA) can be defined as "the determination of potential effects to a subject system resulting from a proposed software change" [11] [17].

Change impact analysis is a process which typically involves several steps. Figure 2.12 presents a generic impact analysis process as discussed in [11], illustrating the inputs and outputs involved in this impact analysis approach. The process begins with a real world modification request that is reduced to a re-specified change request and is the basis for further planning of the change implementation. After the change is re-specified, an initial impact set is determined. Next the IA approach identifies potentially impacted ripple effects related to the modification request. The collection of these impacts and their ripple effects results in the potential impact set of the change, which can then be used to plan, predict measure and accomplish the modification task.

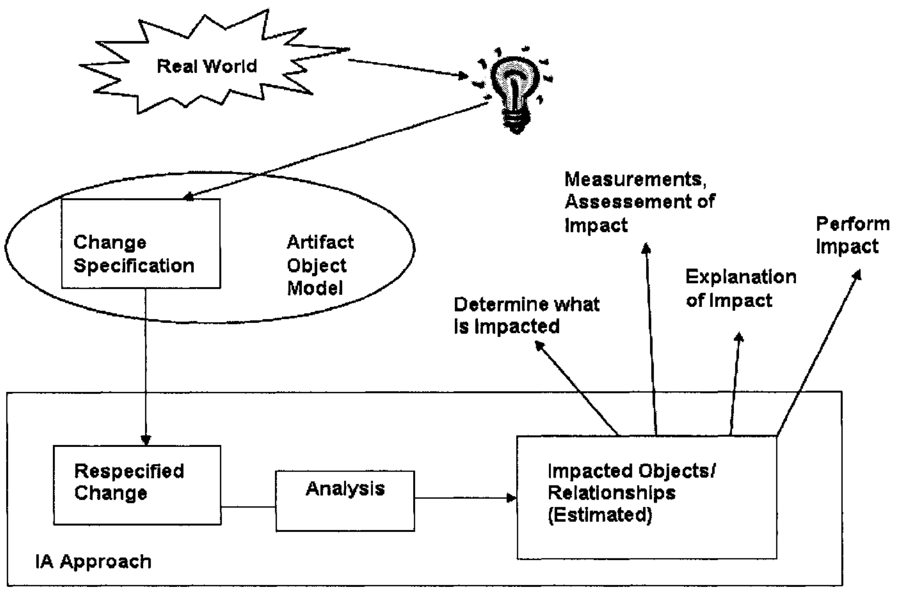


Figure 2-12: Generic Impact Analysis Process [11]

The IEEE Standard 14764-2006: IEEE standard for Software Engineering — Software Life Cycle Processes — Maintenance [57] provides guidelines for the required activities during the Modification analysis phase of software maintenance.

The standard states that impact analysis should [57]:

- analyze Modification Requests(MR) and Problem Reports (PRs);
- replicate or verify the problem;
- develop options for implementing the modification;
- document the MR/PR, the results, and execution options;
- obtain approval for the selected modification option.

These guidelines can serve as a general template for determining what should be accomplished during a modification analysis process.

Most of the existing change impact analysis approaches focus on source code [13, 43, 67, 115] and design level analysis [19, 20].

These approaches can be categorized into two types of impact analysis techniques: traceability analysis [55, 95, 104] and dependency analysis [17]. Traceability analysis focuses on the linking of software life cycle objects (SLOs) generated or modified at different abstraction levels, e.g. requirements or designs to the source code [17]. Figure 2-13 present an example of a traceability model showing the links between the system artifacts [55].

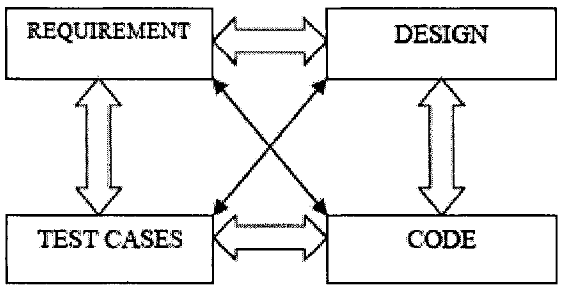


Figure 2-13: A Meta -Model of a Traceability System [55]

Dependency analysis typically focuses on identifying how objects at the same abstraction level are related to one another [17]. A simple schema for dependency analysis is shown in Figure 2-14 [48].

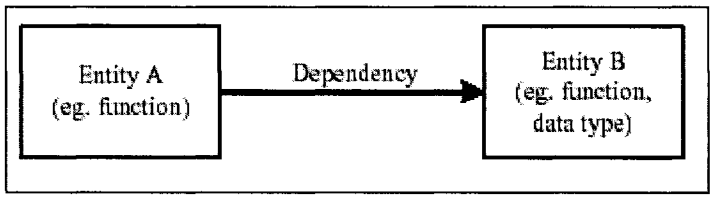


Figure 2-14: A Simple Schema for a Dependency Analysis [48]

There exist other approaches that support syntactical dependency analysis such as static [119], dynamic slicing techniques [63], or hybrid versions which combine both static and dynamic dependency analysis. Common to these source code-based approaches is their computational complexity (e.g. slicing techniques), and their often limited applicability, because they are restricted to a specific programming language.

Other dependency-based impact analysis approaches, such as expert judgment and code inspection, are difficult to automate, often incorrect [74], and also very expensive [90]. There exists only limited work on applying slicing-based dependency analysis at a higher model level [64].

Existing work in change impact analysis and regression testing has focused mainly on identifying changes at the source code level [67] [55]. These source code-based approaches typically result in an accurate analysis of change impacts [17], due to the fine-grained granularity of the information available at the source code level as well as the fact that the source code represents the requirements implemented. However, these approaches also tend to be time consuming and require an understanding of both the system requirements and their implementations which makes them less applicable for management and non-technical decision-makers.

2.3.2 Regression Testing Methodologies

One necessary, but costly maintenance activity is regression testing [96]. After a modification is implemented, a system has to be validated to ensure the modified parts have not introduced any new errors into previously tested code [44]. One approach to re-using test suites is called *re-test-all* and it requires re-running all test cases in the existing test suite. This approach, however, is typically too expensive and unnecessary. In most cases, except for the rare event of a major re-write, changes only affect parts of a system and do not have any effect on the remaining parts of the system. Selective regression testing allows for a reduction of the number of test cases to be re-executed, and therefore, reducing the cost associated with testing, by identifying the subset of test cases that are relevant and have to be re-run. Regression techniques always apply some type of impact analysis in order to determine the coverage needed by the selected regression tests. In [96], a typical selective re-test solution is presented consisting of the following steps:

Given a program *P*, its modified version P', and test set *T* used previously to test *P*; the goal is to find a way to re-use T which provides sufficient confidence in the correctness of P'.

1. Select $T' \subseteq T$, a set of tests to execute on P'
2. Test P' with T', to establish the correctness of P' with respect to T'
3. If necessary, create T'', a set of new functional or structural tests for P'
4. Test P' with T'', to ascertain the correctness of P' with respect to T''
5. Create T'', a new test suite and test history for P', from *T*, T', and T''

The goal of such a regression test is to identify T' and to re-use these parts of T that provide sufficient confidence in the correctness of P' [96].

Various types of regression test techniques are proposed in the literature [98] to improve cost-effectiveness. Four of these techniques involve re-using existing test cases and are discussed further below.

Re-test All

This technique provides the most conservative approach for regression testing. Re-testing all requires re-executing all tests *T* from the test suite in order to cover unnecessary parts of P' that might not be affected by a modification, and therefore, would not need to be re-tested [98]. Indeed, the cost associated with re-running all test cases can be very expensive [111].

Regression Test Selection (RTS)

For regression test selection, test cases are re-used selectively (e.g., [29]) by identifying only the subset of an existing test suite needed to re-test the modified program *P'*. RTS techniques are typically based on the assumption that the test selection is safe (all affected parts are re-tested) and the overall approach is more efficient than a re-test all approach. While safe RTS techniques [102, 29] guarantee that, under certain conditions, test cases not selected could not have exposed errors in the modified program; they might require inclusion of a larger number of test cases in comparison to non-safe RTS techniques. Non-safe RTS techniques [e.g. 98, 46] relax the safety (completeness) rule to improve the efficiency (speed) of their algorithms used to identify test cases. The cost of re-test all regression testing can be reduced by selecting a subset of possible test cases focusing mainly on the parts of the system which have been modified or are most likely to be impacted by a change [112, 48].

Coverage Technique

Coverage techniques for test selection differ from minimization techniques by not focusing on the minimization of the test cases; instead they aim to maximize the test coverage. Therefore, the goal of coverage techniques is to identify all the tests that exercise either modified or affected program parts. Examples for coverage techniques approaches can be found in [122], and [97].

Prioritization Technique

This technique orders an application's set of test cases so that those test cases which are better at achieving testing objectives are executed earlier [99, 111]. By re-testing these high priority tests first, the overall confidence in the system grows quickly initially, before leveling off.

2.3.3 Feature Interaction

From a maintenance perspective, an ideal software system would consist only of features that work and function independently from each other. Such a system would allow features to be modified or added without affecting other features in the system. However, reality differs, and typically one has to deal with situations where one feature interacts/depends on other system features. Furthermore, certain (sometimes unexpected) behavior might only occur when two or more features are combined together and would need to interact with each other [27] [32] [66]. Therefore, we can define feature interaction (FI) informally as:

(1) the situation where one feature modifies or subverts the desired operation of another feature or;

(2) when a system functions incorrectly due to the presence of certain features.

In the past, most of the research in FI analysis has focused on the telecommunications domain [9]. Telephony features are usually complex and difficult to design and implement. The specification of features written in a natural language (e.g. English) can be unclear or ambiguous and may be subject to interpretation. As a result, independent implementations of the same feature may be incompatible. There is a need for a notation to help designers describe, understand and analyze system features. Several notations have been introduced to describe system features (Chisel Diagram [3], Finite State Machines [15], LOTOS [38], Use Case Maps (UCM) [4], etc.). In this work, we focus on UCMs as a feature description language and illustrate its use to describe system features in the telecommunication domain. However, FI has also been studied in other application domains, such as [32]. An example of feature interaction in a telecommunication system is when a callee is subscribed to both VM (voice mail) and CFBL (call forwarding on busy line) when the line is busy. Suppose that caller A dials B, who is subscribed to both VM and CFBL busy. In the case where B is busy, the system cannot determine whether CFBL or VM should be activated. As a result, we have in this case a non-deterministic FI occur (Figure 2-15).

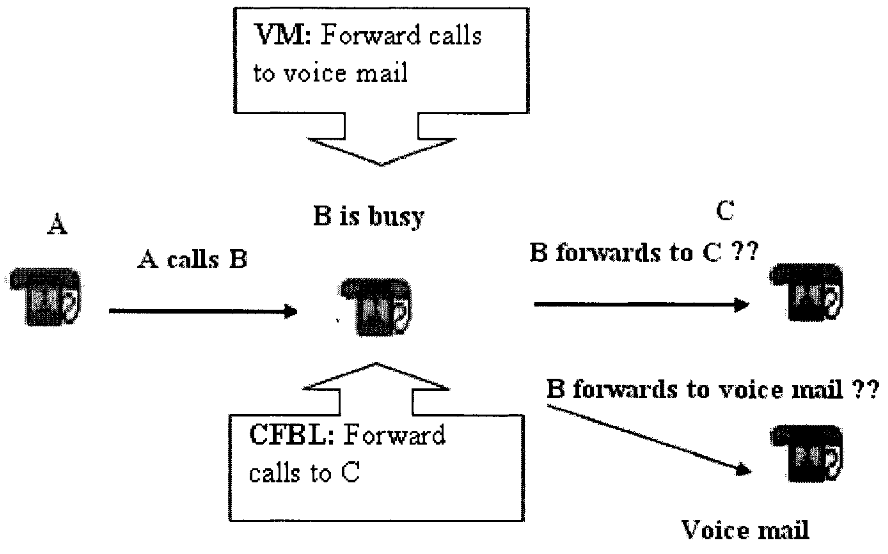


Figure 2-15: Illustration of a Telephony Feature Interaction

It should be noted that in most systems, feature interaction is inevitable and, in fact, necessary as little can be achieved by independent features. Since many feature interactions are intentional by design, determining if a specific feature interaction is intentional, or the result of a specification fault, typically involves some type of application domain knowledge and is outside the scope of this thesis.

2.3.3.1 Addressing feature Interaction

Existing work on feature interaction can be classified in three main problem areas: Avoidance, Detection, and Resolution [28].

Avoidance

For the avoidance of feature interactions, the objective is to prevent the manifestation of unwanted interactions by defining additional guidelines and constraints. The avoidance approach assumes the causes of the interactions in a system are known. However, the required domain knowledge and insights related to the cause of the feature interaction might not be available. Avoidance of feature interaction analysis is typically applied in the early phases of software lifecycle when features are specified and designed. An example of feature interaction avoidance is presented in [47], where by using a judging algorithm at run-time, the authors propose a technique to prevent service initiation only when a terminal assignment causes feature interaction. Another example is shown in [40], where a Wireless Intelligent Network (WIN) system feature interaction problem is solved (partially) by assigning pre-defined priorities to different features [40].

Detection

The goal of detection of feature interaction is to determine whether or not "a set of independently specified features", once composed to a system, can cause conflicts that might lead to undesirable side effects. Feature interaction detection can be applied on various artifacts created throughout a software lifecycle (e.g. requirements, specification, source code) leading to a variety of feature interaction detection approaches [8, 25, 26].

Resolution

The objective of feature interaction resolution is to find solutions to undesired feature interactions within a system [e.g. 22, 25, 26, 47]. Two of the proposed techniques for resolving feature interaction are [22, 26] which focus on:

- replacing the undesired behavior by a reasonable one;
- creating a protocol between the features involved in the interaction consisting of an exchange of necessary information to avoid the interaction.

The detection of all possible feature interactions is often an inherently difficult and expensive task due to possible presence of large number of feature combinations and scenarios, executing these features [68] within a system. Thus, there is a need to reduce the complexity of the analysis for such feature rich systems by limiting the analysis to the scenarios and features that are either affected, or prone to be affected, by a feature modification request. In the context of this thesis we are focusing on the detection of feature interactions.

2.3.4 Bad Smells

2.3.4.1 Background on Bad Smell and Refactoring

The notion of bad smells in source code originated by Fowler and Beck [39], who introduced an initial set of 22 bad smells for object-oriented programs. This set of bad smells was further classified and categorized in [77], providing a taxonomy and grouping of bad smells. Bad smells are generally associated with bad design and/or bad programming. Bad smells do not always imply that the code or the design is wrong; they can be considered as an indication of high complexity which may lead to a serious problem in the future [31]. Bad smells identification can be applied by developers and maintainers to determine parts of software design that would benefit from a set of restructuring, or in the specific case of object-oriented system, refactoring [76] rules. Bad smells can be removed by using these refactoring techniques to improve the quality of the software [92]. Refactorings are modifications made to programs, models or specifications in order to advance their structure, and thus making them more comprehensible, readable and extendable. Refactorings are required to preserve the external behavior of the program/ design model/specification while enhancing their internal structure.

Chikofsky and Cross in [30] classify restructuring as "the transformation from one representation form to another at the same relative abstraction level, while preserving the subject system's external behavior (functionality and semantics)". A restructuring transformation is often one of appearance, such as altering code to improve its structure in the traditional sense of structured design. While restructuring creates new versions that implement or propose change to the subject system, it does not normally involve modifications because of new requirements. Restructuring can improve the comprehension of the subject system by suggesting changes that can improve structural aspects of the system [30]. Opdyke in his PhD dissertation [88] , originally introduced the term refactoring as an object-oriented variant of restructuring; and Fowler et al. , define refactoring as "the process of changing a [object-oriented] software system in such a way that it does not alter the external behavior of the code, yet improves its internal structure" [39]. When introduced at the source code level or design level, the key idea here is to reorganize classes, variables, and methods across the class hierarchy in order to improve complexity of the system and to facilitate future modifications and evolvability [77].

2.3.4.2 Detection of Bad Smells

Most of the existing work in the area of bad smells has focused on the identification of such smells at the source code [39, 77, 76, 83] and design level [33, 86]. Mens et al. [83] introduced a comprehensive overview and classification of existing work in the area of software restructuring and refactoring. More recently the notion of bad smells and refactoring/restructuring has been extended to the formal requirement specification level [31, 37, 78, 79. 80]. In [31, 37, 78, 79, and80] bad smell detection and refactorings have been applied to Object-Z [34], which is an object-oriented extension of the state-based formalism Z [123] used for requirements specification. In [78] the focus is on refactorings that are applicable on single methods only (e.g. extracting methods, simplifying conditions, substituting algorithms). Common to the refactoring techniques mentioned in [37, 79, 80] is that they all focus on the refactoring of object-oriented specifications (written in Object-Z).

The work of Russo et al. [101] proposes the restructuring of natural language requirements specifications by breaking these specifications down into a viewpoint structure. Each viewpoint contains partial requirements of some of the components within the system, and interactions between these viewpoints are explicitly shown. The goal of this restructuring technique is to increase requirements comprehensibility, enabling detection of inconsistencies, and managing requirements evolution.

Ciemniewska et al., in [31], demonstrate another approach to detect bad smells at the requirement specification level, by presenting a technique for supporting use-case reviews based on natural language processing (NLP) tools. Their intention is to find "easy-to-detect defects" (such as duplicate use-cases in the document, inconsistent naming styles of use cases, complex sentence structure in use cases, etc) automatically.

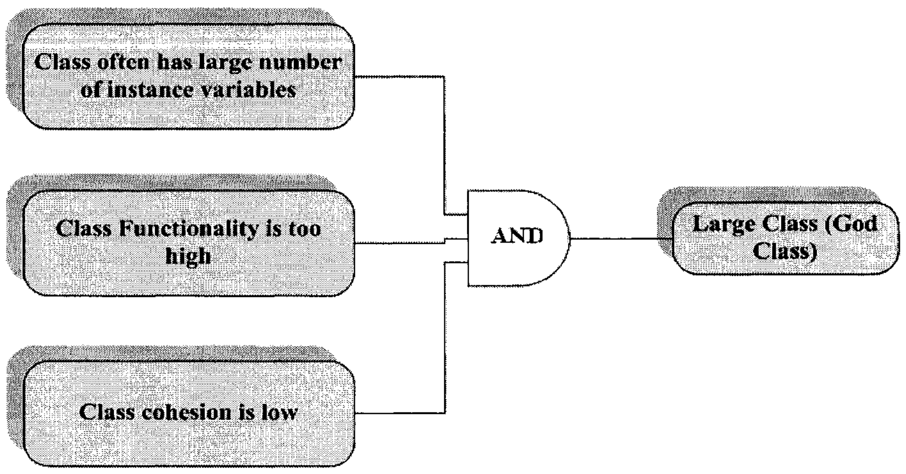
2.3.4.3 Bad Smells Categories and Refactorings

In this research we briefly described all six bad smell categories introduced in [77] and provide a concrete example of a bad smell for each of these categories.

1. The Bloater Category

The Bloater category includes bad smells that "represents something that has grown so large that it cannot be effectively handled" [77]. Bad smells that fall into the Bloater category are: *Large Class, Large Method, Primitive Obsession, Long Parameter List,* and *Data Clumps* [77].

Name of Bad Smell:	Large Class
Description:	Classes that are trying to do too much often have many instance variables and can also suffer from code duplication.


Class often has large number of instance variables
Class Functionality is too high
Class cohesion is low
AND
Large Class (God Class)

Example [1]: The `Person` class includes too much functionality that involves different data structures and access functions to these structures

```
class Person...
    public String getName() {
        return _name;
    }
    public String getTelephoneNumber() {
        return ("(" + _officeAreaCode + ") " + _officeNumber);
    }
    String getOfficeAreaCode() {
        return _officeAreaCode;
    }
    void setOfficeAreaCode(String arg) {
        _officeAreaCode = arg;
    }
    String getOfficeNumber() {
        return _officeNumber;
    }
    void setOfficeNumber(String arg) {
        _officeNumber = arg;
    }

    private String _name;
    private String _officeAreaCode;
    private String _officeNumber;
```

Figure 2-16: Example of a Class with too Many Functionalities

Solutions: Eliminate redundancy in the class itself by using *Extract Class* (by using *Move Method*, and *Move Field*) or *Extract Subclass* [39]. An example is provided below.

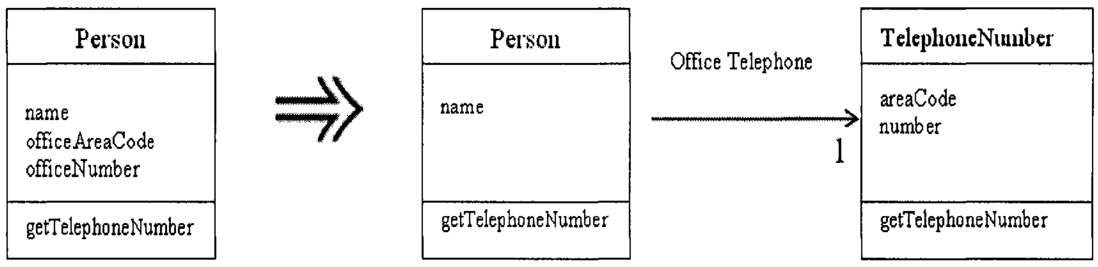


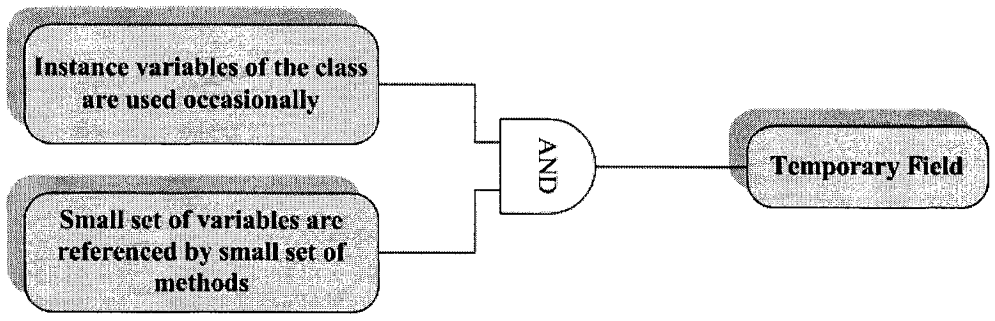
Figure 2-17: Example of Extract Class Solution [39]

2. Object-Oriented Abusers Category

The bad smells in this category are all related to cases in which good OO design is not fully supported in the solutions. Examples of bad smells in this category are *Switch Statements, Temporary Field, Refused Bequest, Alternative Classes with Different Interfaces,* and *Parallel Inheritance Hierarchies* [77].

Name of Bad Smell: Temporary Field

Description: Sometimes in an object, instance variables are set only in certain circumstances. Such code is difficult to understand, because it is expected that the object requires all of its variables.


Instance variables of the class are used occasionally
Small set of variables are referenced by small set of methods
AND
Temporary Field

Example [77]: The `textRep` field is not an essential part of the Person class and it is only used in the `toString()` method.

```
public class Person {
      private String name;
      private String phoneNumber;
      private StringBuffer textRep;
      ...

      public Person(String name) {
            this.name = name;
      }
      public void setPhoneNumber(String phoneNumber) {
            this.phoneNumber = phoneNumber;
      }
      public String getPhoneNumber() {
            return phoneNumber;
      }
      public String toString() {
            textRep = new StringBuffer();
            textRep.append(name);
            textRep.append(phoneNumber);
            textRep.append(...);

            return textRep.toString();
      }
```

Figure 2-18:Example of a Class with Temporary Fields

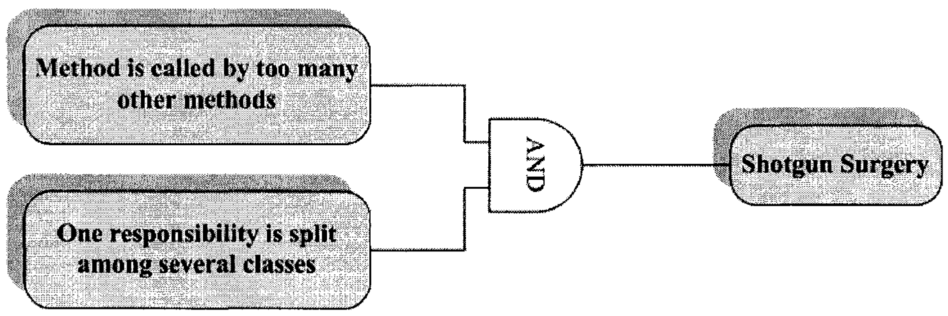
Solutions: It is suggested to use *Extract Class* refactoring in order to create a class for "orphan variables" [39]. Every related code that concerns the variables should be put together into the component. Furthermore, it "might be possible to eliminate conditional code by using *Introduce Null Object* to create an alternative component for when the variables aren't valid." [39]. In the example, refactoring *Extract Class* can be used to move the `textRep` to a new class.

3. Change Preventors

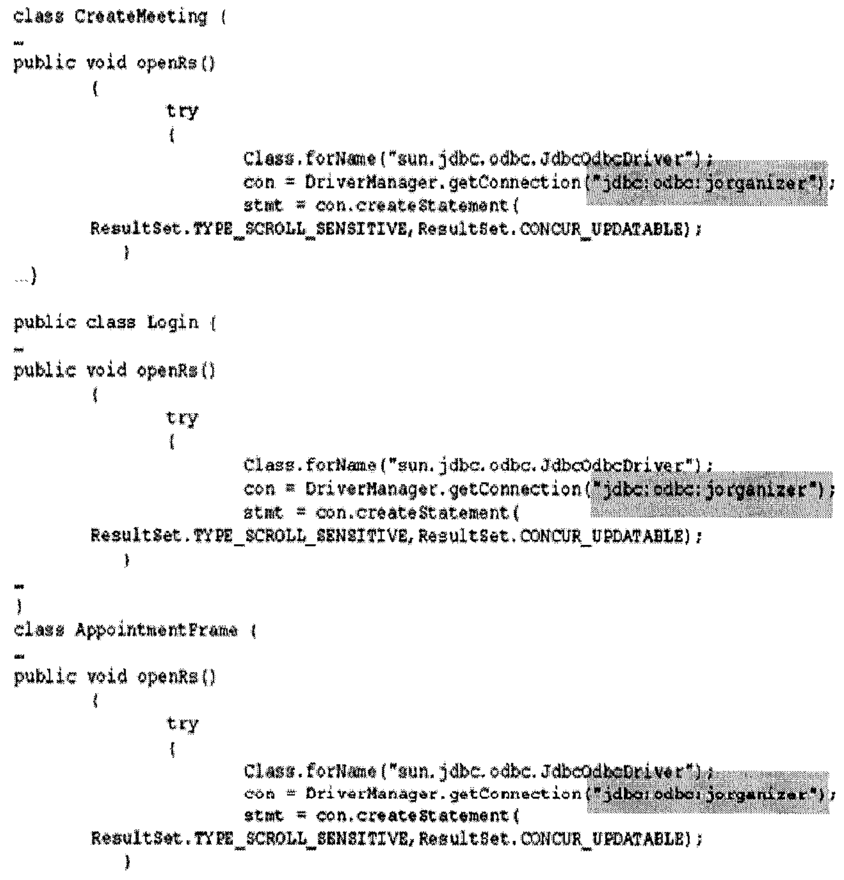
Bad smells in this category are code structures that hinder the modification of the software. The smells in the Change Preventers category are *Shotgun Surgery and Divergent Change* [77].

Name of Bad Smell: Shotgun Surgery

Description: When a change in the program requires lots of code changes in many different classes making it hard to find all the right places that would need changing.


Method is called by too many other methods
One responsibility is split among several classes
AND
Shotgun Surgery

Example: The `openRs` method in Figure 2-19 is responsible for the database access and used in every class. Therefore a change to the database name would result in a change affecting every class containing the `openRs` method.



```
class CreateMeeting {
...
public void openRs()
        {
                try
                {
                        Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
                        con = DriverManager.getConnection("jdbc:odbc:jorganizer");
                        stmt = con.createStatement(
        ResultSet.TYPE_SCROLL_SENSITIVE,ResultSet.CONCUR_UPDATABLE);
           }
...}

public class Login {
...
public void openRs()
        {
                try
                {
                        Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
                        con = DriverManager.getConnection("jdbc:odbc:jorganizer");
                        stmt = con.createStatement(
        ResultSet.TYPE_SCROLL_SENSITIVE,ResultSet.CONCUR_UPDATABLE);
           }
...
}
class AppointmentFrame {
...
public void openRs()
        {
                try
                {
                        Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
                        con = DriverManager.getConnection("jdbc:odbc:jorganizer");
                        stmt = con.createStatement(
        ResultSet.TYPE_SCROLL_SENSITIVE,ResultSet.CONCUR_UPDATABLE);
           }
```

Figure 2-19: Example of a Code Affecting Several Classes

Solution: In this case the *Move Method* and *Move Field* refactoring can be applied (Figure 2-20) to gather all the changes into a single class. However, when no current class seems as a good candidate, creating a new class is a better option [39].

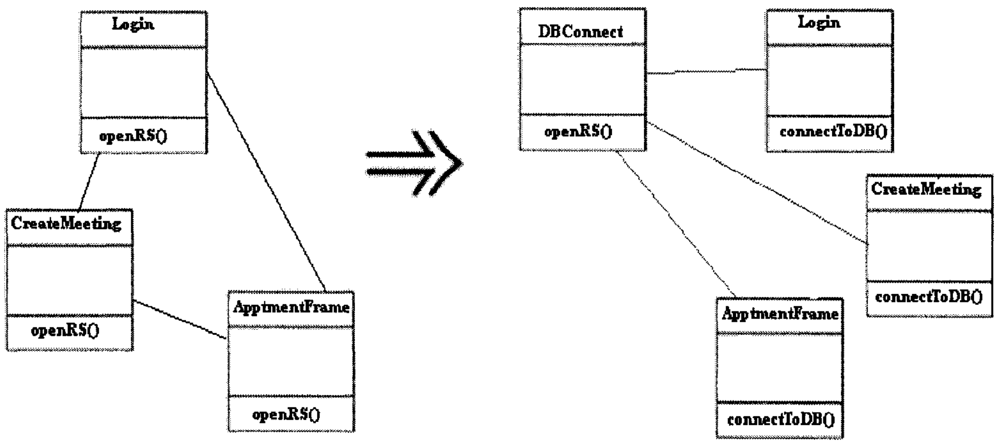


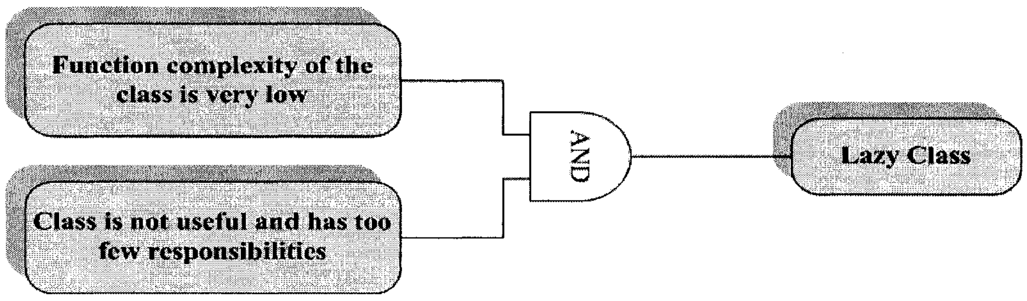
Figure 2-20: Example of Refactoring by Creating a New Class and Move Method Technique

4. Dispensables Category

This category mainly addresses those bad smells which are a result of unnecessary elements in the source code. Members of this category include: *Lazy Class, Data Class, Duplicate Code, and Speculative Generality* [77].

Name of Bad Smell: Lazy Class

Description: These are mainly classes that are providing limited functionality caused either by downsizing due to previous refactoring or because they were added to a system for future changes that never took place [39].


Function complexity of the class is very low
Class is not useful and has too few responsibilities
AND
Lazy Class

Example: The `TelephoneNumber` class in the example below performs limited functionality only [39].

```
class Person...
   public String getName() {
       return _name;
   }
   public String getTelephoneNumber(){
       return _officeTelephone.getTelephoneNumber();
   }
   TelephoneNumber getOfficeTelephone() {
       return _officeTelephone;
   }

   private String _name;
   private TelephoneNumber _officeTelephone = new TelephoneNumber();

  class TelephoneNumber...
   public String getTelephoneNumber() {
       return ("(" + _areaCode + ") " + _number);
   }
   String getAreaCode() {
       return _areaCode;
   }
   void setAreaCode(String arg) {
       _areaCode = arg;
   }
   String getNumber() {
       return _number;
   }
   void setNumber(String arg) {
       _number = arg;
   }
   private String _number;
   private String _areaCode;
```

Figure 2-21: **A Classes With limited Functionalities**

Solution: These classes can either be removed or changed, using the *Inline Class* refactoring method. If they are subclasses, the *Collapse Hierarchy* refactoring can be applied to collapse the existing structure [39].

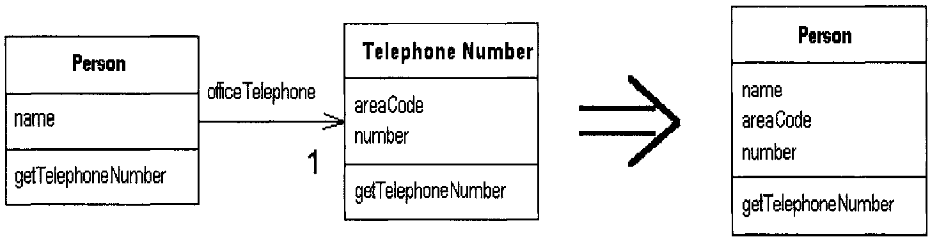


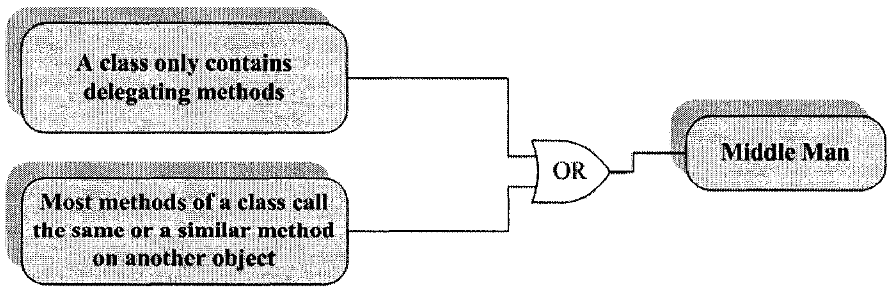
Figure 2-22: Example of Inline Refactoring for a Class with a Few Functionalities [39]

5. Encapsulators Category

In [77], it has been suggested that the Encapsulators category deals mainly with "data communication mechanisms or encapsulation". The smells grouped in this category are *Middle Man* and *Message Chains.* It should be noted that the authors in [77] assert that *Message Chain* could belong to a different category (Couplers) and *Middle Man* could also be added to the Object-Oriented Abusers category. However, since both are dealing with the way in which objects, data, or operations are accessed, they have been assigned to a separate bad smell category.

Name of Bad Smell: Middle Man

Description: When a delegate class is performing no useful or extra work and not really contributing to the application [39].


A class only contains delegating methods
Most methods of a class call the same or a similar method on another object
OR
Middle Man

Example: In the example below (Figure 2-23), `Department` is encapsulated within the Person class, thus to find a person's manager, there is a need to delegate functionality to the `Department` class [39].

```
class Person...
    Department _department;
    public Person getManager() {
        return _department.getManager();

  class Department...
    private Person _manager;
    public Department (Person manager) {
        _manager = manager;
    }
```

Figure 2-23: Example of an Unnecessary Delegation

Solution: In general, the *Remove Middle Man* refactoring can be used to restructure the class in a way that client calls are delegated directly. However, sometimes "delegates are created to provide a sort of façade, keeping the details of messages and structures hidden from the caller" [33]. Thus, it is a good practice to first determine the intention of the delegation, and then take action.

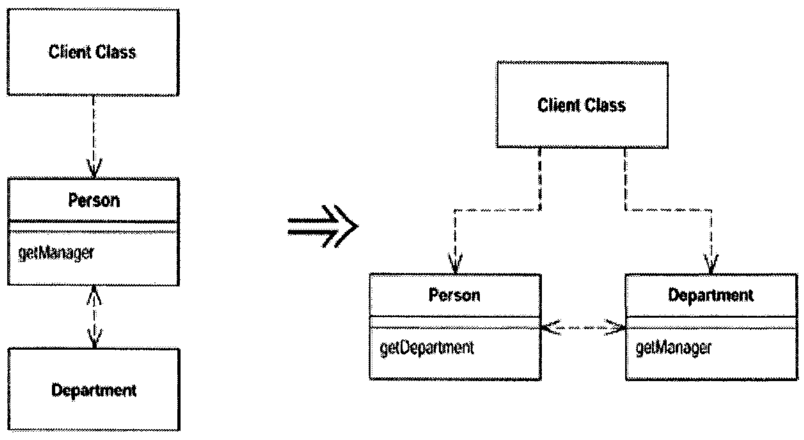


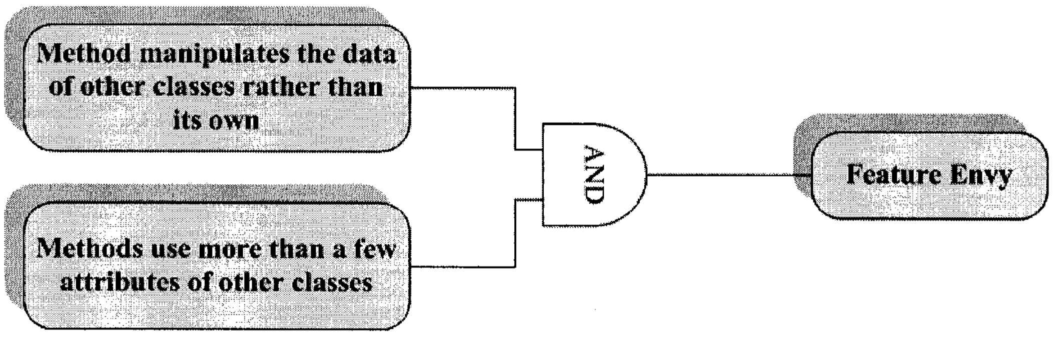
Figure 2-24: Example of *Middle Man* Bad Smell and Remove *Middle Man* Refactoring [39]

6. Couplers Category

Bad smells in this category focus on the coupling among classes. However, the authors in [77] believe these bad smells also could be grouped in other categories. The two bad smells in this group are *Feature Envy and Inappropriate Intimacy.*

Name of Bad Smell: Feature Envy

Description: This is the case when a method is more interested in using the attributes of a class other than the one it is actually in.


Method manipulates the data of other classes rather than its own
Methods use more than a few attributes of other classes
AND
Feature Envy

Examples: In the code snippet below which is taken from [77], the `write` method in class `FigureWriter` is only using features from classes other than its own.

```
public class Figure {
      ...
      public String getName() {
            return name;
      }
      public String getDimensions() {
            return dimensions;
      }
      public String getColor() {
            return color;
      }
      public void update() {
            ...
      }
}
public class FigureWriter {
      ...
      public String write(Figure fig) {
            StringBuffer result = new StringBuffer();
            fig.update();

            result.append(fig.getName());
            result.append(fig.getDimensions());
            result.append(fig.getColor());
            return result.toString();
      }
}
```

Figure 2-25: Example of a Class Using Only Features of Other Classes

Solution: In order to eliminate this smell, *Move Method* refactoring can be applied (Figure 2-26). In the case where only parts of a method are misplaced, the *Extract Method* will be an option [39].

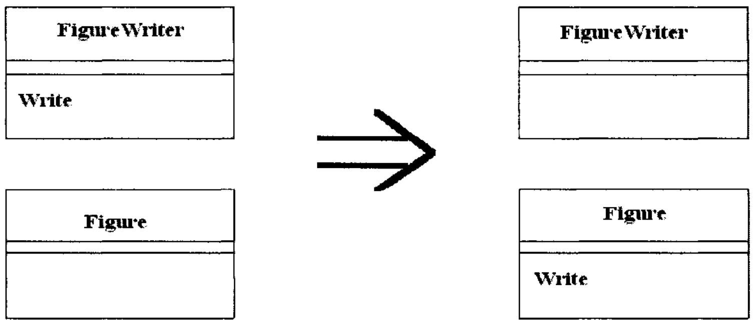


Figure 2-26: Example of *Move Method* Refactoring Feature Envy Bad Smell

3. A UCM Requirements Evolvability Framework

Maintenance processes (e.g., [57]) have been established to guide both managers and maintainers through typical maintenance and evolution tasks. Common to these process models is they require some type of modification analysis (MA) prior to signing up on a particular modification request. During a MA, managers, who are often not familiar with the detailed implementation of a system, have to determine the feasibility, impact, and cost associated with such a modification request. In this chapter, we present a novel framework for supporting modification analysis for early detection of change impact, re-testing effort, feature interaction, and detection of bad smells at the requirements specification level.

In section 3.1, we explain in more detail the motivation behind our research. Section 3.2 presents the research hypothesis and research goals as well as a set of criteria to validate our hypothesis. Section 3.3 describes in detail the requirement modification analysis approach used in this research and illustrates our methodology for each type of analysis. Our major research contributions are summarized in section 3.4.

3.1 Motivation

When developing software systems, it is rare that an initial system design will correspond completely with the final design or implementation of a system [70]. Ever changing customer needs lead to requirements modifications [17]. The efficient management and execution of these changes are critical to software quality and for managing the evolution of software systems [17]. Modification analysis, an essential phase of most software maintenance processes, requires decision-makers to perform and predict potential change impacts, feasibility and costs associated with a modification request. The majority of existing techniques and tools supporting modification analysis focus on source code level analysis and require an understanding of the system and its implementation. In this research, we present a novel approach to support the identification of potential modification and re-testing efforts associated with a modification request without the need for analyzing or understanding the system source code. Our research is motivated by the lack of a requirement level framework which: 1) supports early modification analysis without the need to comprehend the underlying source code; 2) can perform requirement modification analysis at different levels of abstraction and granularity and; 3) provides a visualization of the analysis results to guide both management and maintainers during their decision-making process.

3.2 Research Hypothesis and Research Goals

3.2.1 Research Hypothesis

In this research we present a new modification analysis methodology which utilizes formalization of a requirement modeling technique, Use Case Maps, combined with a formal sensible grouping method, Formal Concept Analysis, in order to assist the managers and maintainers during their early decision-making process.

> *Research Hypothesis:*
>
> A Requirements Evolution Analysis Framework can be developed to assist decision-makers in supporting modification request analysis and evolution of requirements specified in the Use Case Map language.

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

Formalization of UCM

A UCM specification integrates multiple scenarios, some with separate starting points and others that share starting points but under different types of input data or different system states. We adopt the UCM scenarios definition introduced by Hassine et al. [51] and describe system level scenarios as being end-to-end scenarios, where each scenario starts at a start point and ends at an end point. System level scenarios make use of a path data model composed of global (Boolean) variables used on guarding conditions. In UCM, abstract syntax and static semantics are informally defined as XML document type definition resulting in UCM diagrams where scenarios, global variables, and stubs have to be analyzed and traced manually. The interpretation of these UCM specifications is therefore typically left completely to the user of these diagrams. In [51], an operational semantics for the UCM language based on Multi-Agent Abstract State Machines was proposed. The operational UCM semantic itself is expressed using AsmL [82] a high-level executable specification language developed by the Foundations of Software Engineering (FSE) group at Microsoft Research. The resulting ASM semantics are embedded in an ASM-UCM simulation engine designed for simulating and executing UCM specifications.

Change Impact Analysis

As discussed in Chapter 2, resolving specification errors and modifying requirements may introduce new collateral errors. Change Impact Analysis (CIA) techniques have been used to assess the impact of changes in an attempt to prevent the introduction of new errors [11, 18]. CIA techniques can be classified into two main categories: traceability [55, 95, 104] and dependency analysis [1, 17].

Among the various CIA approaches, the approach by Tonella et al. [115] is of particular interest to our proposed research. They introduced an approach that combines slicing and formal concept analysis (FCA) to determine the change impact at the source code level. They referred to their approach as "Concept Lattice of Decomposition Slices". The authors claim their technique is an extension of the decomposition slice graph in which the graph is obtained from concept analysis and can provide maintainers with relevant information on dependencies and interfaces to support program comprehension and to assess change impacts at given program points.

At the requirements and architectural level, we expect to adopt some aspects and techniques of existing work on architectural [11, 18, 52, 115] and requirements impact analysis. Requirements traceability is a quite well-established and mature research domain. Hewitt et al. in [52, 54], recommend a change impact analysis approach that makes use of a system's UCMs specification so that the scope of the change may be determined at an early stage of the change life cycle. From FCA-based application presented in [115], we see the potential of the concept lattice to identify all the execution element dependency within UCM specifications. Specifically, we conjecture that a formalized and executable UCM representation provides sufficient information about the structure and behaviour of a system to determine the potential impact of a change through the use of a formal sensible grouping method (FCA) for dependency analysis at the UCM level.

Regression Testing

As software evolves and new modification requests are introduced, test cases are required to be re-exercised to ensure these modifications did not cause any undesirable behaviour of the system. Different regression testing techniques have been adopted to ensure that new changes have not affected other system features both at the source code level [98, 112] and design level [20, 91]. As discussed in [20], regression testing at the source code level is expensive with respect to required time and hardware resources in comparison to regression testing at the architectural level. Regression testing at the architectural/ specification level focuses on high-level test cases, resulting in a typical small number of test cases and therefore reduced cost for the computation and re-execution of these test cases. Often project managers are focusing on an initial prediction of the testing effort and therefore testing cost associated with a particular change before determining the detailed testing strategy at more fine grained analysis levels (source code).

Tallem et al. [112] developed a Delayed-greedy algorithm using FCA which uses both implications of the requirements and test cases, and delays the application until no more reduction of test cases can be performed and all the essential tests are selected. The algorithm uses attributes and object implications of the requirements and test cases with a focus toward minimizing the number of test cases at the source code level.

Currently no selective regression testing technique exists which can be used at the requirement level; however, a FCA-based regression technique has been introduced in [112] and we therefore expect that FCA can also be reapplied to the traces generated by our formalized UCM semantics to identify selective regression tests.

Feature Interaction Techniques

Feature interaction analysis has become an important task for management and maintainers, due to the fact that features might be designed, maintained and evolved at different phases of the software life cycle. For this reason, there is not only a need to automatically detect existing features in a system, but also their interaction [66]. Thus, over the past years, various techniques have been proposed to avoid, detect and solve feature interactions at all stages of the software life cycle, many focusing on the requirement stage including UCM specifications [25, 26, 68].

Eisenbarth et al.'s work focuses on conducting feature analysis by means of Formal Concept Analysis. In [36], they introduce a technique to support program understanding by deriving feature-component correspondence utilizing dynamic information for the part of the program to be understood. Each concept in the lattice represents the common source code portion executed by different usage scenarios, thus proving the lattice of feature-component correspondence can identify all software components (e.g. functions, procedures, statements) that contribute either to a certain feature or all features to which a component contributes. While in our research features will be identified at the requirements (UCM) level, we presume the basic concept introduced in [36] will also be applicable for UCM.

Bad Smell Techniques

The notion of bad smell has mainly been discussed at source code [39, 69] and design level [33, 86]. Bad smells are motivated by the need to identify those parts of a software design that would benefit from a set of restructuring in order to improve the software quality. Various techniques and formalisms have been proposed and used to support restructuring and refactoring activities, for example formal concept analysis [83].

Snelting and Tip in [109] use concept analysis for refactoring object-oriented class hierarchies. Tonella uses FCA to restructure software modules [115], and Moha et al. in [85], use formal concept analysis to "to Suggest Refactorings to Correct Design Defects" and detect cohesion and coupling by means of FCA. They provide a sketch of the target design by grouping methods and fields into cohesive sets and separate classes, to represent a better trade-off between coupling and cohesion [85].

More recently the notion of bad smells and refactoring/restructuring has been extended to include the formal requirement specification level [31, 34, 37] thus motivating us to use the formalized and executable UCM traces and analyze them by means of FCA to detect possible bad smells at the UCM specification level.

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

Null Research Hypothesis:

> The research hypothesis will be rejected if at least one of the validation criteria will not hold and therefore no Requirements Evolution Analysis Framework can be developed to support modification request analysis and evolution of requirements specified in the Use Case Map language.

3.2.2 Sub-goals

As stated in the last section, the general goal of this research is to develop a requirements evolution analysis. In what follows, we divide this general research goal into some more specific sub-goals to be addressed by this thesis:

- Collect traces from the formalized UCM semantics and analyze these traces using FCA to:
    - Support impact analysis of modification requests at the UCM level
    - Perform regression test analysis to identify test cases that have to be re-tested as part of a modification request
    - Identify use case scenarios containing feature interaction by means of FCA

    - Identify the impact of a feature change on other system features
    - Suggest potential bad smells at the UCM level to improve the overall structure and design of the UCM
- Implement a proof of concept tool to support a semi-automatic approach for impact analysis, selective regression testing, feature interaction and bad design detection at the UCM requirements level
- Provide case studies and examples to illustrate both the applicability and limitations of the proposed approach

3.3 Requirement Modification Analysis Methodology

During the analysis of modification requests, decision-makers face several challenges: 1) the ability to determine the potential affect of a modification request on the overall system early in the software maintenance cycle; 2) as part of this decision process, there is a need to identify which UCM scenarios contains which features; and 3) the ability to analyze whether a feature interacts with other features and determine the testing effort related to a particular change. In what follows, we describe our methodology including the major activities to be performed for analyzing and supporting requirements evolution (Figure 3-1).

In step 1, we utilize a formalized UCM semantics to generate scenario traces from the formalized UCMs (Section 3.3.1). In step 2, we define different types of dependencies between the UCM elements captured by the recorded UCM traces to support further modification analysis (Section 3.3.2). In step 3, we combine our UCM dependency analysis and FCA to support modification analysis through impact, feature interaction and regression test selection analysis at the requirements level (Section 3.3.3), which are divided into four types of evolution analysis: Change Impact Analysis (3.3.4), Test Case Selection Results (3.3.5), Feature Interaction Analysis (3.3.6) and Bad Smell Detection (3.3.7).

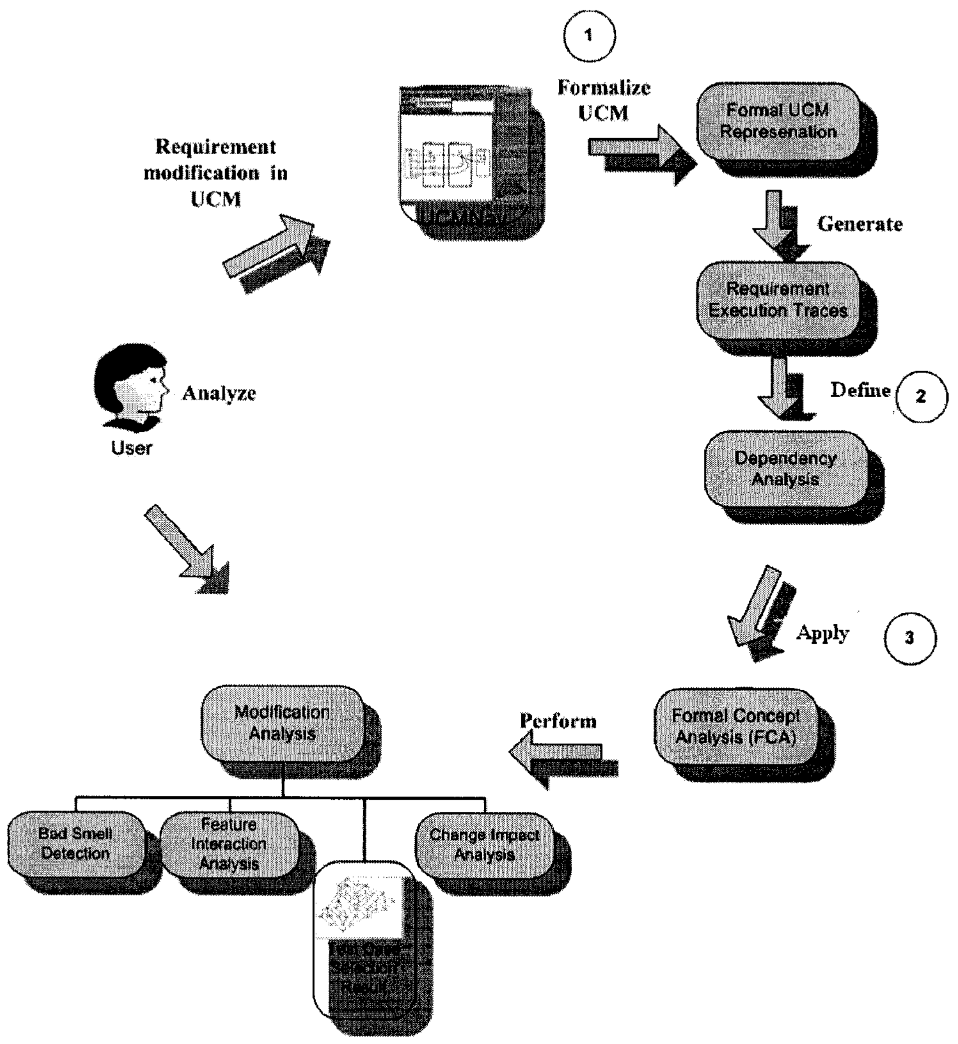


Figure 3-1: Requirement Modification Analysis Methodology

3.3.1 Generation of UCM System Level Scenarios (traces)

In UCM, abstract syntax and static semantics are informally defined as XML document type definitions. This informal representation of UCM diagrams results in a notation that does not support symbolic execution of scenarios captured in these diagrams. In its original form, UCM elements such as scenarios, global variables, and stubs have to be analyzed and manually traced to gain an understanding of the various dependencies that might exist in a UCM. This results in a situation where the interpretation of UCM specifications is left completely to the user. In [50, 51] an operational semantics for the UCM language-based on Multi-Agent Abstract State Machines was proposed. The definition of the ASM formal semantics consists of associating each UCM construct with an ASM rule to model its behavior. The resulting ASM semantics are embedded in an ASM-UCM simulation engine designed for simulating and executing UCM specifications. It is written in AsmL [82] which is a high level executable specification language developed by the Foundations of Software Engineering (FSE) group at Microsoft Research. For a detailed description of UCM semantics, the reader is invited to consult [51]. Thus, as mentioned in [51], a scenario definition contains a name, initial values for the global variables, a list of start points, and (optionally) post-conditions expressed using the global variables. Based on the initial values of the specification's global variables, the ASM-UCM simulation engine generates system level scenarios (traces) which resolve choice points on the OR-forks and on dynamic stubs. In cases where more than one condition evaluates to true, the AsmL-based simulation engine non-deterministically selects one choice [51]. Note that throughout this thesis we use the terms scenario, trace and "system level scenario" interchangeably since traces are a direct result of scenario executions.

A UCM specification integrates multiple scenarios; some with separate starting points and others that share starting points but under different types of input data, or different system states. We define system level scenarios as being end to end scenarios in which each scenario starts at a start point and ends at an end point. System level scenarios make use of a path data model composed of global (Boolean) variables used on guarding conditions. A scenario definition contains a name, initial values for the global variables, a list of start points, and (optionally) post-conditions expressed using the global variables. The telephony system (Figure 2-1) integrates seven features into one single view. A particular combination of features is obtained by initializing the feature specific global variables. Feature global variables can informally be defined as those variables which correspond to a feature plug-in in the system (i.e. subOCS) and allow a user to subscribe to this feature if their value is true during a scenario execution. For instance, to enable a feature F, we subscribe the user to the feature (sub_F:= true) and satisfy its precondition (for instance Busy:=false). Based on these initial values of the global variables, the ASM-UCM simulation engine generates system level scenarios (traces) which resolve choice points found in OR-forks and in dynamic stubs. In cases where more than one condition evaluates to true, the AsmL-based simulation engine non-deterministically selects one choice.

3.3.2 UCM Trace Analysis

In our approach, each scenario represents a test case which executes all UCM scenario elements in a particular scenario. Due to feature interactions, a requirement modification will often affect more than one scenario. Having the traces from the executable UCM allows us to apply dynamic dependency analysis to determine the impact of a modification on the overall system. In our research, we define and apply both feature and execution dependency analysis to answer questions about what and how objects in a UCM are related to one another.

Feature dependency. Scenarios are directly mapped to functional requirements involving one or more system features. Therefore, we can informally define scenarios as being feature dependent when the following holds:

- two or more scenarios represent the same functional features or;
- two or more scenario contain the same sub-scenarios.

We further refine this definition by assuming functional features are represented by UCM plug-ins/stub combinations. We can now state that feature dependency at the UCM level exists if one of the following two conditions holds:

- two or more scenarios contain the same sub-scenarios (share sub-scenarios with the same start and end points) or;
- two scenarios *Sc1* and *Sc2* are feature dependent if they share the same plug-in (feature) P , where $Pa \subset P= \{Pa, Pb \ldots Pz\}$.

One of the main challenges in analyzing functional dependencies in UCM is the use of *dynamic stubs*. Dynamic stubs allow for the specification and visualization of alternative behavior of scenarios. Being able to model such dynamic behavior is gaining more importance due to the widespread use of protocols and communicating entities in most systems. UCMs support the modeling of dynamic behavior through sub-maps, called *plug-ins*, that are associated with a dynamic stub. Conditions (global variables) in dynamic stubs define which plug-in is selected at run-time. A major advantage of formalizing UCM is that these dynamic dependencies of the plugs are resolved dynamically by executing their conditions and calling the respective plug(s) as part of the selected execution path.

Execution dependency. Execution dependency, a more fine grained dynamic dependency analysis, focuses on the interaction of scenarios and shared elements within a model. Therefore, scenarios are execution dependent if they share common elements during their executions. In a UCM context, one can apply scenario execution dependencies at two abstraction levels: component and domain element.

- *Component execution dependency*

From an UCM perspective, a component dependency exists when two or more scenarios share the same component C', where C' $\subset$ {set of UCM components}.

- *Domain element execution dependency*

Domain elements are execution dependent if their scenarios share common Use Case Map domain elements. We base our definition of domain elements on a subset of the elements introduced in [52]. We can now state that E is a set of UCM domain elements where $E = \{SP \cup EP \cup R \cup AF \cup AJ \cup OF \cup OJ \cup ST\}$, where SP is the set of Start Points, EP is the set of End Points, R is the set of Responsibilities, AF is the set of AND-Fork, AJ is the set of AND-Join, OF is the set of OR-Fork, OJ is the set of OR-Join, and ST is the set of Stubs. Thus, we can specify now that two scenarios *sc1* and *sc2* are domain element execution dependent if both scenarios share any executed element E', where E' $\subset$ E.

Execution dependency, therefore, can be seen as an extension of feature dependencies allowing for the support of different impact analysis granularity levels. For example, the component execution dependency can be applied for distributed and/or larger systems (both typically modeled in UCM) to comprehend components (subsystems) and how these might be affected by a specific requirements modification. The domain element execution dependency, on the other hand, focuses on a more fine grained analysis at the UCM domain element level and provides insights on how these lower level domain elements might be affected by a modification request. It should be noted, it is similarly possible to assess global variable dependency amongst scenarios which can be defined as scenarios that share the same global variables.

3.3.3 Combining UCM with FCA

In Figure 3-1, we presented a general overview of our UCM_FCA methodology including the different types of modification analysis supported by our approach. In what follows we focus on the process of applying FCA for UCMs.

In our research, we apply FCA to automatically identify different types of dependencies from collected UCM traces based on the object/attribute pairs used as input to FCA (shown in. Table 3-1).

Dependency Analysis Type	FCA-Objects	FCA-Attributes
Functional dependency	UCM traces (Scenarios)	UCM plug-ins (within stubs)
Domain element execution dependency	UCM traces (Scenarios)	UCM domain elements
Component dependency	UCM traces (Scenarios)	UCM components

Table 3-1: The Mapping of Execution Dependencies to FCA Concepts

In our approach, UCM scenarios are executed first to allow the collection of execution traces, and then filtered to generate the FCA format inputs. The FCA algorithm then calculates concepts, and builds up the dependency lattice from the UCM execution traces collected. As shown in Table 3-1, we distinguish three types of dependencies to support our evolution analysis at the UCM level. *Functional Dependenc*ies: To identify the functional dependencies through FCA, we apply the following object/attribute pair as input to the FCA analysis. UCM traces are mapped to objects and UCM plug-ins (mini scenarios) to attribute(s). *Domain element execution dependency:* Similarly, we create for the domain element execution dependency the following object/attribute pair as input to FCA: UCM traces (scenarios) become objects and UCM domain elements are mapped to be attributes for the analysis. *Component dependency:* For the component dependency analysis, we map the UCM traces to the objects and UCM components as attributes.

3.3.3.1 Apply Formal Concept Analysis

As illustrated by figure 3-2, generated /existing Use Case Maps will be validated and

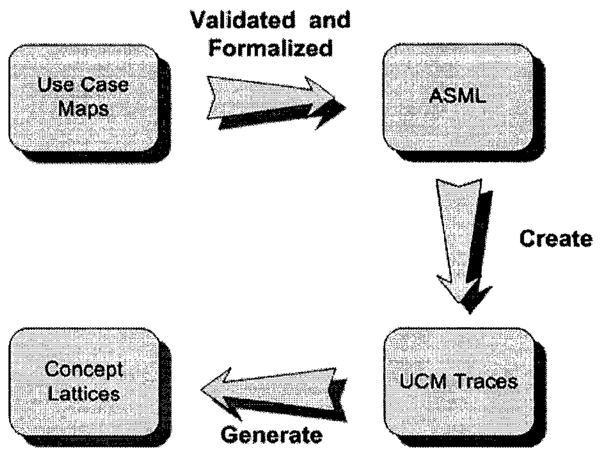


Figure 3-2: The Process of Generating Concept Lattice from UCM Traces

formalized through ASML engine [51] which will output UCM traces (an example is shown in Figure 3-3). These traces will be filtered via a UCM_FCA filtering technique to prepare them in a context format required by FCA algorithm.

This context format is simple and is stored in a text file composed of lines of object names followed by their attributes, similar to what follows:

object1 : attribute1.1 .... attribute 1.n

object2 : attribute2.1 …. attribute 2.n

…..

objectN : attributeN.1…. attribute N.n

```
Start Executing: MainReq
MainReq.active:in1

MainReq:Rule of Start Point:Req
MainReq.active:e1

MainReq:Rule of Stub_Construct:SOrig
Execution of Plugin: Orig_plugin
MainReq.active:Orig_in1

MainReq:Rule of Start Point:Start
MainReq.active:O1

MainReq:Rule of Stub_Construct:Sscreen
Execution of Plugin: DEF_Plugin
MainReq.active:DEF_in1

MainReq:Rule of Start Point:Start
MainReq.active:DEF1


MainReq:Rule of End point:fail
MainReq:UpStub executed
MainReq.active:e5
Responsibility: fwd_sig in component: Agent_Orig
MainReq.active:e6

MainReq:Rule of End point:busy
MainReq:Execution Terminated successfully
```

Figure 3-3: An Example of a UCM Trace

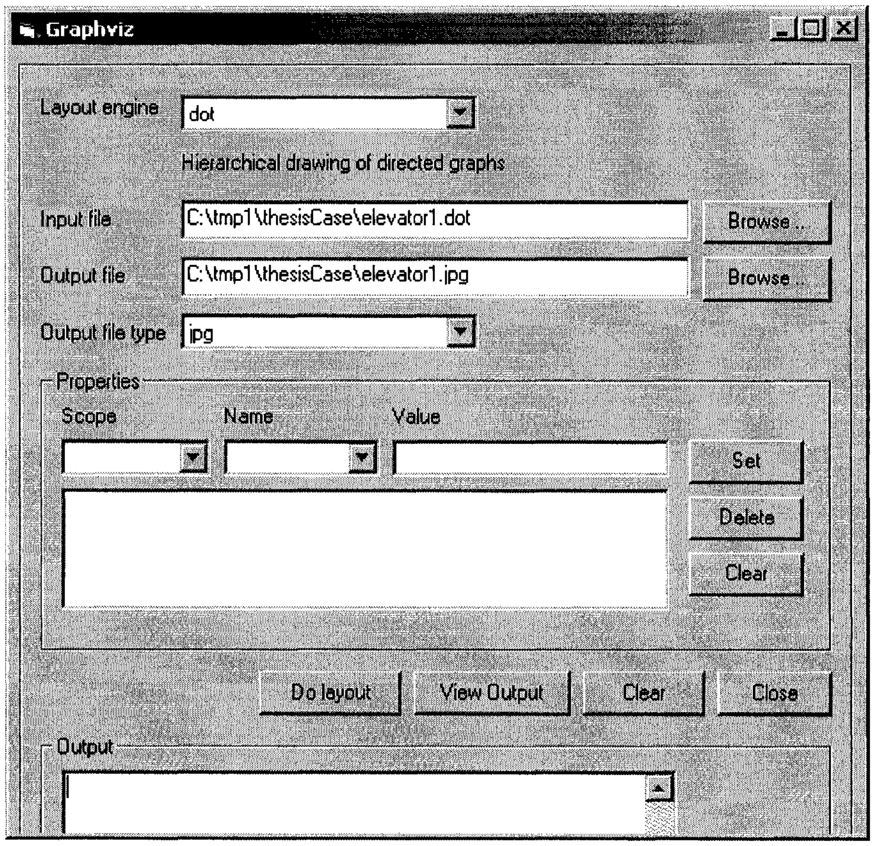
The calculated concepts and their relationships are then listed and an output file is created in a specific format (Graphviz [12], section 3.3.3.2) as shown in Figure 3-4.

```
graph conceptlattice {
label = "concept lattice"
node7[label="\nA",fontsize=10,labelfloat=true];
node6[label="\nH",fontsize=10,labelfloat=true style = dotted];
node5[label="Sc4\nC",fontsize=10,labelfloat=true];
node4[label="Sc1\nB*",fontsize=10,labelfloat=true style = bold ];
node3[label="\nG* D*",fontsize=10,labelfloat=true style = bold color =red];
node2[label="Sc2\nF",fontsize=10,labelfloat=true style = bold color =red];
node1[label="Sc3\n",fontsize=10,labelfloat=true style = bold ];
node0[label="\n",fontsize=10,labelfloat=true style = bold];
node1 -- node0;
node2 -- node0;
node3 -- node1;
node3 -- node2;
node4 -- node0;
node5 -- node4;
node6 -- node1;
node6 -- node4;
node7 -- node3;
node7 -- node5;
node7 -- node6;
}
```

Figure 3-4: A Sample Concept Lattice in Graphviz dot Format

3.3.3.2 FCA Visualization

From the textual representation of the FCA results (Figure 3-4), it is difficult to identify and comprehend the relationships among the various FCA concepts. Providing a visual concept lattice representation can reduce the information complexity and enhance the analysis and comprehension of the results. As part of this research, we performed a brief survey of existing graph visualization software and selected Graphviz [12] as our visualization engine for generating the concept lattice. It was selected mainly because our original FCA engine was easily adaptable to the Graphviz input format. Furthermore, Graphviz, an open source graph visualization software, "takes descriptions of graphs in a simple text language, and make diagrams in several useful formats" [12]. It has several layouts and diagram drawing features (colors, fonts, tabular node layouts, shapes, etc) and has various output formats (such as images, Postscript for inclusion in PDF, etc). Figure 3-5 displays a screenshot of Graphviz program.



Layout engine: dot

Hierarchical drawing of directed graphs

Input file: C:\tmp1\thesisCase\elevator1.dot

Output file: C:\tmp1\thesisCase\elevator1.jpg

Output file type: jpg

Figure 3-5: **A Screenshot of Graphviz Program**

In this thesis, we used the dot layout which can perform the hierarchical drawing of directed FCA graphs.

3.3.4 UCM Change Impact Analysis

Impact analysis focuses on identifying those parts of a system that are (potentially) affected by a modification request. As stated earlier, one of the objectives of our research is to develop an approach that predicts such potential changes without the need to analyze and comprehend the source code of the system to be modified. In our approach we combine UCMs with FCA to perform such a high-level change impact analysis.

In what follows, we present our approach for generating the impact sets related to a modification request at the UCM level. Before presenting our approach, we introduce the assumptions we have made with respect to the change set. As part of our assumptions, we define a change set as all the known elements that will be affected by the planned change. We also assume the Modified Element (ME) is known prior to performing the change impact analysis.

For the impact analysis itself, we apply an approach similar to the one introduced by Tonella in [115]. Tonella et al. identified relationships between execution scenarios and the executed programming entities by means of FCA to determine the impact set. Tonella's approach uses program slicing to generate the decomposition slices for the analysis of FCA and to reduce the complexity of the data set generated from the source code. In our approach we do not use decomposition slices since: (1) while performing impact analysis at the requirements level, we typically deal only with a limited complexity compared to the source code based approach; (2) Tonella had to apply decomposition slices to identify code relevant to a given scenario while in our context we have already dealt with abstractions through the formalization of the UCM elements and the generation of the UCM traces; (3) the UCM traces generated by our system do not provide sufficient fine-grained information (data dependencies) to perform program slicing therefore requiring us to introduce different types of dependency analysis in order to identify the potential UCM elements and to identify scenarios that are either directly or indirectly affected by a modification request.

However, as mentioned in [16,19], by considering both direct and indirect (transitive closure) impacts, the results impact set can become extremely large due to overestimating the result set. Given such a large impact set, its analysis and verification might become impractical. Therefore, in many cases, some sort of measurement (i.e. distance measure [19]) has been suggested in order to reduce the number of false-positive cases and/or to limit the scope of the analysis. Overestimation becomes even more likely at the requirement specification level due to the lack of detailed fine-grained (implementation level) knowledge and the nature of Use Case Maps (isolating and encapsulating of complex details of lower levels of abstraction [10]). Therefore, we limit our dependency analysis by default to one level of transitivity. It should be noted however, that users can extend the analysis by specifying other levels of indirect dependencies to be considered in the analysis.

In the first part of our impact analysis, a user specifies a scenario element to be modified and selects the type of dependency analysis to be performed. In what follows, we illustrate the computation of direct impact set by means of scenarios that share the modified element (ME).

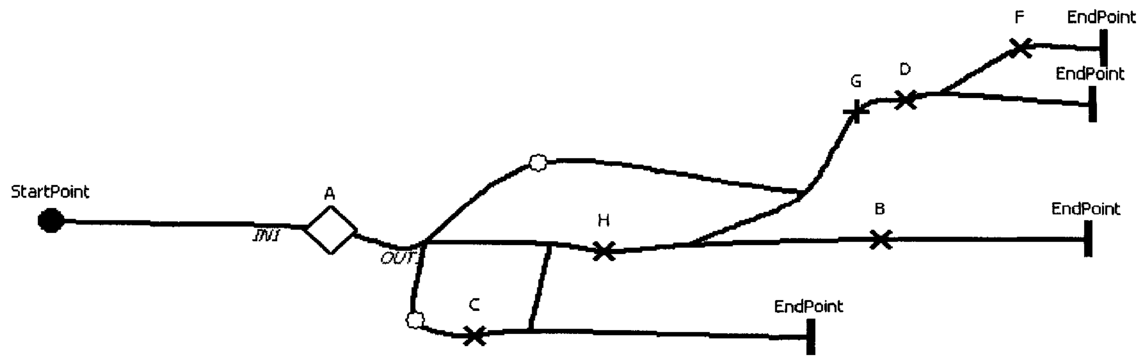


Figure 3-6: A UCM Root Map Example

Based on the UCM example in Figure 3-6, we first create the required traces for the FCA analysis by executing the formalized UCMs in the ASML engine and filtering them into FCA supported format. Figure 3-7 shows UCM trace being generated and used as input to FCA analysis.

```
Sc1: A  C  H  B
Sc2: A  G  D  F
Sc3: A  H  G  D
Sc4: A  C
Sc5: A  G  D
```

Figure 3-7: Simple FCA Context

The FCA algorithm then computes concepts based on their commonality and builds up the dependency lattice. Next the ME is chosen and all scenarios that are potentially affected by this change request are identified automatically.

Given the concept lattice in Figure 3-8 and a change to element H (in this case, a UCM responsibility domain element), the concept containing H will include all these concepts in the change set that can be identified by traversing the lattice downwards, including all reachable nodes (high lighted).

In order to distinguish between objects and attributes, the nodes in the lattice include two parts. The object (scenario) is always on the upper line and the attribute (list of UCM elements) is placed on the lower line.

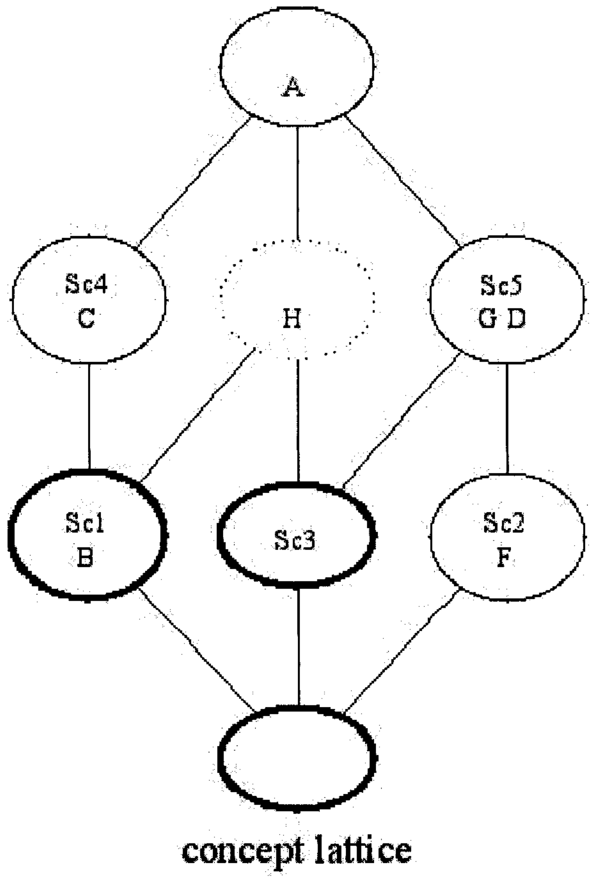


Figure 3-8: Concept Lattice Representation

In order to determine the impact of modification request for element H on the remaining parts of the system, further analysis is required. In the case of H in Figure 3-8, we will know the two scenarios Sc1 with attributes {A,C,H,B} and Sc3 with attributes {A,H,G,D} are potentially affected by the modification. Since FCA provides an analysis based on commonality among objects (in our case scenarios), the FCA lattice will automatically include all scenarios directly reachable in the concept lattice. It has to be noted that FCA will not consider the potential effects of execution sequences on these impact sets, since the FCA analysis is performed on a non-ordered set. Without the information on the execution order of elements, the analysis will therefore consider elements as potentially affected regardless if they are executed prior to the change or not.

This imprecision in the analysis can lead to a large change set of domain elements potential affected by the modification.

We address this limitation of our original FCA-based analysis by integrating information about the order of domain element executions within the FCA lattice. This information is derived by introducing a separate forward filtering step of the UCM traces that identifies all elements modified after the modification point. We apply the following simple forward filtering algorithm to identify these elements executed after the modification point.

```
Input:UCM Scenario traces + (Modified Element ME)
OutPut: Filtered UCM traces, reduced set of potentially impacted elements
Step1: ( /* Searching ME */)
If (ME = component) then
        If (ME found) then Compute all the elements which are bound to the component  and the
        consequent components; Go to step 2
        Else notify user and exit;
        End if
Else /* ME ≠ component */
If (ME found) then Go to Step 2
        Else notify user and exit;
        End if
Step2: /* UCM Forward Traversal*/
 If (ME = component)
 Take all the collected components and their elements and traverse to scenario's end point

If (ME ≠ component)
Simply start from the ME and collect all the elements towards the endpoint.
```

The result of this filtering is shown in Figure 3-9. Elements denoted by a star are those elements that are executed after the modification to element H is performed, and therefore, are also likely to be affected (direct or indirectly). Based on extended analysis, not only Scenario 1 and Scenario3 (containing elements D and G) are identified as being directly affected, but also Sc2 might be indirectly affected. Note that all scenarios which are affected, either directly (bold) or indirectly (dashed) by the modification request, are highlighted in the concept lattice.

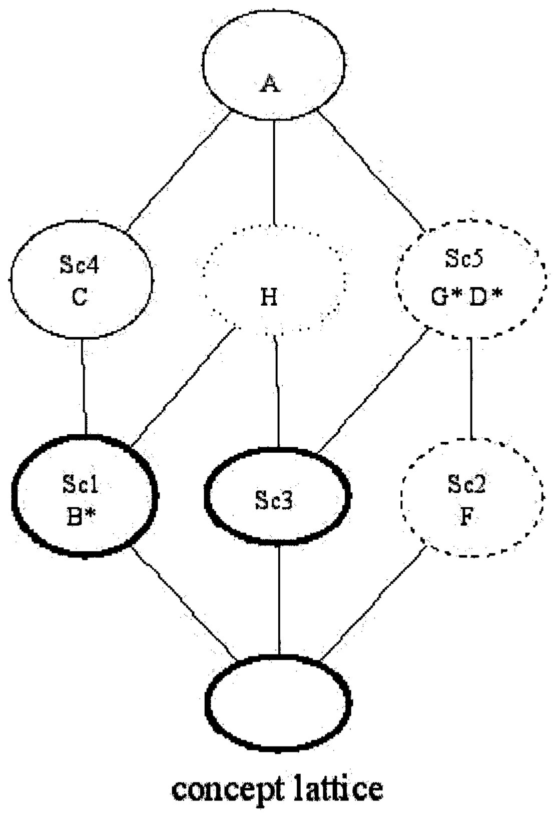


Figure 3-9: Concept Lattice Indicating Direct and Indirect Impacts

3.3.5 UCM Regression Testing

For the selective regression test analysis, a concept (requirement) to be modified is specified. Based on the selected modified requirement, the regression test analysis identifies all test cases that need to be potentially re-tested after the modification is performed. The test cases are identified by traversing the execution dependency lattice downward until all reachable leaf nodes (test cases) that execute the modified component are included. Test cases which are non-leaf nodes and contained in the path between the modified node and its reachable leaves are ignored, since they are already covered by the leaf node test cases.

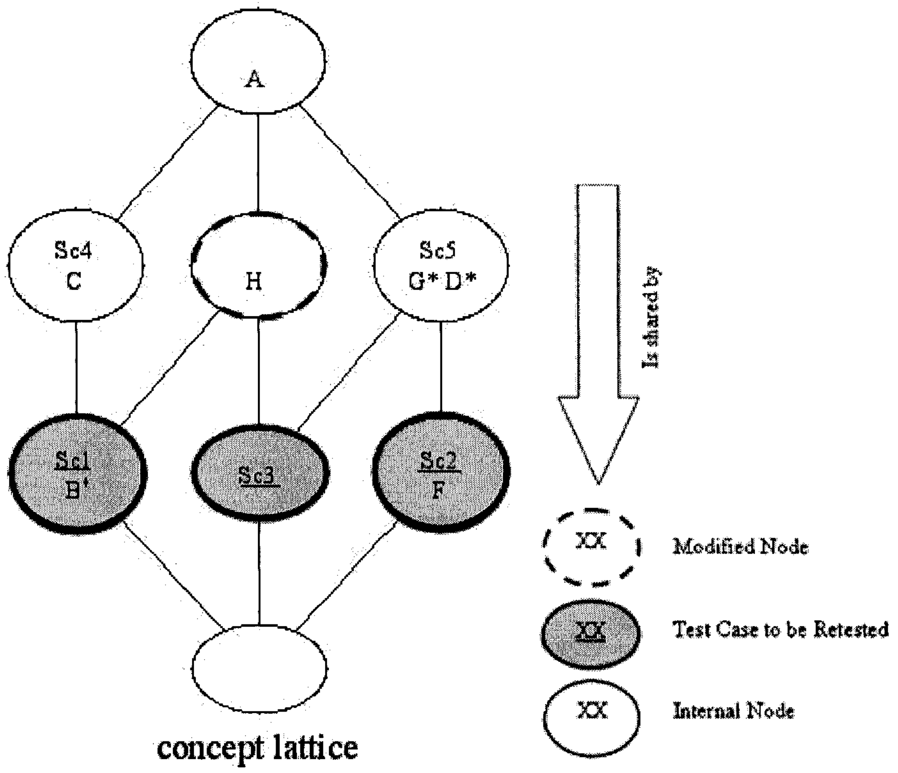


Figure 3-10: Concept Lattice Representing Selected Regression Tests

In what follows, we describe how our method supports selective regression test case analysis at the UCM level. We will also discuss how the selection of the re-usable test cases can be performed. The goal of our test case selection technique is to identify every test case exercising a modified requirement element, thus similar to the coverage techniques for regression testing discussed in section 2.3.2. As stated in the criteria of our hypothesis, we can identify all the test cases that execute a particular UCM path and its UCM elements since each node in our dependency lattice represents a group of UCM elements that are executed by a particular set of test cases (scenarios). The dependency lattice (execution/functional or component) is traversed downwards based on the FCA reading technique for non-redundant-labeling lattice - passing up the objects and passing down the attributes to the concept of interest (see section 2.2). As a result, all the objects that are downward reachable leaf node objects (scenarios/test cases) from the concept being changed are the test cases that execute the modified request.

However, there are cases where a test case can exist in non-leaf nodes located between the modified node and its reachable leaves. If such leaf-level test cases are invoked, all the non-leaf-level test cases included on this path will automatically be invoked. Thus, the non-leaf-level tests can be omitted, and the list of test cases to be re-tested can be reduced to include only the leaf nodes.

3.3.6 UCM Feature Interaction

Scenarios are behavioral definitions of use cases which typically correspond to user requirements. From a maintainer's perspective, it becomes important to identify which scenarios contain which features, and whether a feature interacts with other features. The formalization of UCM allows for the execution and generation of traces to be used in FCA. The quality of the FCA depends directly on the quality and coverage achieved by the traces used for the analysis. For that reason, we assume every scenario at the UCM level was executed at least once.

For the analysis of feature interactions we adopt a three step methodology that analyzes and classifies features based on their interactions. We apply the following classifications for the feature interactions [49, 68]:

- *FI never occurs:* Corresponds to scenarios which contain none or only one feature.
- *FI occurs*: FI occurs will occur, if there exists two feature global variables (sub_F1, and sub_F2) but only one corresponding feature plug-in (plug-in_F1 OR plug-in_F2).This corresponds to a situation where the system selects one of the two possible features.
- *FI can occur (FI prone):* A scenario is *FI prone* when there exist two feature global variables (sub-F1 AND sub_F2), as well as their corresponding plug-ins (plug-in-F1 AND plug_in_F2).

Before introducing the three steps involved in our scenario analysis, we introduce our assumptions. A scenario *Sc1* is a subset of all UCM scenarios SC, where Sc1 $\subset$*{SC}*. We assume that a featured-scenario will contain at least one feature implemented through one or more non-default plug-in(s). For illustration purposes, we also assume that plug-ins are annotated with the name of a feature. For example, feature Call Forwarding Busy line (CFBL, see section 5.3), which is implemented using three plug-ins {Busy CFBL, Busy Setup CFBL and Busy Disconnection CFBL}, is annotated simply as CFBL plug-in since all contain CFBL in their plug-in name. In Figure 3-11, our three-level feature interaction process is presented followed by a description of each step.

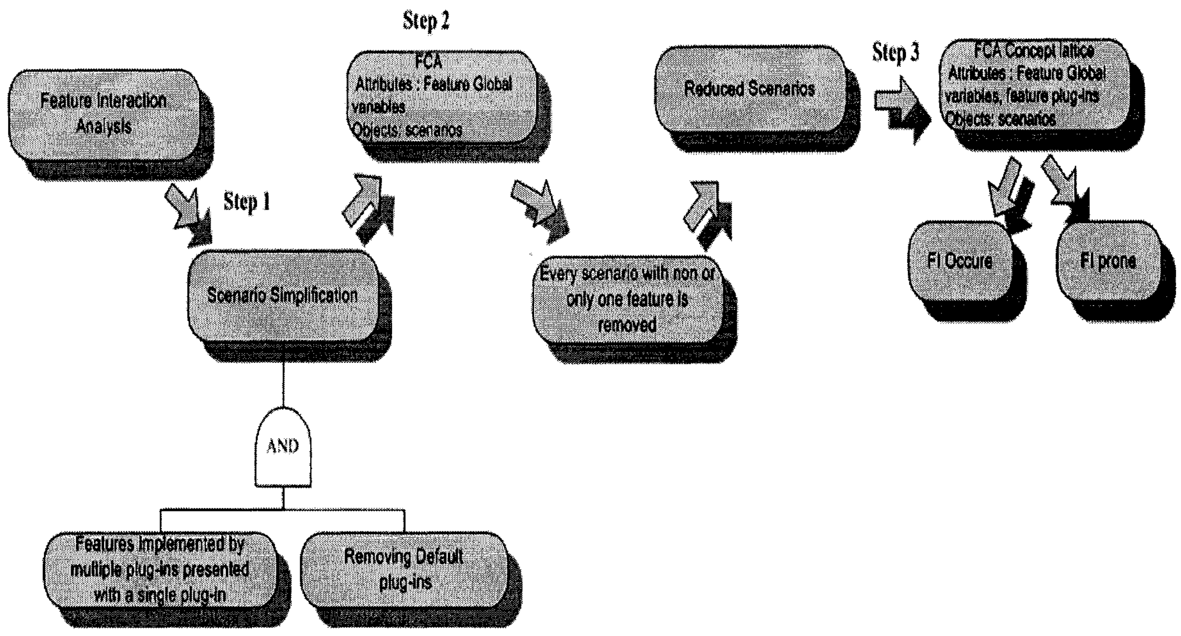

Step 2
FCA
Attributes : Feature Global variables
Objects: scenarios
Reduced Scenarios
Step 3
FCA Concept lattice
Attributes : Feature Global variables, feature plug-ins
Objects: scenarios
Feature Interaction Analysis
Step 1
Scenario Simplification
Every scenario with non or only one feature is removed
FI Occure
FI prone
AND
Features implemented by multiple plug-ins presented with a single plug-in
Removing Default plug-ins

Figure 3-11: Feature Interaction Methodology Process

First Step. Execution traces are generated for all scenarios defined in the UCM. The collected traces form the input to the FCA analysis. It should be noted that in this step the annotations of the plug-ins are simplified and default plug-ins are eliminated.

Second Step. A FCA concept lattice is generated: Global variables of features corresponding to attributes and scenarios become the objects. Based on the resulting FCA, the system can traverse up each scenarios and eliminate those scenarios which contain none or only one feature global variable. These scenarios can be excluded from the next step of the analysis since they are "FI never occurs", thus reducing the number of scenarios that have to be analyzed for FI.

Third Step. The actual remaining scenarios are classified as either FI occur or FI prone by passing down all attributes from the upper levels in the concept lattice associated with the particular scenario.

Once the scenarios are classified as either FI never occurs, FI prone or FI occur the modification request analysis can be performed.

3.3.7 UCM Bad smell detection

In the context of our research we extend the original definition of bad smell at the source code and design level to the UCM specification level. In what follows, we present 5 bad smells based on Fowler's definitions and apply the notion of bad smells at the UCM specification level. The motivation for using these bad smells is to provide analyst/managers with some guidance in identifying potential parts of the Use Case Maps at the requirements level that are not well designed or difficult to evolve over time. From a UCM perspective, UCM language elements (e.g., scenario, responsibility, plug-ins) provide different levels of abstraction that can be analyzed to detect anomalies.

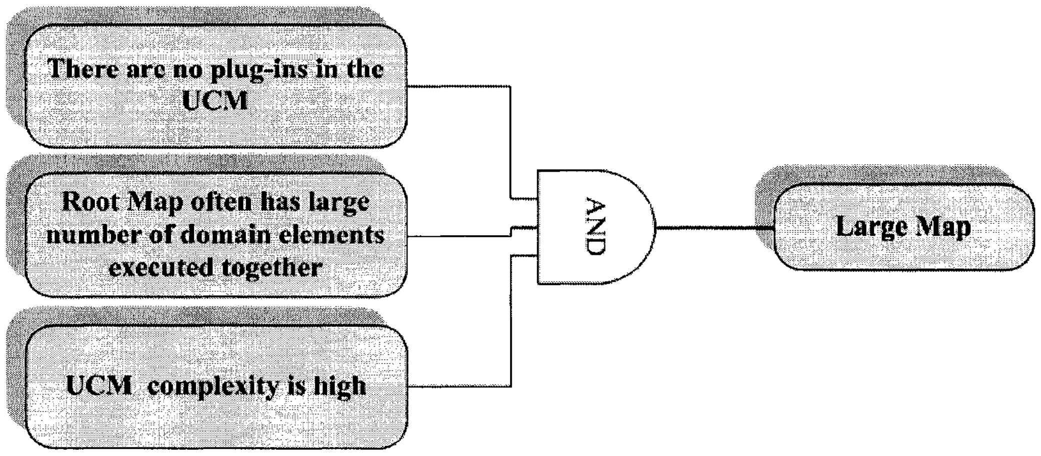
In this section we present several bad smells, and introduce major UCM specification anomalies that can be flagged by our system.

3.3.7.1 UCM Bad smells

1. Bad Smell: Large Map

This bad small is adopted from the idea of *Large Method*, introduced by Fowler [39], and mentioned in section 2.2.1.

Definition: We state that a Use Case Map is identified as a *Large Map* bad smell if a UCM map contains a large number of domain elements and the mechanism of sub-map for implementing features is not applied; thus making the map unnecessarily complex and difficult to comprehend.


There are no plug-ins in the UCM
Root Map often has large number of domain elements executed together
UCM complexity is high
AND
Large Map

Effect: This bad smell results in an increase in system complexity and complicates the comprehension and evolvability of features modeled within such a map.

Example: The following example illustrates how a large map can complicate both the comprehension of the map and the identification of features and their interaction.

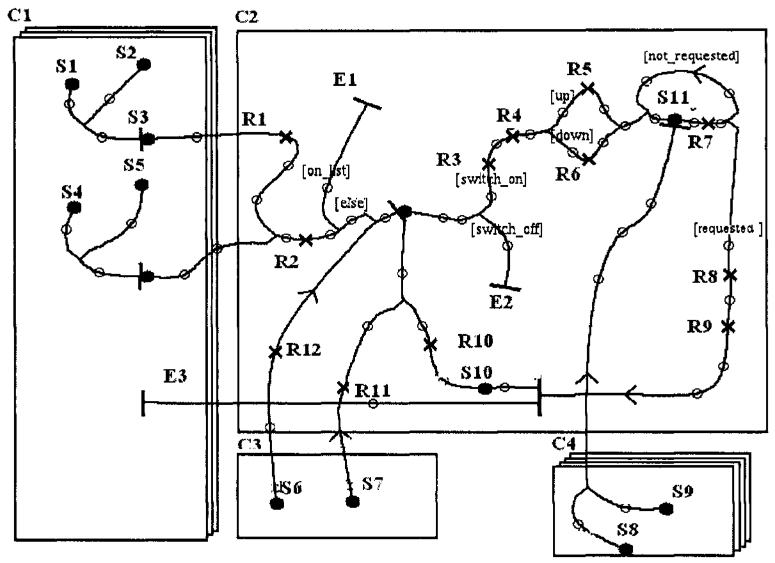


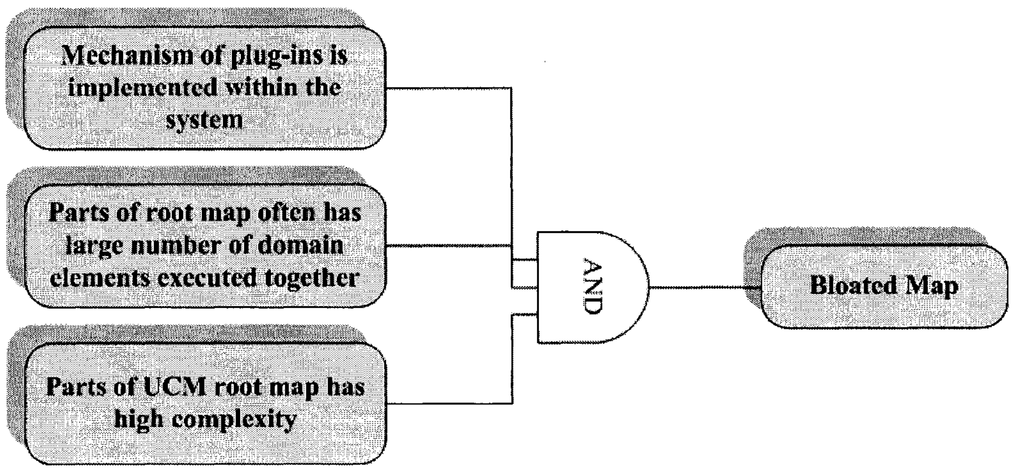
Figure 3-12: Example of a complex UCM with Large Map bad smell

Solution: Possible solutions include the creation of plug-ins to simplify the root-map.

2. Bad Smell: Bloated Map

In an object-oriented program, a Blob represents procedural thinking [24]. Our definition for a *Bloated Map* is very similar to of that of *Large Map*; however, in Bloated Map, plug-ins have been partially implemented, but other parts of the map may still suffer from a lack of plug-ins being used to further reduce the complexity of the map.

Description: A UCM map with some existing sub-maps is regarded as bloated if complex parts (mainly groups of UCM domain elements which are executed together) could be implemented as a plug-in.


Mechanism of plug-ins is implemented within the system
Parts of root map often has large number of domain elements executed together
Parts of UCM root map has high complexity
AND
Bloated Map

Affect: The larger the number of attributes found in a bloated map, the more the re-use and comprehensibility of the map is affected.

Example: In the following example R1, R2, R3, and R5 correspond to four responsibilities that are executed together no matter what condition is chosen. Therefore, to reduce the complexity of the root map, these responsibilities can be implemented in a re-useable plug-in, making the root map simpler and easier to comprehend.

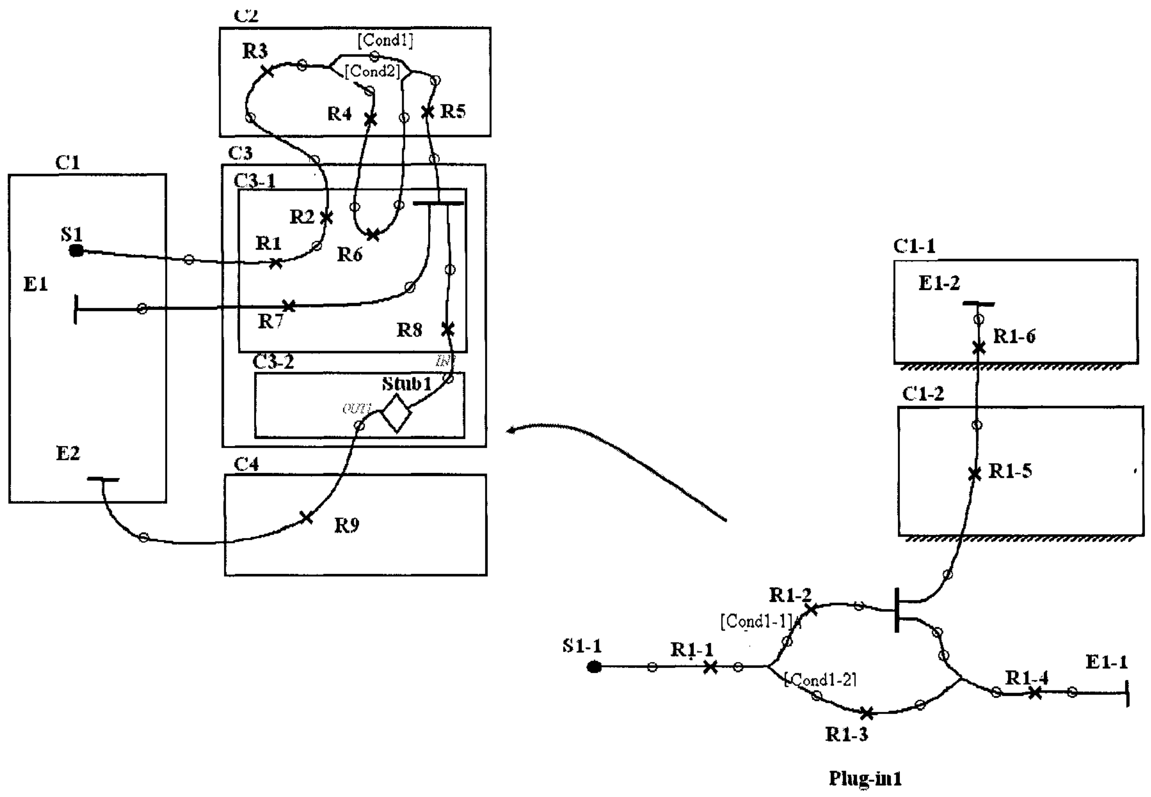


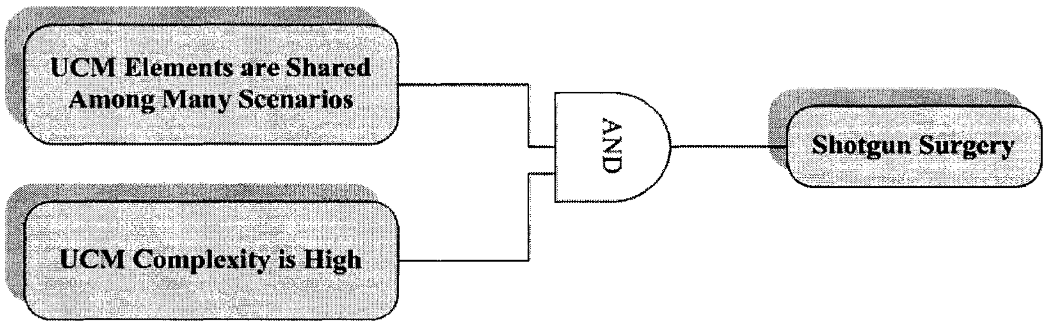
Figure 3-13: Example of a Complex UCM with Bloated Map Bad Smell

Solution: Bloated parts of the system are possible candidates for restructuring by clustering the complex parts (mainly groups of UCM domain elements which are executed together) into plug-ins.

3. Bad Smell: Shotgun Surgery

Shotgun smell is an adaptation of Fowler's [39] *Shotgun Surgery* bad smell, however change smells are not directly visible in the UCM and can typically only be identified by applying FCA on the UCM traces.

Description: When a change in one element of a UCM scenario results in changes in many other scenarios, due to the fact that these scenarios all share this same element, the UCM is considered to have *Shotgun Surgery* bad smell.



Affect: An attribute with *high-change-coupling* bad smell can result in many unwanted ripple effects. In addition to complicating comprehensibility and re-usability of the system, when an attribute is shared by many scenarios, a change in that attribute could potentially affect all of those scenarios.

Example: We can use the example presented in Figure 3-14. Many domain elements within this UCM (i.e. S1, R1, etc) have high-change-coupling because they are shared in most of the UCM system scenarios. However, identifying this type of bad small is a very difficult task in UCMs, especially in more complex systems because of a system's dynamic behavior and the need for execution dependencies to perform the detection.

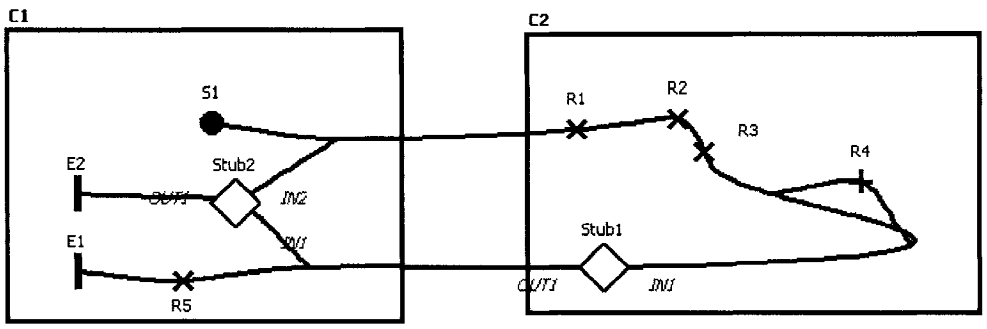


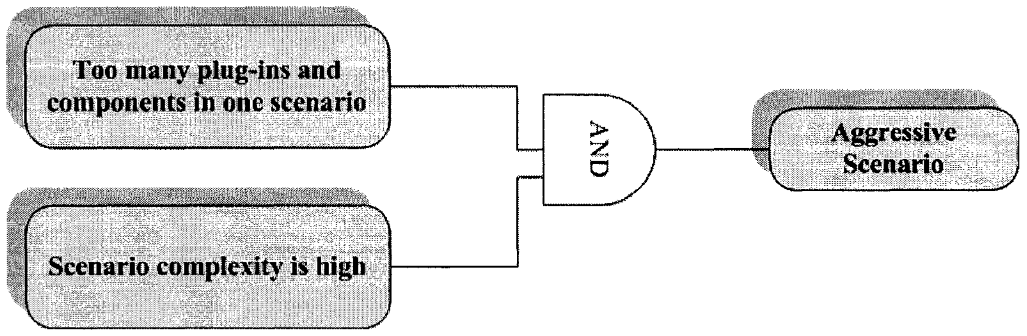
Figure 3-14: Example of a Complex UCM with Shotgun Surgery Bad Smell

Solution: As mentioned in [39], ideally one should arrange the UCM elements so there is a one-to-one relationship between common changes and scenarios. Modifying a shared attribute should take place with extreme caution due to the potential impact on the other elements.

4. Bad Smell: Aggressive Scenario

The forth bad smell introduced at the scenario level is based on the idea of cohesion and high-complexity defects such as *Large Class* bad smell in section 2.2.1. The idea is to adopt "The Swiss Army Knife" design defect smell which can be described as "a complex class that offers a high number of services to address many different needs" [92].

Description: We state that a scenario in a UCM system can be called an aggressive scenario if the scenario is trying to perform a large amount of functionality and involves a large number of plug-ins (features), components (in many cases components represent classes) or both.


Too many plug-ins and components in one scenario
Scenario complexity is high
AND
Aggressive Scenario

Affect: When a scenario contains too many plug-ins or components, keeping track of interactions among these elements is difficult and makes it hard to maintain and modify the system. Also, *Aggressive Scenarios* might be the core scenarios of the system and contain a significant amount of the business rules, and therefore modification to this scenario might impact a large number of other scenarios. The complexity of these scenarios and their interactions with other UCM language constructs will have a negative impact on their comprehensibility and maintainability.

Example: The example below illustrates an Aggressive scenario with plug-ins and components being nested and scenarios containing a combination of these plug-ins and components. Even though these plug-ins are shown in Figure 3-15, determining manually which nested components and plug-ins are used by a particular scenario is difficult. The comprehension problems are due to the potential complexity of scenarios and their dynamic behavior as a result of the use of dynamic plug-ins.

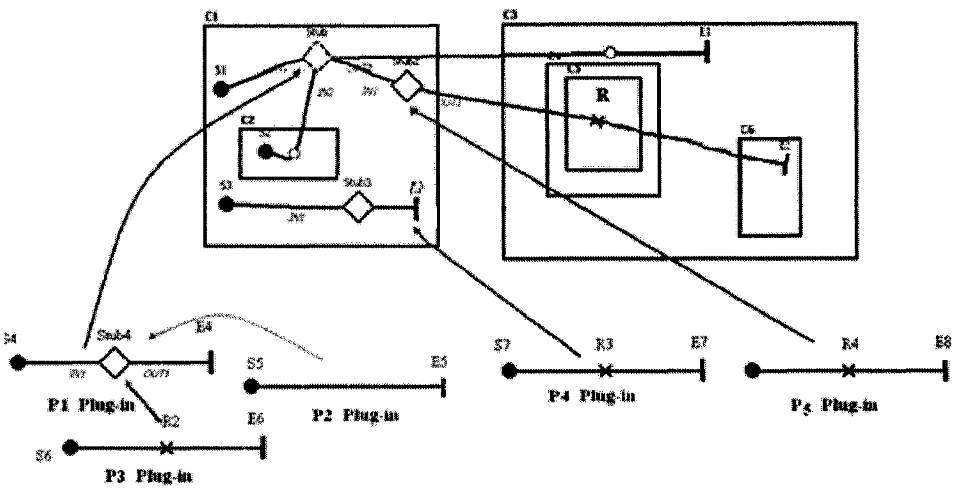


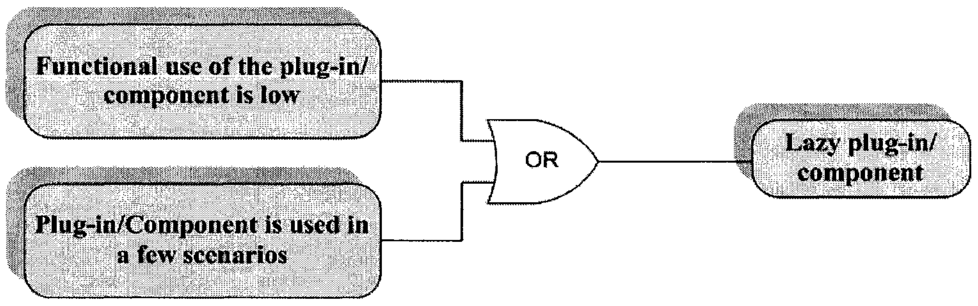
Figure 3-15: Example of a Complex UCM with Aggressive Scenario Bad Smell

Solution: A complex scenario can possibly be composed to smaller more easily handled UCM scenarios.

5. Bad Smell: lazy component /plug-in

The fifth bad smell introduced in this research is adopted from the idea of *Lazy Class* bad smell introduced by Fowler [39] and described in section 2.1.4. This bad smell is related to those components or plug-ins that are providing only minor functionality or services to other scenarios using them.

Definition: Those plug-ins or components which are contained only in very few scenarios may suggest they are not performing much of a system's functionality, and therefore, they are a *Lazy Component/Plug-in.*


Functional use of the plug-in/ component is low
Plug-in/Component is used in a few scenarios
OR
Lazy plug-in/ component

Affect: Each UCM construct will cost effort, money and time to maintain and understand. A not-so-very useful construct will not be worth the cost.

Example: This type of UCM bad smell is also difficult to detect from the UCM itself, and there is a need for other techniques to aid in the detection.

Solution: Lazy UCM constructs are good candidates to be eliminated from a system or to be joined with other existing constructs.

3.3.7.2 UCM Bad Smell Detection

The manual detection of the previously-introduced UCM bad smells is difficult due to the dynamic behavior of scenarios and the execution dependency amongst UCM domain elements. In what follows, we demonstrate our approach, and how by means of FCA these previously defined bad smells can be detected at the UCM level.

1. Large Map

Intuitively, one sign of the Large Map bad smell is its sheer size and complexity. This bad smell occurs when no plug-ins are used to reduce the size of a map. Figure 3-12 (section 3.3.7.1) illustrates such a *large map* bad smell, which can be detected through a visually inspection of the UCM. Applying FCA enables us to go a step beyond just detecting the obvious large map smell; it also allows for the identification of a potential grouping of elements that are always executed together, and therefore, would make a good candidate(s) to be implemented as part of a plug-in. For the analysis we use and perform FCA to generate the concept lattice shown in Figure 3-17. (In the trace represented below, Figure 3-16, Sc represent scenarios, while S is for Start points, R, responsibilities, and E stands for end points).

Sc1: S1 S3 R1 R2 E1

Sc2: S1 S3 R1 R2 E2 R3 R4 R5 S11 R7 R8 R9 S8 S9 S10 R10 R11 R12 S6 S7 E3

Sc3: S2 S3 R1 R2 E2 R3 R4 R5 S11 R7 R8 R9 S8 S9 S10 R10 R11 R12 S6 S7 E3

Sc4: S4 R2 E2 R3 R4 R5 S11 R7 R8 R9 S8 S9 S10 R10 R11 R12 S6 S7 E3

Sc5: S5 R2 E2 R3 R4 R5 S11 R7 R8 R9 S8 S9 S10 R10 R11 R12 S6 S7 E3

Sc6: S4 R2 E2 E1

Figure 3-16: UCM Traces of Large Map Graph

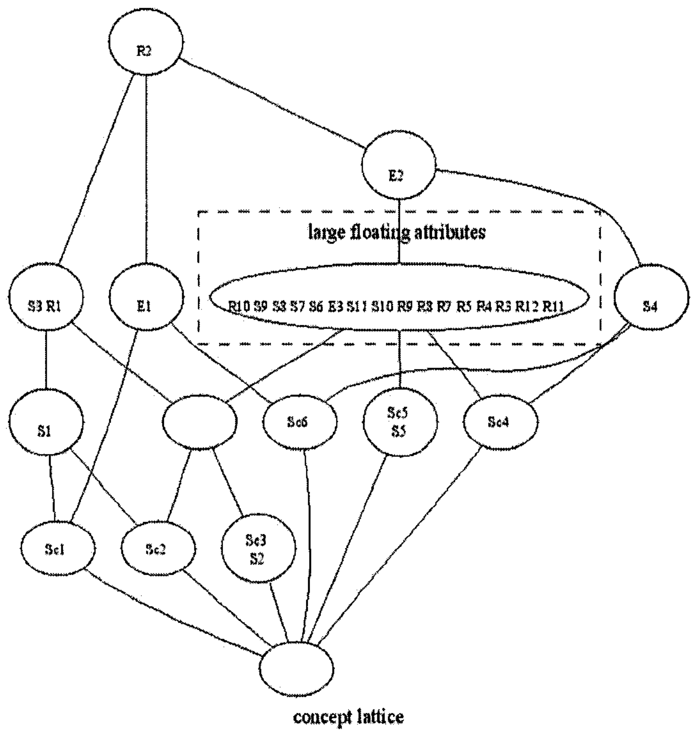

R2
E2
large floating attributes
R10 S9 S8 S7 S6 E3 S11 S10 R9 R8 R7 R5 R4 R3 R12 R11
S3 R1
E1
S4
S1
Sc6
Sc5
S5
Sc4
Sc1
Sc2
Sc3
S2
concept lattice

Figure 3-17: Concept Lattice Corresponding to Large Map Bad Smell

As shown in Figure 3-17, concept #5, framed by the dashed line, suggests that these attributes should be further investigated for the possibility of breaking them into smaller groups of plug-ins. One approach to identify these potential bad smells is introducing a user-defined threshold (depending on application domain, system, etc) based on the ratio between the number of attributes in each concept (NOA) and the total number of attributes (totalNOA) in the lattice. Cases, where the ratio of attributes is larger than the specified threshold, might indicate a potential bad smell.

2. Bloated Map

For the detection of the UCM *Bloated Map* bad smell we re-apply FCA on the UCM traces to identify bloated concepts. For this bad smell, a concept is regarded as bloated if the ratio of attributes, in this case UCM domain elements (NOA) divided by the total number of attributes (totalNOA) in a lattice, is greater than a user-defined threshold.

An example is given in the UCM shown in Figure 3-13 (section 3.3.7.1) consisting of 3 scenarios and their corresponding responsibilities, start/endpoints and stubs. From these scenarios, the FCA lattice shown in Figure 3-19 can be generated. Concepts shown with dashed lines correspond to *bloated* concepts flagged by our tool. These bloated concepts indicate that the number of domain elements associated with the concept exceeds the user defined-threshold and further analysis is suggested. However, it should be noted that the concept containing scenario 2 (Sc2) contains, as a domain element a stub, and therefore, a plug-in has already been applied. However, our tool will still identify it as a candidate of a *bloated map* bad smell.

```
Sc1: S1 R1 R2 R3 R5 R7 R8 St1_R13 R9, E2
Sc2: S1 R1 R2 R3 R5 R7 R8 St1_R12 St1_R16 St1_E12 R9 E2
Sc3: S1 R1, R2 R3 R4 R6 R5 R7 R8 St1_R13 R9 E2
```

Figure 3-18: Bloated Map Bad Smell UCM Traces

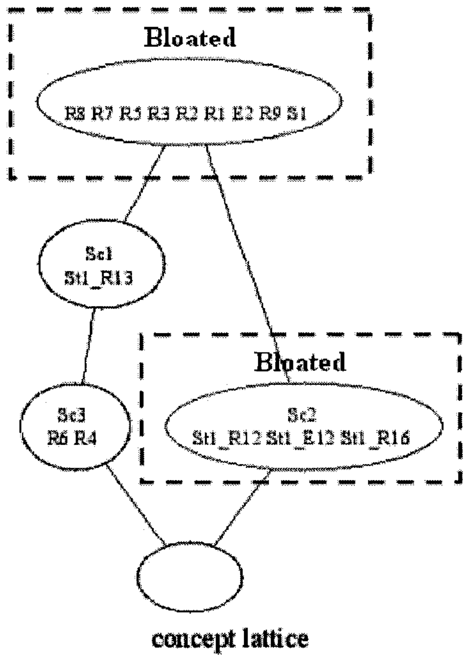


Figure 3-19: Concept Lattice Corresponding to Bloated Map Bad Smell

3. Shotgun Surgery

As a result of using FCA for UCM, attributes located in the lattice supermum (top) are common amongst all the existing scenarios. Therefore, a change in those attributes may result in a change in all other scenarios, thus making them prime candidates for further analysis to improve maintainability of the system. The *Shotgun Surgery* bad smell identifies such modification-prone attributes by computing the ratio between the number of scenarios (NOS) sharing the attributes of a concept divided by the total number of existing scenarios (totalNOS) within the concept lattice. In what follows, we revisit the UCM presented in (Figure 3-14) and study the generated concept lattice to identify attributes with shotgun surgery bad smell.

Sc1:S1 R1 R2 R3 R4 ST1 R5 E1

Sc2:S1 R1 R2 R3 ST1 ST2 E2

Sc3:S1 R1 R2 R3 R4 ST1 ST2 E2

Sc4:S1 R1 R2 R3 ST1 R5 E1

Sc5:S1 ST2 E2

Figure 3-20: Shotgun Surgery Bad Smell UCM Traces

As shown in the Figure 3-20, one can observe that concept 1 and 2 in the lattice are identified as potentially containing domain elements with high change coupling (Shotgun Surgery).

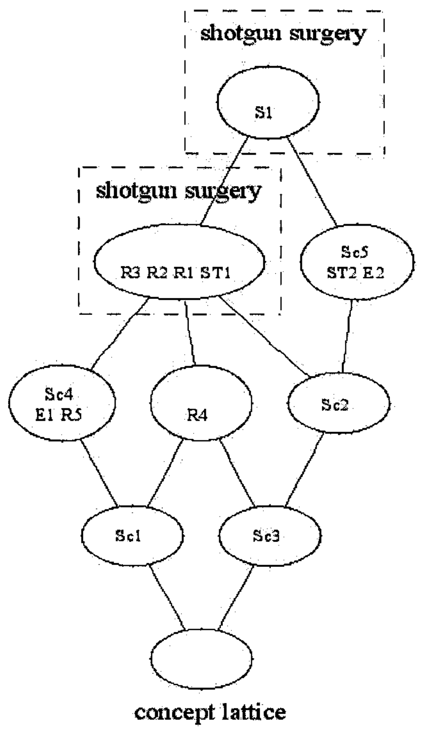


Figure 3-21: Concept Lattice Corresponding to Shotgun Surgery Bad Smell

4. Aggressive Scenario

For the detection of UCM *Aggressive Scenario* bad smells, for each scenario we calculate the ration between number of plug-ins or features (NOP), number of components (NOComp),or both, by dividing them with the (total(NOP + NOComp)) within the concept lattice. If the ratio is greater than a user-defined threshold, our system will flag these scenarios as an *Aggressive Scenario* bad smells.

Given is the UCM example presented in 3-15 consisting of 5 scenarios, 4 plug-ins (nested plug-ins) and 6 components (C1, ...C6).

Sc1:P1 P2 C1 C3

Sc2:P1 P3 C1 C3

Sc3:P1 P3 P5 C1 C2 C3 C4 C5 C6

Sc4:P1 P2 P5 C1 C2 C3 C4 C5 C6

Sc5:P4 C1

Figure 3-22: Aggressive Scenario Bad Smell UCM Traces

From the concept lattice (Figure 3-23), one can identify the scenarios Sc3 and Sc4 have both been flagged by the system (dashed lines) as potentially aggressive scenario candidates, suggesting this complex scenario should be simplified by possibly decomposing them into additional scenarios.

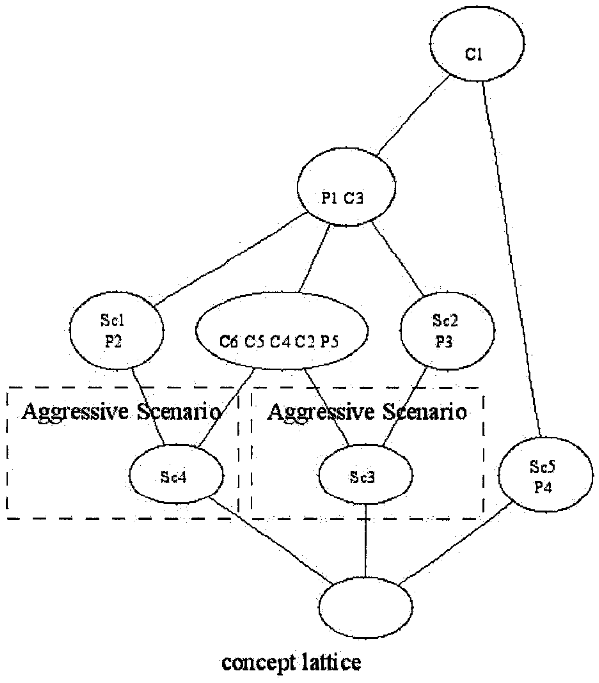


Figure 3-23: Concept Lattice Corresponding to Aggressive Scenario Bad Smell

5. Lazy Component/Plug-in

A plug-in or component can be identified as a *Lazy plug-in/component smell* if the number of scenarios (NOS) that each plug-in or component is used in, divided by the total number of scenarios (totalNOS) in the lattice, is less than a user defined ratio.

Detecting Lazy Component/Plug-in through UCM (i.e., Figure 3-15) can possibly become difficult for larger and more complex UCMs where identifying which plug-ins or components are shared by which scenarios is no longer trivial. For example in Figure 3-24, plug-in 4 (P4) will be identified as a *Lazy plug-in* since it is only used by scenario 5 (Sc5). This suggest further analysis is needed to determine if the plug-in should be kept as is, merged with other plug-in(s), or just simply removed from the system.

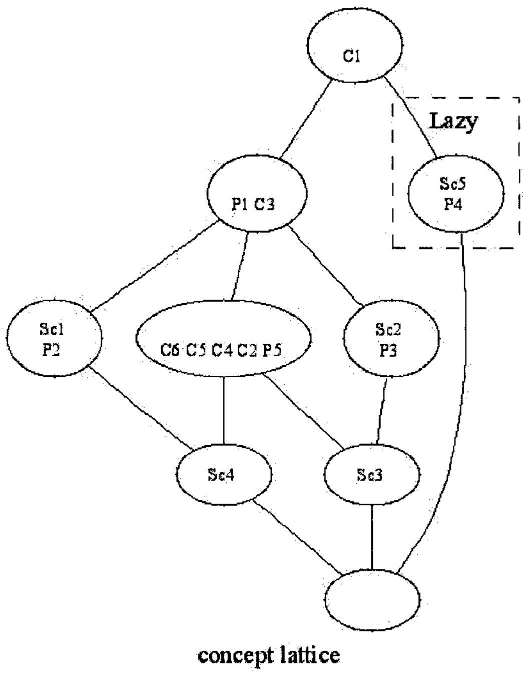


Figure 3-24: Concept Lattice Corresponding to Lazy Plug-in/Component Bad Smell

3.4 Summary

In this chapter, we introduce our research hypothesis, goals, validation criteria and some related work on applicable analysis techniques to support our research objective. We also presented different analysis techniques to support the evolution of requirements at the UCM level. It should be noted the modification analysis in our approach has limitations similar to those of other impact analysis and regression testing approaches. Like many impact analysis techniques, our approach supports mainly modifications of type alternation. Potential impacts and re-testing effort involved in this type of change can be predicted quite well using our UREAF. For modifications involving an addition or deletion of domain elements, an iterative analysis approach can be applied. Modification occurs at the UCM level, then the new set of UCM traces will be executed and the system behavior after re-executing the UCM scenarios can be analyzed and compared.

4. UREAF_Analyzer

This chapter briefly describes some of the architectural and design aspects of our UREAF Analyzer tool developed to demonstrate the applicability of supporting a semi-automatic requirements evolution analysis at the UCM level. The UREAF_Analyzer provides a proof of concept implementation of our requirements evolution methodology. The tool supports: (1) different types of dependency analysis based on a user-specified analysis request; and (2) visualizes the results from the change impact analysis, RTS technique, feature interaction analysis, and bad smell detection.

4.1 Tool Architecture

The UREAF Analyzer is a Java implementation providing an analysis framework that utilizes existing analysis techniques (FCA) with traces from formalized UCM, and visualizes the results using the Graphviz visualization component.

In our system, each component acts as a single processing transformer - accepting inputs and producing outputs which will be used as inputs for the subsequent components - thus implying a pipe and filter architecture for our system (Figure 4-1).

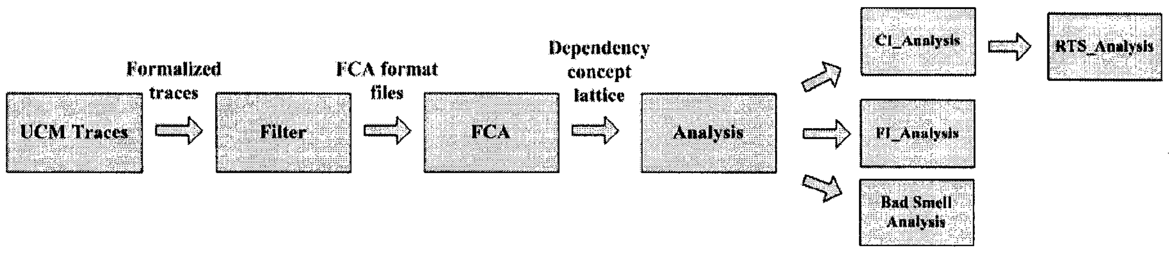


Figure 4-1: UREAF_ System Architecture

Figure 4-2 shows a component model to illustrate the overall architectural of our UREAF Analyzer tool.

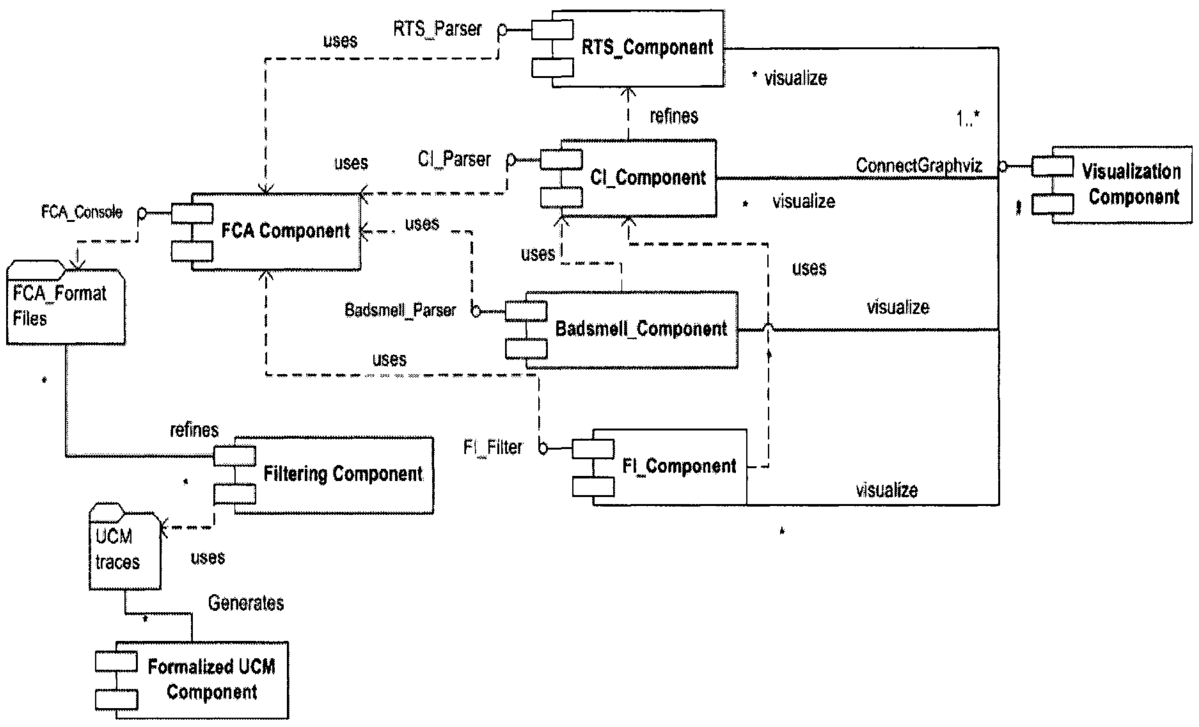


Figure 4-2: UREAF_Tool Component Diagram

The *Filtering Component* transforms the UCM traces, generated by the *Formalized UCM Component,* into an *FCA* .ctx compatible format creating the *FCA_Format Files. The FCA_Format Files* are used for FCA, performed by the *FCA_Component.* The *FCA_Component* is responsible for generating the corresponding concept lattices based on the FCA Format File. Depending on the selected analysis to be performed, the corresponding analysis components will be called and the results of the analysis will be visualized in the form of a concept lattice using the *Visualization Component.* The different analysis components are:

CI_Component determines the impact set of a specific UCM element and identifies the scenarios and their containing elements which may potentially be affected.

RTS_Component determines the list of test cases to be re-tested after the program modification is performed. The system first requires specification of the modified elements (by the user) to be handled through *CI_Component* ,and then, the required analysis will take place through *RTS_Component.*

FI_Component applies several filtering algorithms to identify and remove those scenarios from the set of UCM scenarios that contain none or only one feature, since they are not involved in any feature interaction. In the second step, the remaining scenarios are further analyzed to detect those scenarios which either contain, or may contain, feature interaction.

Bad Smell_Component implements different algorithms to detect five possible types of bad smells within the UCM system by analyzing the result from the FCA.

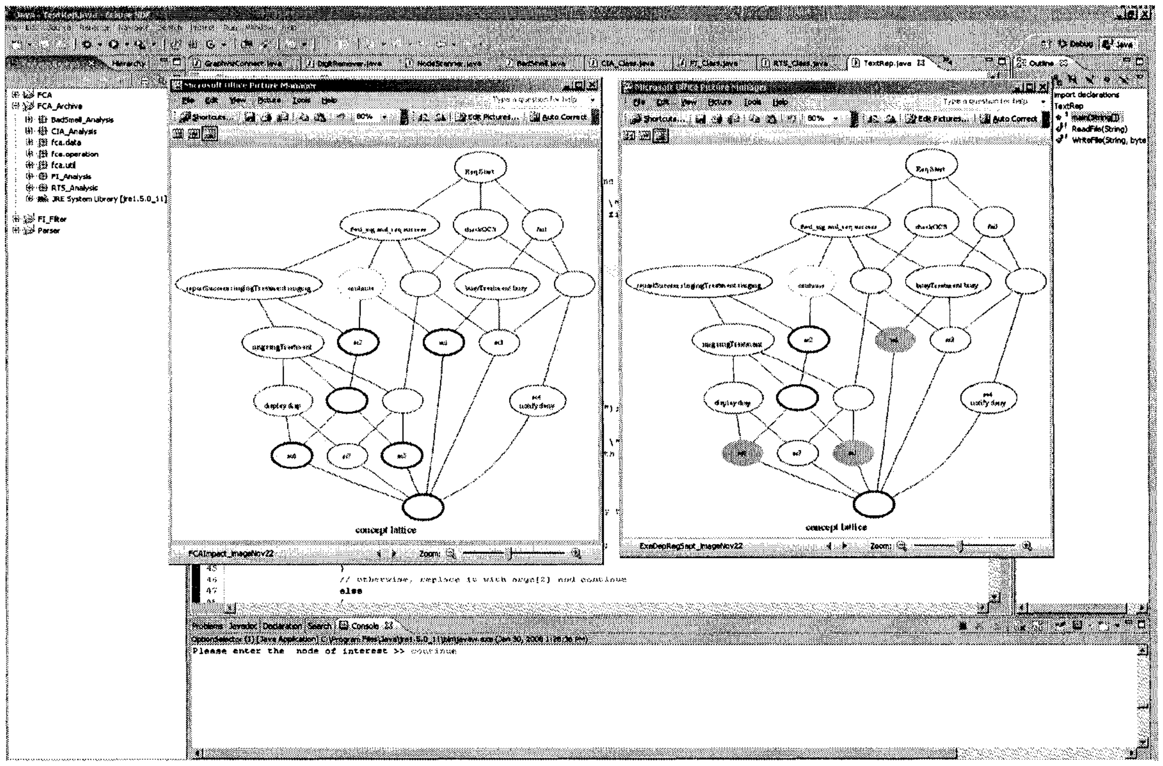

```
        }
        // otherwise, replace it with args[2] and continue
        else
```
```
Please enter the  node of interest >> continue
```

Figure 4-3: A Sample Screenshot of Our Tool showing CIA and RTS Analysis

Visualization component:

In our research we have used Graphviz as an external graph visualization system for our tool to provide a visual concept lattice representation of the results. Once analysis is performed, the output file is generated (depending on the type of analysis) and the Graphviz program will visualize the results. For example in (Figure 4-3), in the case of Change Impact analysis and Regression Test Selection analysis, our tool identifies the modified node as well as the selected test cases' nodes within the execution dependency lattice and uses Graphviz to highlight these nodes in the lattice.

This chapter briefly described the overall architecture of the UREAF tool, by providing a component view and explaining some of the major components and those support evolution analysis at the UCM level.

5. Application Examples

In this chapter, we present several application examples to demonstrate an initial proof of concept for both, our requirements evolution methodology and our tool implementation. In what follows, we apply our approach to existing UCM examples in order to show that it is possible to apply formal concept analysis on UCM to support the analysis of change impact, regression testing selection, feature interaction and, bad smell detection at the UCM requirements specification level. In particular, this brief study will select various examples of the UCM and apply the different aspects of our approach to produce the required information for our requirement evolution analysis framework.

It should be noted that in this set of examples, we are limiting ourselves to direct dependencies (direct impacts) in order to avoid overestimating the results at the requirement specification level. This is significantly due to our lack of data control dependency knowledge at UCM requirement specification level.

5.1 Change Impact Analysis

One of the main challenges in analyzing functional and/or execution dependencies in UCMs results from the use of dynamic stubs and the need to identify inter-scenario dependencies that might exist in a UCM plug-in.

We revisit the telephony case study (Figures 2-1 and 2-4 in section 2.1) to illustrate the applicability of our approach. This telephony system contains 4 functional features: basic call, OCS, CND, and the combination of both OCS_CND. A scenario definition in UCM consists of an identifier, a name, initial values for the global variables, a list of start points, and post-conditions (optional) based on the global variables. In the telephony case study, seven system-level scenario definitions can be identified (Table 5-1). It should be noted that the detailed UCM for Simple Telephony System [81] contains an additional plug-in and some other scenarios which have been eliminated from the model in order to simplify the case study.

Scenario Group	**Number**	**Scenario Name**	**Variables**			
			Busy	**OnOCSList**	**subCND**	**subOCS**
Basic Call	1	BCbusy	T	-	F	F
	2	BCsuccess	F	-	F	F
OCS Feature	3	OCSbusy	T	F	F	T
	4	OCSdenied	F	T	F	T
	5	OCSsuccess	F	F	F	T
CND Feature	6	CNDdisplay	F	-	T	F
OCS_CND	7	OCS_CNDdisplay	F	F	T	T

Table 5-1: Telephony System Scenario Definitions [52]

Feature Dependency

In UCM, a plug-in represents a group of sub-scenarios containing a certain feature. An example for such a feature is the OCS_plugin (depicted with dashed line in the concept Table 5-2). All scenarios containing the OCS plug-in share the OCS functionality and are therefore functionally dependent on the OCS feature. From the context table, one can determine that Scenarios Sc3, Sc4. Sc5, and Sc7 share the OCS-plug-in. Therefore, any change in this feature, or any of its elements, will potentially affect these 4 scenarios.

	Orig_plugin	term_plugin	DEF_plugin	OCS_plugin	CND_plugin
Sc1	x	x	x		
Sc2	x	x	x		
Sc3	x	x		x	
Sc4	x			x	
Sc5	x	x	x	x	
Sc6	x	x	x		x
Sc7	x	x		x	x

Table 5-2: Feature Dependency Context Table

For larger UCMs, the number of objects and attributes increases rapidly and the context table representation will not provide sufficient abstraction to analyze and interpret the dependencies.

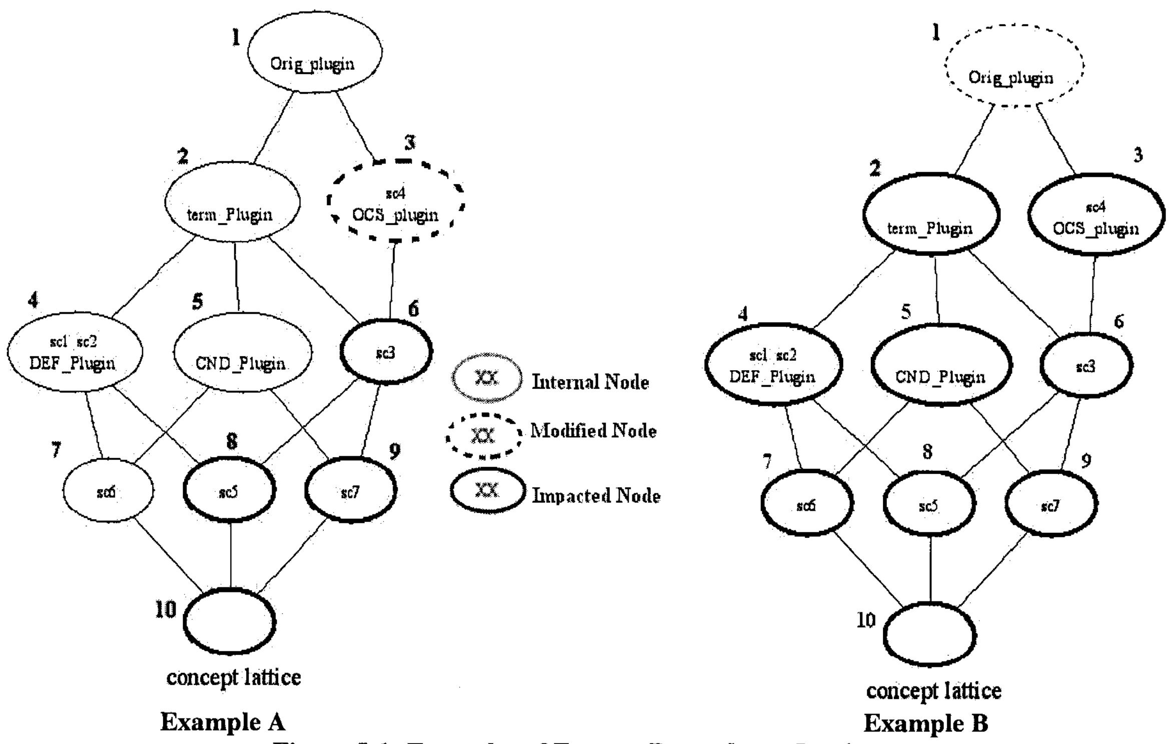


Figure 5-1: Examples of Feature Dependency Lattice

Figure 5-1 shows the resulting graphical representation of the feature dependency in the form of a concept lattice (based on Table 5-2). Objects and attributes in the concept lattice can be distinguished by:

- The object (scenario) is always on the upper line (i.e., Sc4);
- The attribute (or list of attributes) is found in the lower part of each concept node (i.e., OCS_plugin).

In Example A (Figure 5-1), a modification request for the *OCS_plugin* (concept# 3) is analyzed. After selecting the concept node, all objects are passed from bottom levels up to this node. As a result, scenarios sc4, sc3, sc5 and sc7 can be identified as potentially be affected by the *OCS_plugin* modification request.

In Example B (Figure 5-1), a modification request for *Orig_plugin (concept# 1)* occurs. The *Orig_plugin* (the topmost concept) implements the originating call features that are shared by all other features (scenarios) in the telephony system. Traversing the concept lattice, one can identify that in this case every scenario in the telephony system might be potentially affected by the change.

Domain Element Execution dependency

As we have illustrated in the previous examples, FCA can ease the effort involved in identifying potential ripple effects in scenarios. This analysis can be further refined if one considers domain elements (such as start-points, responsibilities, etc) as the unit of change. Changes at the domain element level might occur again as part of incorrect requirement analysis and/or misinterpretation of client requirements.

The concept lattice in Figure 5-2 provides a view of the dependencies between domain elements in the UCM and scenarios.

In this example, we select a concept (concept#7), containing only one attribute "ring" as domain element (responsibility) and pass all of its objects up to this node. In this example, all scenarios (sc2, sc5, sc6, and sc7) sharing this UCM element are included in the change set of scenarios potentially affected (depicted in bold).

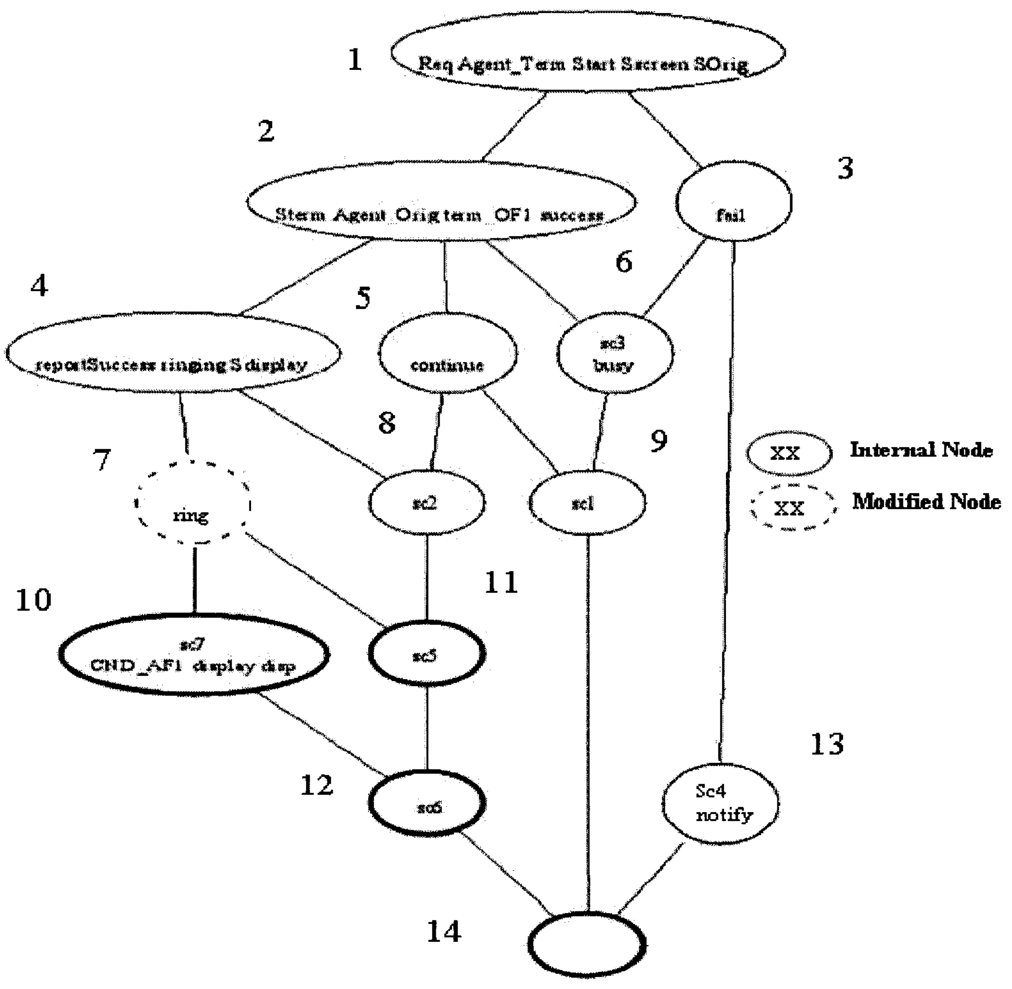


Figure 5-2: CIA_Domain Element Execution Dependency Lattice

A similar analysis can be performed for the other concepts in order to identify their execution dependency between domain elements, concepts and scenarios.

5.2 Selective Prediction Regression Testing

In this example, we re-use the case study from section 5.1 to illustrate our predictive regression testing approach. Given a concept lattice based on the execution of UCM domain elements, we identify the scenarios that have to be re-executed after a modification request is completed.

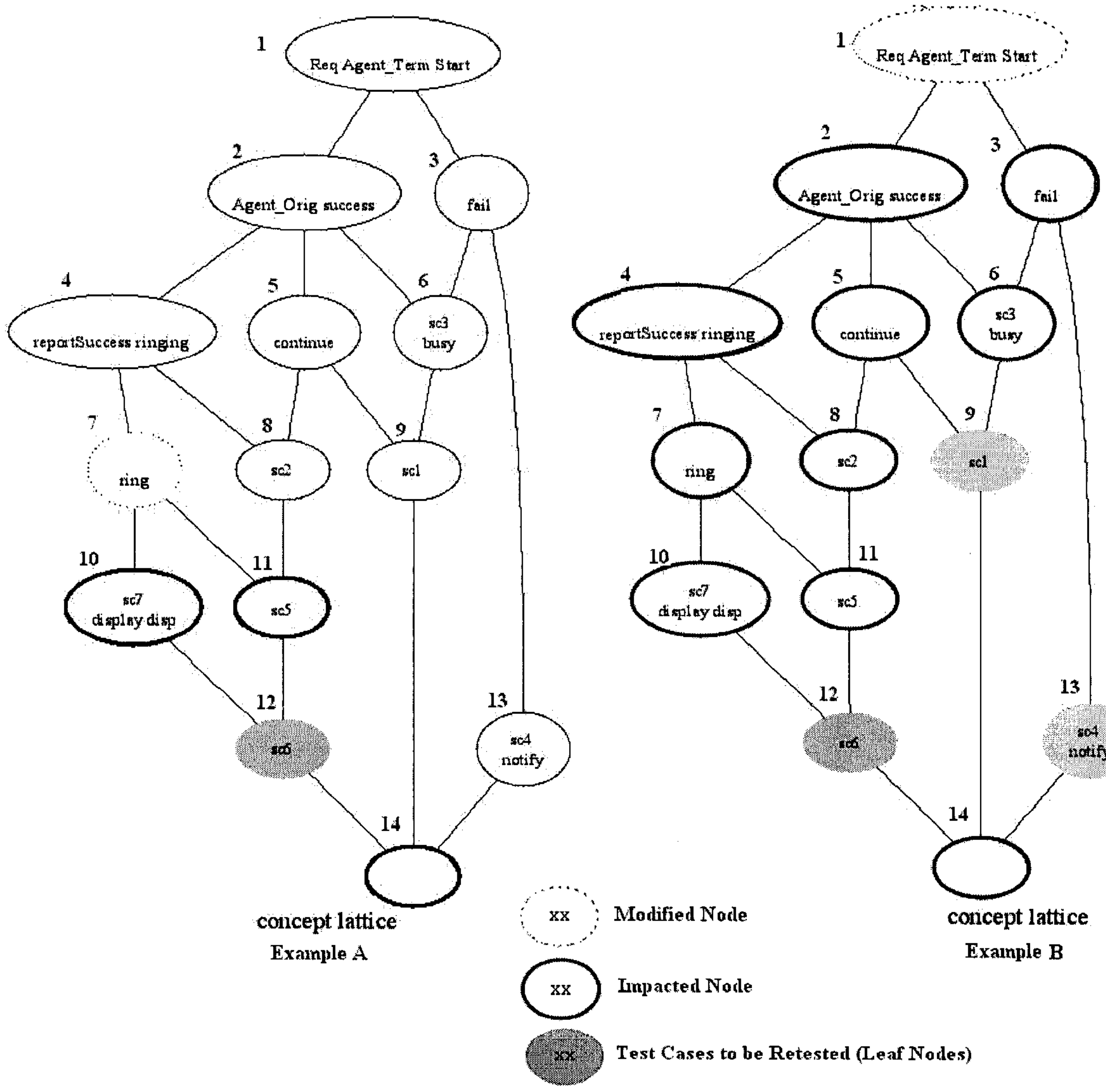


Figure 5-3: RTS_Domain Element Execution Dependency Lattice

Assuming a given modification request (Figure 5-3 – Example A) involving concept #7, this modification will affect potentially *sc6*, *sc7*, and *sc5*. Applying our FCA-based regression test selection approach, one can limit the number of scenarios to be re-tested to *sc6* (at the minimum), since *sc6* also includes *sc5* and *sc7*.

In the second example (Figure 5-3, - Example B), we introduce a change in the most common domain elements (concept #1). Using our selective regression testing approach, one can quickly identify that only sc6 (concept #12), sc1 (concept #9) and sc4 (concept #13) have to be re-tested (leaf nodes). Our initial case study shows that our test case selection technique can be applied at different analysis granularity levels to reduce the number of test cases at the requirements level.

5.3 Feature Interaction

In the next example, we introduce a POTS telephony service using UCM. The POTS system is based on a 4-phase service decomposition [49]. The four service groups provided by the telephony system are: (1) Service Request (2) Information Check (3) Provide the service (4) Disconnection. Figure 5-4 shows the corresponding UCM root map, with stub 1 representing the call request, stub 2 corresponding to the call checking phase, stub 3, 4, 6 and 7 represent the call setup phase and finally stubs 5 and 8 correspond to call disconnection. The basic call model (BCM) describes the core activities involved in establishing a communication between two users, A and B. The basic call can be represented as a model containing the stubs {1: Default, 2: Default, 3: Default, 4: Default, 5: Default, 6: Default, 7: Default, 8: Default}

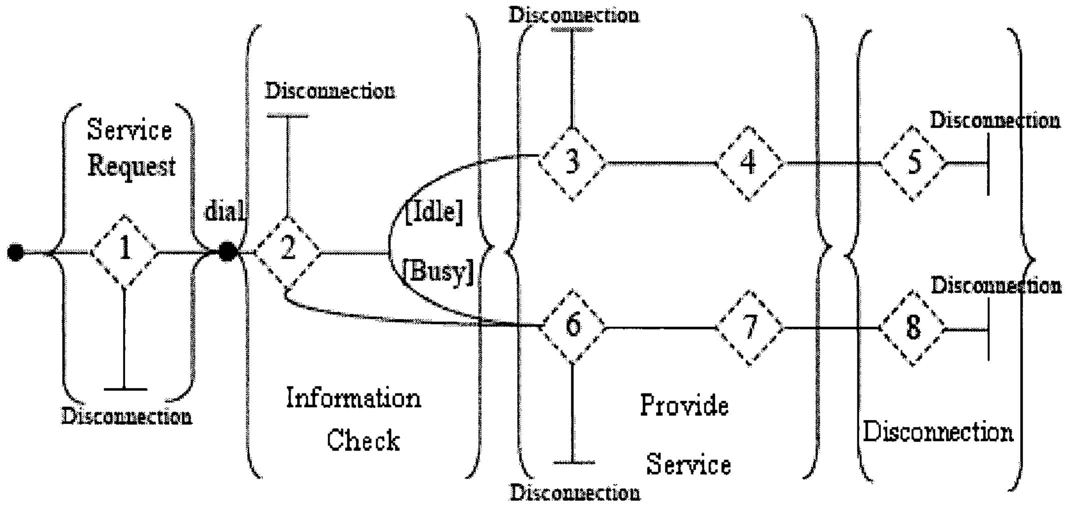


Figure 5-4: UCM Call Model (root map) [49]

Adding Feature to the Basic Call Model (BCM)

From a requirements evolution perspective, modeling features, and their interactions are an important aspect. In UCM, a combination of a root map, stubs and plug-ins can be used to represent system features. The plug-in maps are sub-maps that describe locally how a feature modifies the basic behavior. Adding features to such UCM collections is often achieved by creating new plug-ins for the existing stubs or by adding new stubs containing either new plug-ins or instances of existing plug-ins.

In what follows, we assume that features are an extension/modification of the POTS and stubs are used to define them. As a result, adding features to the system will automatically extend the scenarios in the basic call mode by adding stubs, plug-ins and corresponding global variables (feature global variables) to the system. Consequently, a scenario will only execute a feature if the feature specific plug-in(s), and enabling conditions, are part of this scenario. The following is a list of features added to the original BCM, with each of these features corresponding to a sub-map (plug-in) which either adds a new, or substitutes an existing, stub [49]. For this example, we use a combination of seven features that were originally presented as part of a common contest case study for feature interaction detection [45, 62]. Each feature is described with an end-to-end point of view and the different actions are not bound to network entities.

INTL (IN Teen Line)

The Teen Line feature restricts outgoing calls based on the time of day (i.e., hours when homework should be the primary activity for teens). This feature can be overridden on a per-call basis by anyone with the proper identity code. INTL modifies the BCM by overriding the default plug-in of stub1 with an INTL specific plug-in. As result, the following INTL plug-in configuration is defined: {1: INTL plug-in, 2: Default, 3: Default, 4: Default, 5: Default, 6: Default, 7: Default, 8: Default}

TCS (Terminating Call Screening)

The Terminating Call Screening, (TCS) feature restricts incoming calls. Calls from lines that appear on a screening list are redirected to a vague but polite message. Adding the TCS feature is accomplished by plugging the TCS sub-map into the second default plug-in; therefore creating {1: Default, 2: TCS plug-in, 3: Default, 4: Default, 5: Default, 6: Default, 7: Default, 8: Default}

CND (Calling Number Delivery)

The CND feature allows a called telephone to receive a callee's Directory Number (DN) and the date and time. In the on-hook state, the delivery of this information occurs during the long silence between the first and second power ringing cycles. The CND feature is added to the BCM through a CND sub-map plugging a *Calling Number Delivery* plug-in into the third default plug-in. The new CND feature in the BCM corresponds to the following scenario {1: Default, 2: Default, 3: CDNplug-in, 4: Default, 5: Default, 6: Default, 7: Default, 8: Default}

INFB (IN Free Phone Billing)

The IN Freephone feature allows subscribers to pay for incoming calls. The addition of INFB is done by plugging INFB sub-map "INFB plug-in" into the forth default plug-in, creating the following scenario {1: Default, 2: Default, 3: Default, 4: INFBplug-in, 5: Default, 6: Default, 7: Default, 8: Default}

OCS (Origination Call Screening)

The OCS feature forbids a call to phone numbers on a screening list. The OCS is added to the BCM by plugging in the OCS sub-map into the second default plug-in, generating the following new scenario {1: Default, 2: OCSplug-in, 3: Default, 4: Default, 5: Default, 6: Default, 7: Default, 8: Default}

CFBL (Call Forwarding Busy Line)

This feature enables all calls to the subscribing line to be re-directed to a predetermined number when the line is busy. The subscriber pays any charges for the forwarded call from his station to the new destination. The subscriber's originating service is not affected by this new service.

The addition of CFBL requires adding the CFBL plug-in to the basic root. {1: Default, 2: Default, 3: Default, 4: Default, 5: Default, 6: CFBLplug-in, 7: Default, 8: Default}

VM (Voice Mail)

All calls to the subscribing line are re-directed to a voice mail system. The voice mail feature is added to the BCM by plugging in the VM plug-in into the basic root, leading to the following new scenario: {1: Default, 2: Default, 3: Default, 4: Default, 5: Default, 6: VMplug-in, 7: Default, 8: Default}

In this section we demonstrate how our methodology can support maintainers during the modification analysis through the support of feature interaction analysis. In what follows we revisit our original telecommunication example Figure 5-4 to demonstrate the applicability of our methodology in guiding maintainers during feature interaction analysis and feature modification impact analysis at the requirements level.

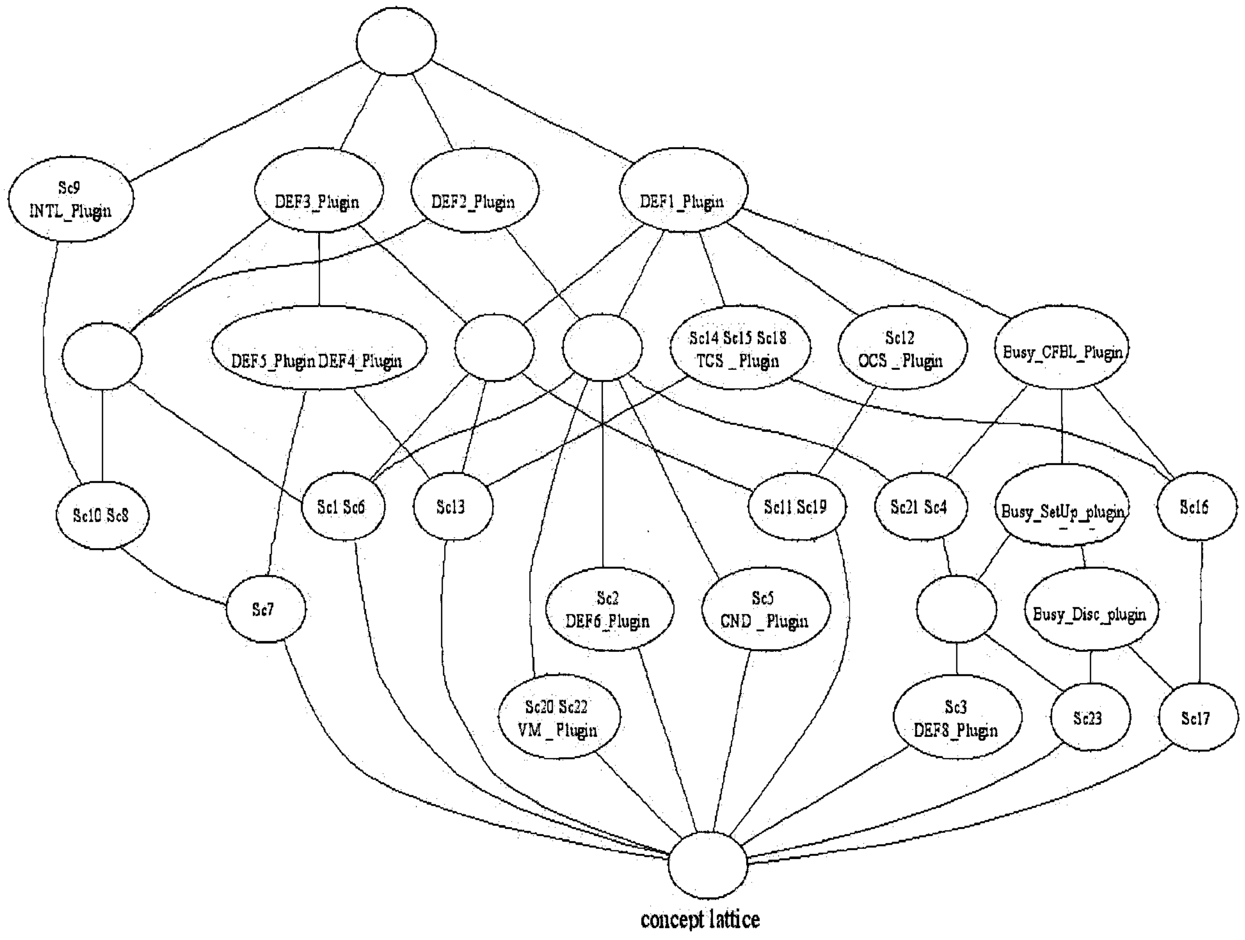


Figure 5-5: Call Model System FCA Lattice

Our feature interaction identification and modification analysis approach starts with the simplification of scenarios (the original concept lattice for the system is shown in Figure 5.5) by applying the following simplification steps:

1) Annotation of features that are implemented by multiple plug-ins and presenting them as a single plug-in. The only feature matching this criterion in our example is the CFBL feature. The CFBL feature itself is implemented through three different plug-ins: {Busy CFBL, Busy Setup CFBL and Busy Disconnection CFBL} and sharing the same sub_CFBL global variables. Also, all default plug-ins are removed from the lattice. Figure 5-6 shows the resulting lattice after the simplification.

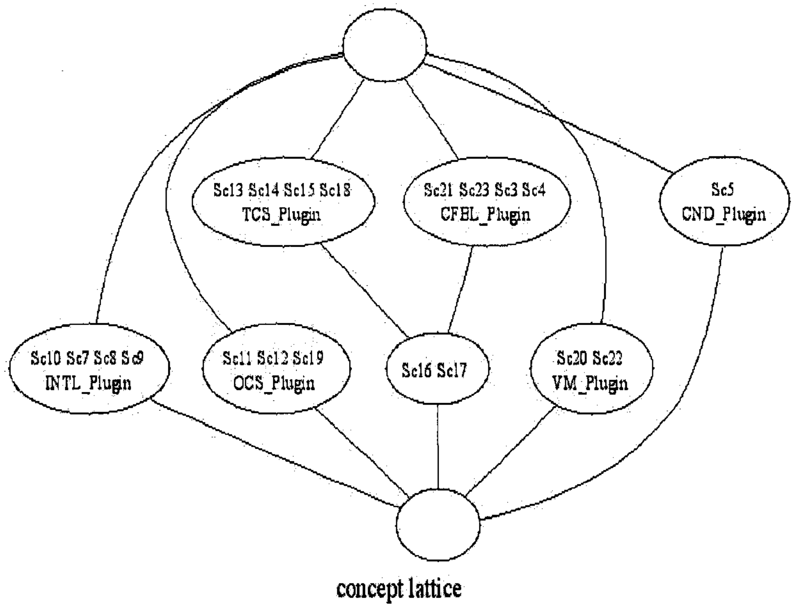


Figure 5-6: FCA lattice of Annotated Plug-ins

2) In this step (shown in Figure 5-7), we perform a new FCA analysis to create a view that is focusing on feature-related global variables (attributes) and scenarios (objects).These global variables represent feature subscription of each scenario (i.e. scenarios 7, 8, 9, and 10 are subscribed to INTL feature because they share subINTL).

More specifically, objects and attributes during this FCA analysis are:

- Object(s) are scenarios (e.g. Sc8, Sc9) shown in the upper part of the concepts
- Attribute (s) are global variables shown in the lower part of a concept

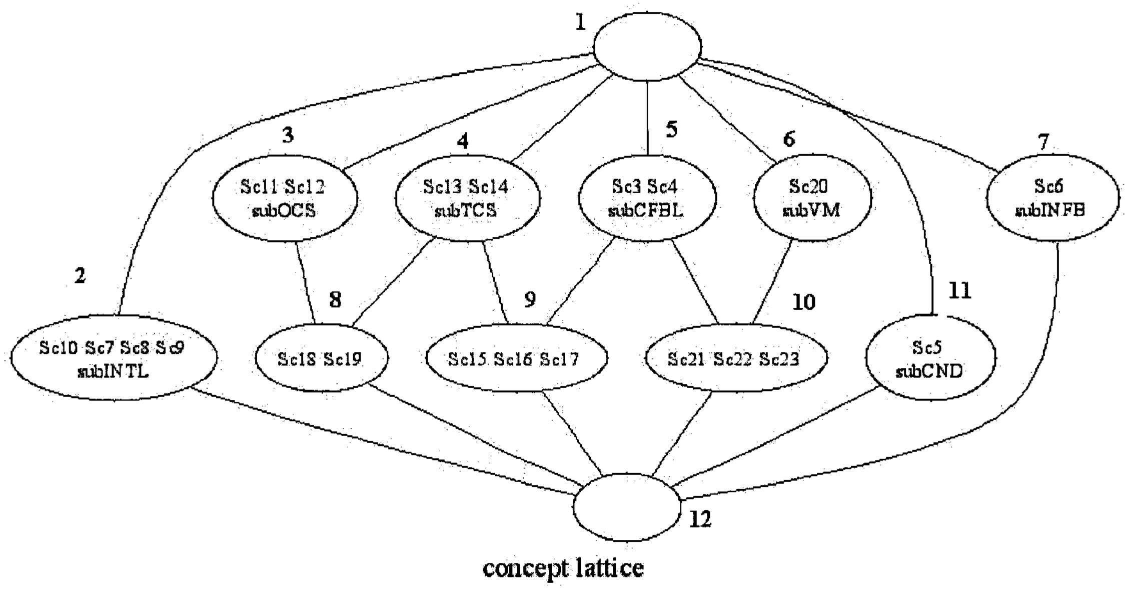


Figure 5-7: Concept Lattice Representing Single Feature Scenarios

Through the feature-related global variable analysis, those scenarios without any feature-related global variables will be eliminated. This is because these scenarios lack attributes to be used for the FCA grouping, and therefore, these scenarios are excluded from the lattice. For the remaining scenarios, a scenario analysis will be performed by our system. For example, while analyzing *Sc6* (concept #7), the system will pass down *Sc6*'s attributes *subINFB* to the current node. Since there is only one attribute for this scenario, Sc6 will be categorized as a scenario involving only one feature. In another example scenarios *Sc18*, or *Sc19* (each will be process separately) are part of concept #8; the two attributes *subOCS* (concept #3) and *subTCS* (concept #4) will be passed down to this node from the upper concepts. As a result, both scenarios have more than one attribute and the system will categorize these scenarios as scenarios that need additional analysis during the third analysis step. This process continues until every scenario has been analyzed and those scenarios with none or only one feature will be removed. Therefore in our case study, only 8 of the original 23 scenarios (Figure 5.8) will be further analyzed.

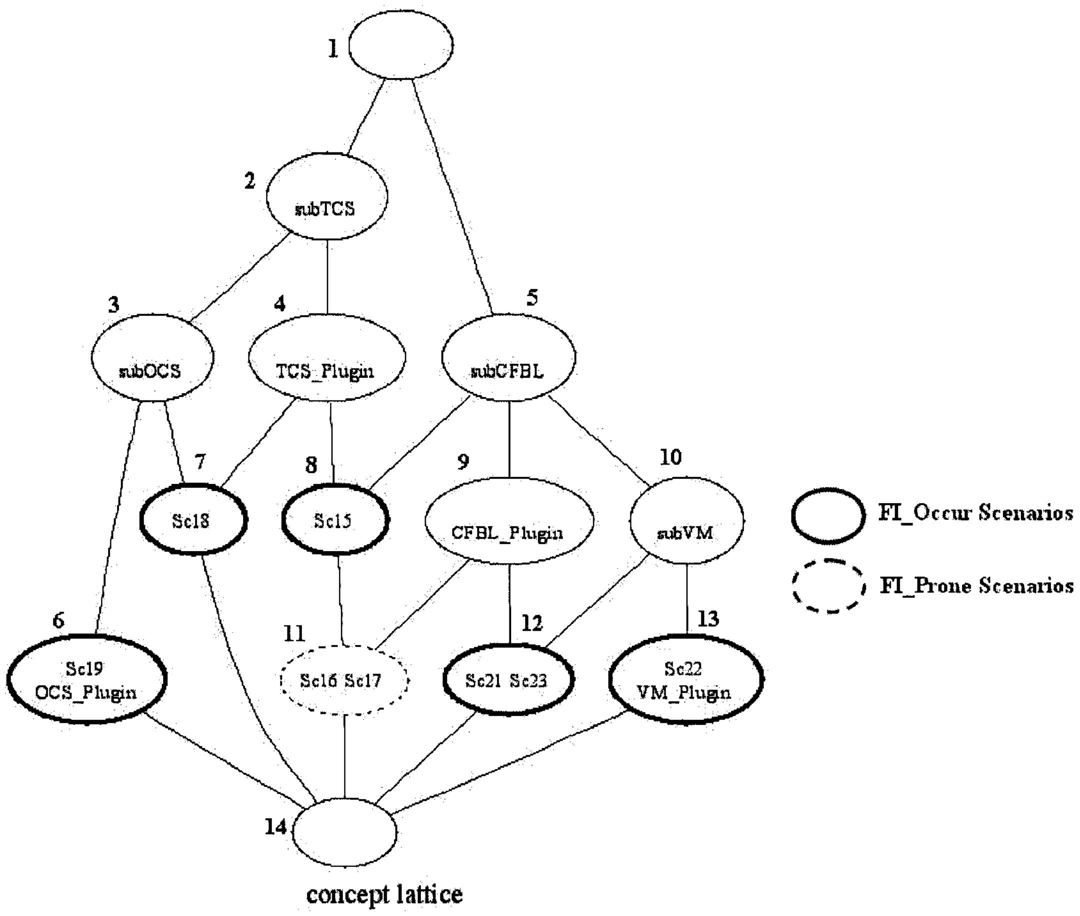


Figure 5-8: Feature Interaction Concept Lattice

3) In the last analysis step, the actual feature interaction analysis will take place. The analysis is based on the reduced concept lattice derived from the first two analysis steps. Figure 5-8 shows the feature interaction concept lattice with scenarios being its objects and feature-related global variables and plug-ins being its attributes.

For the feature interaction analysis, a scenario is selected, for example. *Sc19* (concept #6), and all its attributes will be automatically passed down {*subTCS, subOCS*, and *OCS_plug-in*}. Based on the global variables, the system maintainer can now easily identify two global subscription variables subTCS and subOCS, but only one corresponding plug-in (OCS_plug-in). The fact that the *TCS_plug-in* is missing is caused by non-determinism between the *TCS* and *OCS* feature. Based on our feature interaction definition, such a non-determinism instance corresponds to an FI-occur feature interaction between the *TCS* and the *OCS* feature. The FI occurred is indicated in the concept lattice by the concepts in bold. Another example for the feature analysis is *Sc16* (concept#11), in which again all attributes are passed down {*subTCS, TCS_plug-in, subCFBL,* and *CFBL_plug-in*}to concept #11. Through the FI analysis, we can now identify that two features, *TCS* and *CFBL,* and their associate plug-ins are part of *Sc16.* Based on our FI definitions, scenario *Sc16* is categorized as FI prone (shown as dashed lines).

The FI analysis results for our telecommunication case study shows that from the 8 remaining scenarios, only two scenarios *Sc16* and *Sc17,* are FI prone in the telecommunication system. The remaining 6 scenarios {Sc19, Sc18, Sc15, Sc21, Sc22, Sc23} are all FI occur, meaning that feature interaction in these scenarios exist. The remaining 15 scenarios eliminated during the initial two analysis steps are all FI never occurs (table 5-3).

	Original	**Non FI**	**FI occur**	**FI prone**
Scenarios	23	15	6	2
	100%	65%	26%	9%

Table 5-3: Feature Interaction Analysis

5.4 Bad Smell Detection

As discussed throughout, UCMs are able to show both the relationship among use case, as well as the progression of their scenarios in a map-like approach. However, often these

UCM diagrams contain unnecessary complexity due to modeling problems and these maps end up complex and unnecessarily difficult to comprehend. Therefore, detecting such modeling problems through bad smells in the UCM system can assist requirements engineers in identifying the parts of the system requirements that could potentially be restructured to improve the comprehensibility and maintainability of the UCM models.

In this section, we illustrate how our UREAF Analyzer tool can be applied to detect potential bad smells in UCM system.

5.4.1 Large Map Bad Smell

The following example models a UCM for an elevator system containing 6 scenarios (shown in the Table 5-4) [117].

Scenario Group	**Number**	**Scenario Name**	**Variables**				
			switchOn	OnList	UP	Req1	Req2
atFloor	Sc1	On_list	-	T	-	-	-
atFloor	Sc2	Up_Two_Requested floors	-	F	T	F	T
atFloor	Sc3	Down_to requested_floor	-	F	F	F	T
inElevator	Sc4	Up_Two_Requested floors	T	F	T	F	T
inElevator	Sc5	Down_to requested_floor	T	F	F	T	T
inElevator	Sc6	On_list	-	T	T	-	-

Table 5-4: Elevator System Scenario Definitions

System description: In the elevator system, an elevator is called by pressing a call button, either at a floor or inside the elevator. A floor number can be added to the list if the switch is on (otherwise the system will shutdown). The doors will be closed and depending on whether the up or down buttons were pushed, direction will be decided and the elevator will move. It keeps moving and passing the floors that were not requested when approaching each floor until the elevator passes by floor F. The elevator will stop at floor F, and when stopped, it will open the doors. When the elevator doors have been opened, they will automatically close after a delay and the floor number F will be deleted from the list (Figure 5-10).

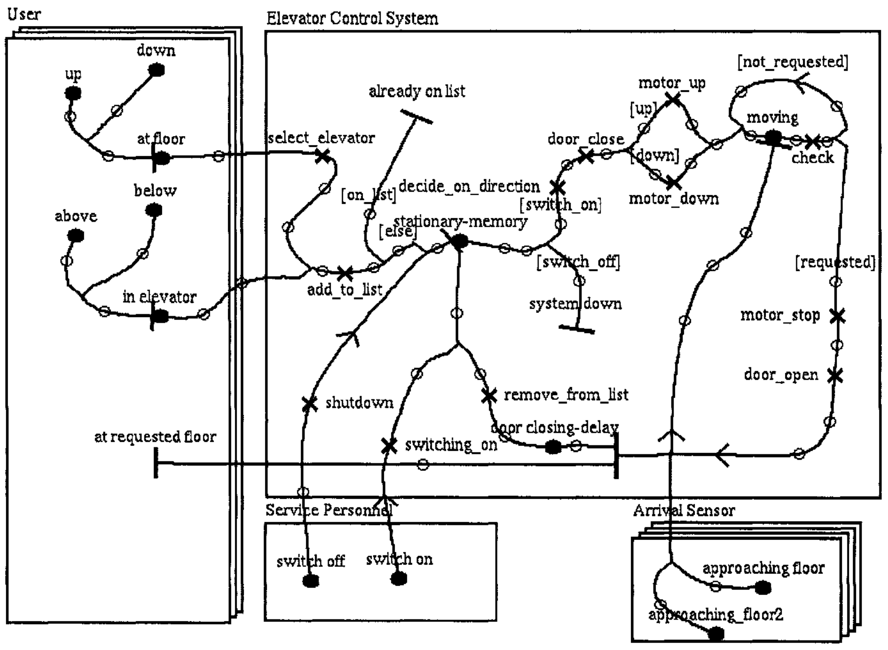


Figure 5-9: Elevator system UCM [117]

As shown in Figure 5-10 (concept lattice), the resulting UCM contains only one complex root map and no sub-maps, making this UCM quite difficult to comprehend. One approach to improve the UCM is to identify UCM domain elements that can potentially be logically grouped together and then move them into a plug-in. In order to create such a grouping, we apply FCA to identify the domain elements that are executed together and are common within scenarios. Using scenarios as objects and the UCM domain elements as attributes as input to the FCA analysis, we can generate the following concept lattice (Figure 5-10). In order to identify a large map bad smell, users will have to define a user-specific threshold (see section 3.3.7.2) to be used by our bad small detection algorithm to identify such large map bad smells. In this example, a default threshold value of 25% was adopted, indicating that a concept contains more than 25% of the total number of concept attributes in a system as too large. As shown in Figure 5-10, the concept identified by our tool as a *Large Map* contains 57% of the total of attributes. This indicates these parts in the concept might be good candidates to be moved into a sub-map.

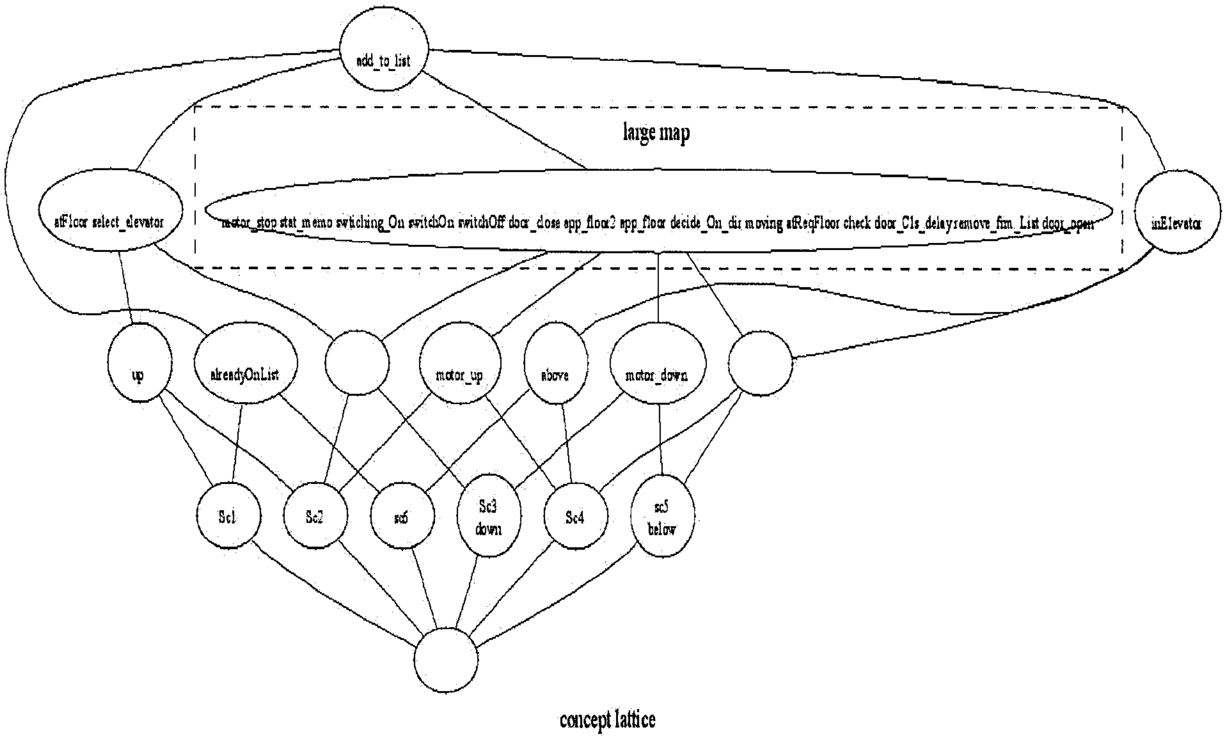

add_to_list
large map
atFloor select_elevator
motor_stop stat_memo swtiching_On switchOn switchOff door_close app_floor2 app_floor decide_On_dir moving atReqFloor check door_Cls_delayremove_fm_List door_open
inElevator
up
alreadyOnList
motor_up
above
motor_down
Sc1
Sc2
sc6
Sc3
down
Sc4
sc5
below
concept lattice

Figure 5-10: Concept Lattice for *Large Map* Bad Smell

5.4.2 Bloated Maps

The following example, taken from [117], describes a pizza delivery system. The corresponding UCM for the pizza system is shown in Figure 5-11). The UCM includes 3 scenarios shown in the Table 5-5.

Scenario Group	Number	Scenario Name	Variables	
			AllIngredients Avail	CreditCard Stolen
Primary Scenarios	Sc1	Normal Case	T	F
Exceptional Scenarios atFloor	Sc2	Missing Ingredients	F	F
	Sc3	CreditCardStolen	T	T

Table 5-5: Pizza System Scenario Definition

System description: When ordering Pizzas, the system checks the customer credit cards. If the credit is approved, the order will be confirmed, the pizza will be made and delivered. If the credit card has been reported as stolen, the bank will request the customer's address from the receptionist. To avoid the customer becoming suspicious and knowing the fraud has been discovered, the bank will pay for the pizza and the pizza will be delivered.

The multi-step process of making a pizza is presented in the MakePizza plug-in with two possible scenarios. In the first scenario, the pizza order is received and all the ingredients are available which results in a pizza ready to be delivered. For the second scenario, some of the requested ingredients are missing and the pizzeria will reimburse client for the delay by refunding some money to the customer's credit card.

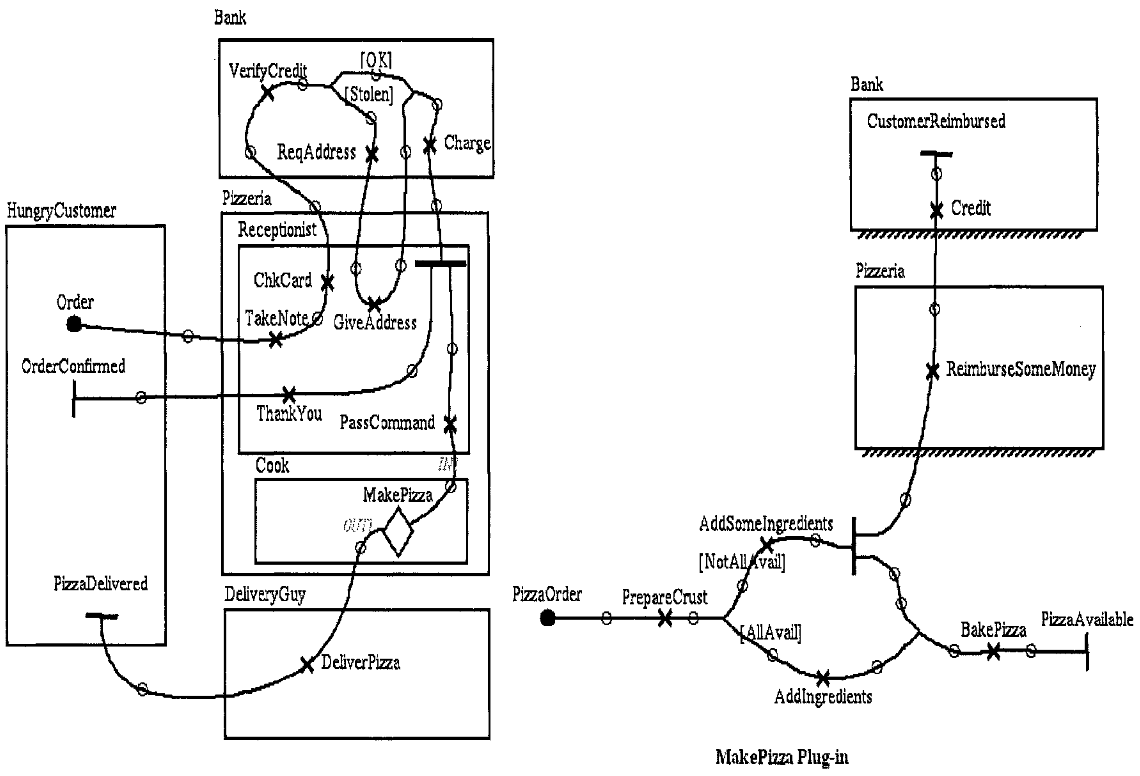


Figure 5-11: Pizzeria UCM [117]

As shown in the UCM diagram in Figure 5-11, the map contains a sub-map to cluster some of the scenarios. However, there are still parts of the map that are unnecessarily complex and difficult to understand. Using FCA on the scenario traces, our tool can identify UCM domain elements that are commonly executed together and should be further analyzed. In this study, we again apply a default threshold value of 25%, suggesting that any concept that contains more than 25% of all attributes in the system will be considered bloated. Figure 5-12 shows that one concept can be identified that contains 56% of the total of attributes and can be seen as a potential candidate for another sub-map.

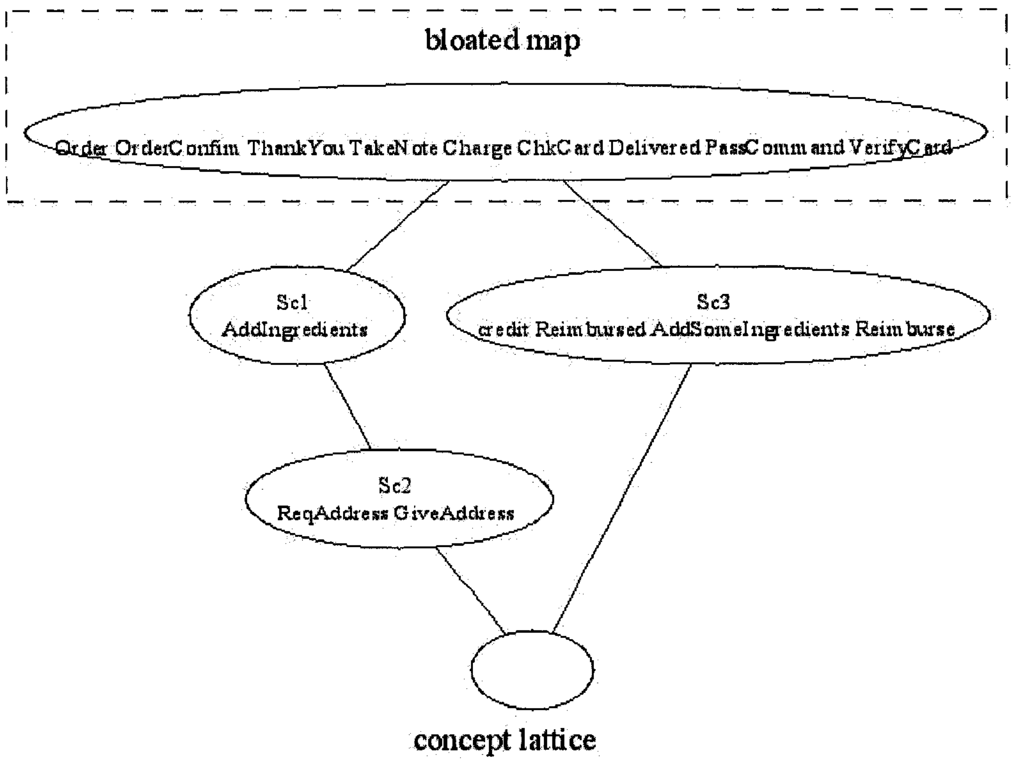

bloated map
Order OrderConfirm ThankYou TakeNote Charge ChkCard Delivered PassCommand VerifyCard
Sc1
AddIngredients
Sc3
credit Reimbursed AddSomeIngredients Reimburse
Sc2
ReqAddress GiveAddress
concept lattice

Figure 5-12: Concept Lattice for *Bloated Map Bad Smell*

5.4.3 Shotgun Surgery

For the *Shotgun Surgery* bad smell, we revisit the example used in Figure 5-9, but this time we look for those attributes that are shared among many scenarios, and therefore their modification would result in changes to many other scenarios. Similar to the previous defect measurement, we allow the user to decide on a threshold to be applied to identify the bad smell.

In the following example we apply a default value of 75%, reflecting that an attribute (UCM element, plug-ins, etc) is considered to cause a *Shotgun Surgery* bad smell if it is shared amongst more than 75% of the total number of concept objects (scenarios) in a system. Analyzing the elevator UCM (Figure 5-9), it is very difficult to determine exactly which elements are highly shared among different scenarios. Applying our UREAF tool we can identify such shared elements amongst the scenarios. In this concrete example (Figure 5-13), any element which is used among more than 5 scenarios (75% of 6 scenarios) will be flagged as a potential “Shotgun Surgery” bad smell .

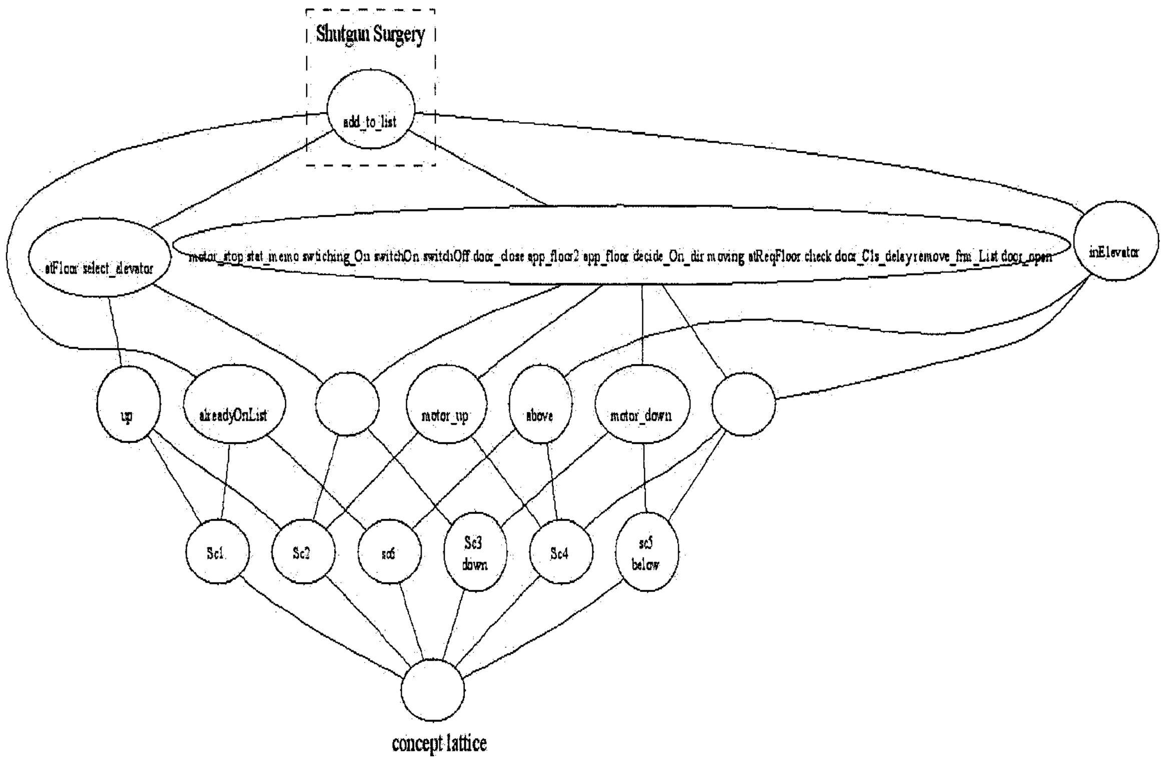

Shotgun Surgery
add_to_list
atFloor select_elevator
motor_stop stat_memo switching_On switchOn switchOff door_close app_floor2 app_floor decide_On_dir moving atReqFloor check door_Cls_delayremove_frm_List door_open
inElevator
up
alreadyOnList
motor_up
above
motor_down
Sc1
Sc2
sc6
Sc3 down
Sc4
sc5 below
concept lattice

Figure 5-13: Concept lattice representing Shotgun Surgery Bad Smell

As shown in Figure 5-13, the attribute in the top-most concept is shared by all scenarios (100%) making it a candidate for the shotgun surgery bad small.

5.4.4 Aggressive Scenario

In the next example, adopted from [120], an online store web-application is presented. For simplicity sake, we have chosen 6 of the 7-mentioned scenarios in [120] shown in Table 5-6. (In order to simplify the table, we do not show the global variables).

Scenario Group	Scenario Name	Description
BaseCase	Sc1	Customer buys one widget and everything works fine.
invalidAccount	Sc2	Customer can not download widgets without a valid account
SecondThoughts	Sc3	Customer brows again to review the cart.
MultipleProducts	Sc4	Customer buys many widgets and everything works fine.
EditingCart	Sc5	Customer brows again to review the cart and remove/add widgets to the cart
MulitpleOrderes_ConcurrentCustomers	Sc6	Two customer buy many widgets and download them

Table 5-6: Online Store Scenario Definition

System description: In the online store system, a customer is able to navigate through the store's homepage and will be presented with various categories of widgets. A customer can view each category and select widgets from them to be added to his/her shopping cart. The system will update the shopping cart and when no more items are added to it, the customer can proceed to the checkout system. The checkout system requires the customer to provide his/her account information. If the provided account information is valid, the system will build an order summary including the totals and number of items to be checked-out. Once the customer confirms the order, the payment will be processed and the invoice will be displayed. At this point, the customer can proceed to the download area where s/he can download the purchased widgets. If customer's account information is not valid, the process will terminate (Figure 5-14). It should be noted that the authors [120] included some intentional errors in the requirement specification for their own research purpose of detecting faults. Since these seeded errors did not affect our research context, we focus on detecting modeling problems rather than requirement faults. These original requirement errors were not removed.

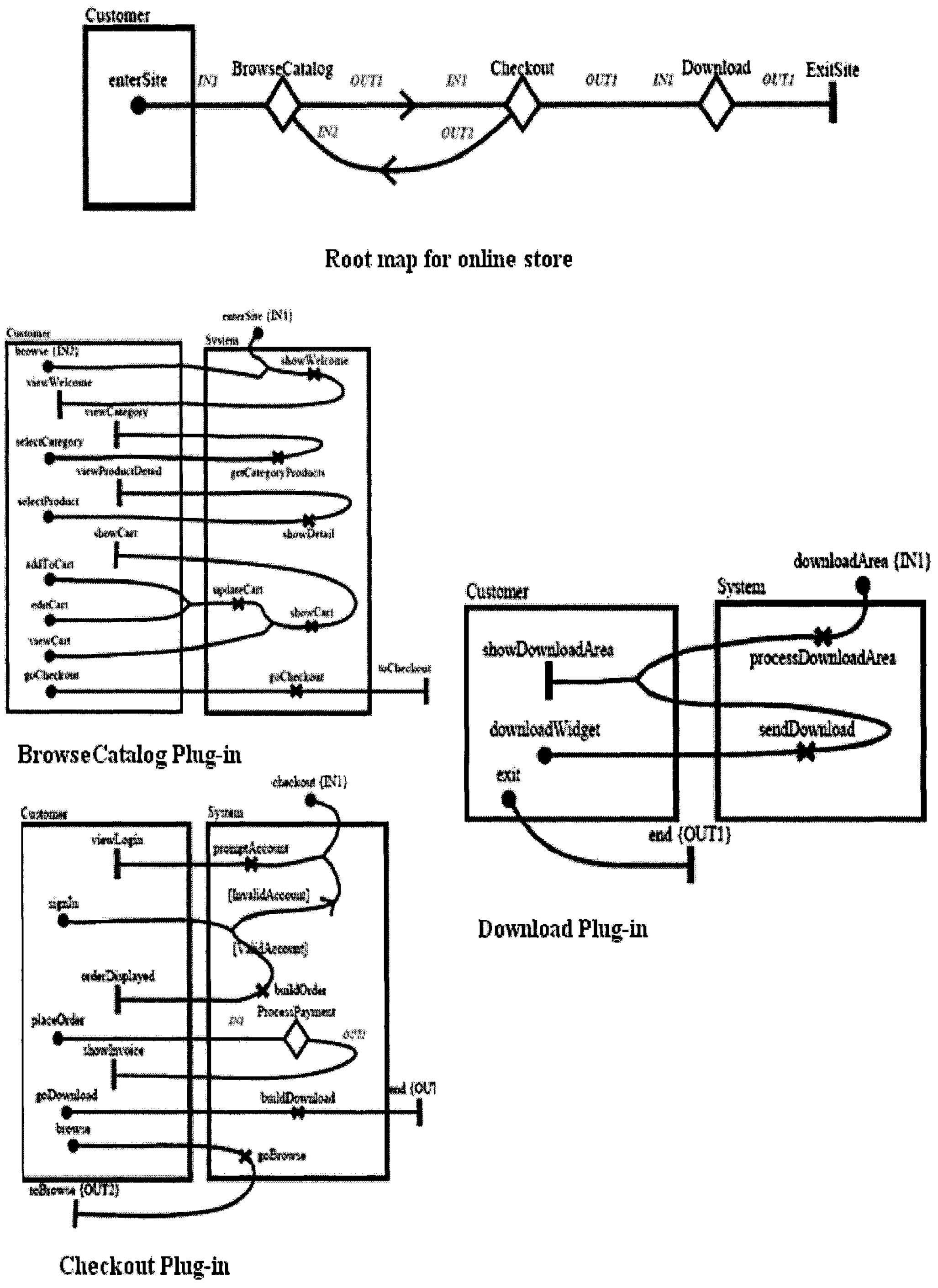
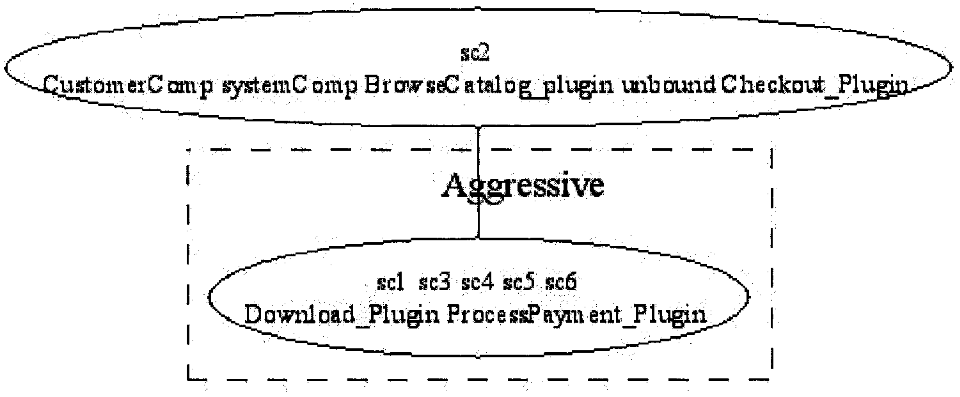


Figure 5-14: An Online Store UCM [120]

In order to detect whether a UCM contains aggressive scenarios, one has to determine which scenarios involve a large number of UCM components/plug-ins. This is difficult for large systems, especially when there might be more than one scenario with such conditions.

Once again, we apply the notion of a threshold, this time specifying a default value of 75%, indicating that a scenario that contains more than 75% of a system plug-ins/components is considered as being an aggressive scenario bad small.


sc2
CustomerComp systemComp BrowseCatalog_plugin unbound Checkout_Plugin
Aggressive
sc1 sc3 sc4 sc5 sc6
Download_Plugin ProcessPayment_Plugin

concept lattice
Figure 5-15: Concept Lattice Corresponding to Aggressive Scenario Bad Smell

Figure 5-15 shows that scenarios 1, 3, 5, and 6 contain all the existing plug-ins and components in the given system making the maintenance and modifications of this plug-in difficult to manage due to its coupling with all scenarios.

5.4.5 Lazy Plug-in/Component

For the *Lazy Plug-in/ Component* bad smell, we re-use the example used for the Feature Interaction case study (section 5.3). *Lazy* UCM constructs are good candidates to be either eliminated or merged with other existing constructs. In order to determine if a plug-in or a component belongs to this bad smell category, we adopted a default threshold value of 10%, indicating that a plug-in/component is considered to be lazy if it is contained in less than 10% of the total number of scenarios within a system.

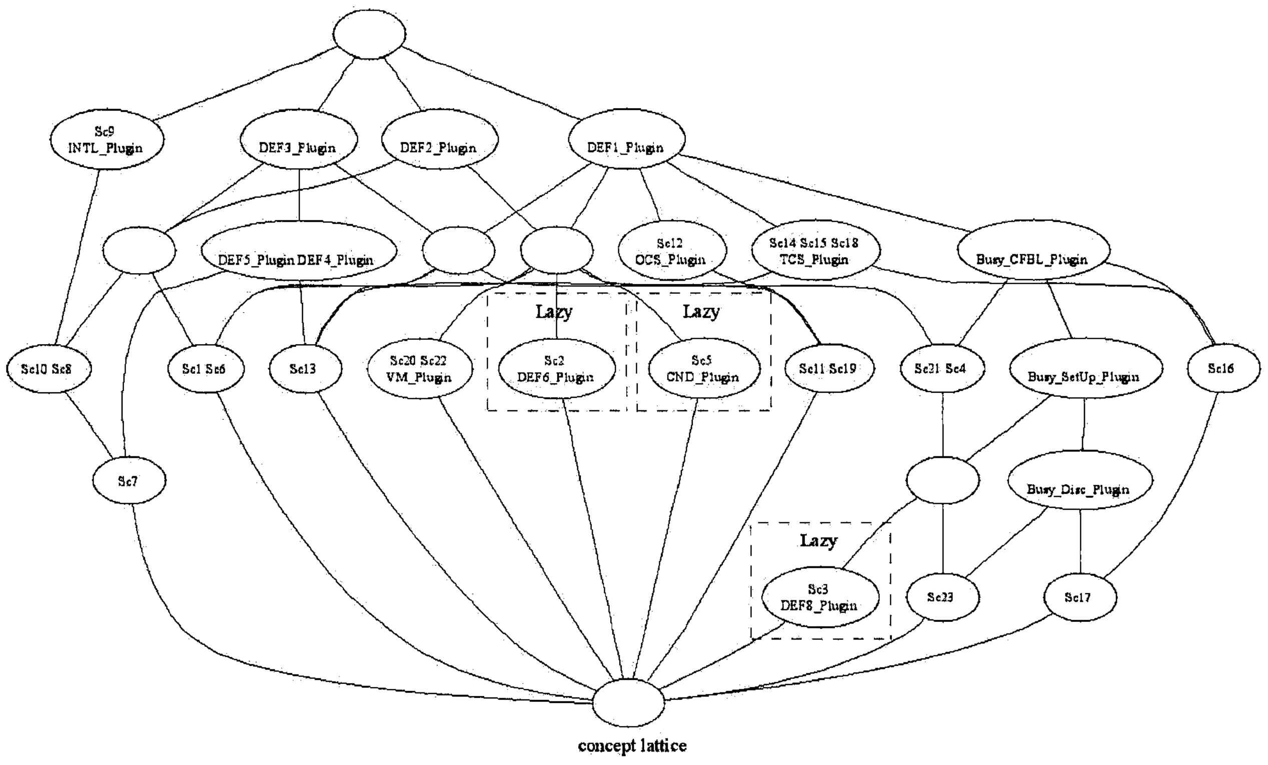


Figure 5-16: Lazy Plug-in example

From the analysis in Figure 5-16, we can see that all those plug-ins (dashed box) are used in one scenario only and are therefore flagged as a *Lazy* plug-in/component bad smell. One approach to fix these bad smells is to eliminate or merge these plug-ins with other maps. It has to be noted that one of the reasons these lazy plug-ins exist, could be the fact that they might have been intentionally created for further re-use or extension of future scenarios. This is left to the maintainer to decide.

5.5 Related Work and Discussion of the Results and Limitations

In this section we will discuss and compare our work with existing research that is closely related to ours with respect to requirements evolution by means of change impact analysis, regression testing, feature interaction analysis, and bad smell detection analysis.

Change Impact Analysis. A significant body of research exists on impact analysis with the majority of this work focusing on identifying changes and their impact at the source code level [13, 43, 67, 115]. There is less work on impact analysis at the design [19, 20, 61] or requirement specification level [52, 54, 61]. Change impact analysis at the code level has the advantage of being more accurate because it is based on final implementation of the design. However, it also requires maintainers to have a previous understanding of both the requirements and their mapping to source code in order to be able to identify and localize the potential change.

Our approach differs from these source code-based impact analysis approaches by identifying the potential impact of a change at the requirement level. Our methodology does not require either a previous knowledge of the source code to be analyzed or the source code itself. In [19], the authors proposed an impact analysis approach that can be applied on UML models to detect direct or indirect impacts at the UML level. However, their technique is based on traceability analysis and on the definition of specific change propagation rules, with neither their completeness nor their correctness being guaranteed [35]. Other approaches to identify the impact of requirement changes at the requirement and specification level can be found in [52, 61,93 ,54]. In both [61, 93], guidelines for changing requirements and design documents based on traceability techniques are introduced. Our approach differs from this work through identification of actual impacts using dependency analysis. In [52, 54], a lightweight approach base on Use Case Maps to analyze the potential impact of requirement changes on a system is introduced. However it was based on a static analysis approach leading to an imprecise handling of dynamic plug-ins. Furthermore, the dependency analysis was not fully automated and did not provide different levels of granularity. Our approach is automated and based on dependency analysis technique utilizing the benefits (analysis and logical clustering) of FCA. FCA has been previously applied in conjunction with slicing for determining the change impact analysis at the source code level [115] called "Concept Lattice of Decomposition Slices". FCA has also been applied previously for extracting class hierarchies at the specification level [109] and for recovering design patterns [116].

However, to the best of our knowledge, no previous work exists on utilizing FCA for change impact analysis at the requirement level. Our approach is based on dependency analysis technique and we believe it combines the benefits of formal modeling and the analysis and clustering capabilities of FCA. Furthermore it allows for visual representations of requirement changes. It allows us to easily generate different views to enable maintainers and managers to better understand the impact of a requirement change before actually committing to or implementing the change.

Regression Testing. Similar to the impact analysis approaches, most of the work on regression test selection has been focused on the source code level [44, 96,97, 98, 99,112] and some at the design level [20, 91]. However, to the best of our knowledge, there exists no previous work on selective regression testing by means of FCA at requirement level. In [20], a mapping between design changes in UML and code changes has been created to classify code tests. In [91], an approach for selective re-test strategy was presented which relies on categorizing changes to UML design.

The most closely related work to ours was presented in [112], where a greedy algorithm using FCA uses attribute and object implications among the requirements and test cases to minimize the number of test cases. However, it is source code-based and does not provide support for change impact, feature interaction, and bad smells analysis.

Feature Interaction. There exists a wide range of notations and techniques in the literature to describe features (LOTOS [8] [38], Chisel Diagram [3], Finite State Machines [15] Use Case Maps [4], etc). Common to all of these notations is that a need still remains for maintainers to be able to identify and understand the interaction among features in a system. Among the existing research in the FI problem domain [27, 60], the authors in [27] provide a comprehensive review of various feature interaction analysis approaches. Some of these techniques are based on formal methods which are very accurate however, their use in the industry is still quite restricted to people with the necessary background in formal methods and are often difficult to implement. In Chisel, for example, features are added in a "gluing nodes" fashion making it hard to identify scenarios within an entire system [68]. The use of FSM in their approach has the advantage of detecting all existing FIs; however, as the number of features grows the number of states in the model also grows larger exponentially. There have been several techniques proposed for FSM to reduce the number of states [28], however, these techniques can be very complex and expensive [68].

The most closely related work to ours is [68], in which the authors propose a two-phase FI filtering method based on UCM. A stub configuration is used and the analysis is performed on a SC-matrix. In comparison to this work [68], our approach is more conservative when it comes to distinguishing FI prone scenarios, and we are also not able to detect all FI free cases within those FI prone scenarios. Common to both approaches is their computation complexity and that both can be semi-automated. Our approach differs from [68] in that we apply FCA, which is a more flexible, domain independent analysis technique. Furthermore, because of FCA, the analysis is not limited to UCMs and its notation; in fact, any formalized notation that allows the generation of traces can be integrated with our FCA tool. Other differences between these two approaches are: 1) we determine FI between features from executable scenarios that resolve some of the non-determinism during the analysis; and 2) we can also assist maintainers in determining the potential impacts of a feature modification request by combining UCM with FCA.

Bad Smell. Research on bad smell has focused mainly on source code [39, 69, 83, 109] and design level [33, 86]. The work presented in [112] is similar to ours with respect to detecting bad smells by means of FCA. The approach uses concept analysis for refactoring object-oriented class hierarchies based on detected bad smells. In [115], FCA was applied to restructure software modules; however, both FCA-based approaches [109 and 115] are at the source code level. Moha et al. in [85] use formal concept analysis to "to Suggest Refactorings to Correct Design Defects. " However, they restrict their analysis to only the detection of cohesion and coupling by means of FCA. While some techniques have been used to detect bad smells and perform refactoring and restructuring at the formal requirement specification level [78, 79, 103], none of these techniques have used FCA in their analysis. In [78] only some simple rules are explained which include refactoring at method and class level , while in [103] a similar work has been done on all class system level and the authors in [78] claim that all refactoring should be covered by combining both approaches together.

Limitations and Discussion

We take advantage of FCA and its analysis and logical clustering flexibilities to support additional modification analysis (e.g. feature interaction). Furthermore, because of the use of FCA, the analysis is not limited to UCMs and its notation; in fact, any formalized and executable requirement notation can be integrated within our modification analysis framework.

We note that our approaches have several limitations, for example our change impact analysis approach similar to other impact analysis approaches supports only modifications or deletions. For new requirements, it is possible to use an iterative approach by introducing the modification at the UCM level and to compare the system behavior after re-executing the UCM scenarios. Also the accuracy of the FCA analysis depends on the quality of the traces. In the context of our research, we assume the executed scenarios achieve domain element coverage, meaning that every domain element was executed at least once.

From a FI analysis perspective, our approach has limitations with respect to the lack of a domain knowledge integration technique which could aid us in better clarifying cases of FI prone scenarios.

Other limitations of our system are caused by the fact that our UCM traces do not provide sufficient detail to allow for a more specific data dependency analysis, as well as the fact that FCA does not consider sequence of events during the analysis. Both of these limitations limit the ability of our approach to provide a minimization RTS technique.

The issue of scalability can be viewed as another limitation. Even though the approach conducts the analysis at the requirement specification level and the size of the *dependency lattice* might not get very large at this level, the possibility of encountering a very large lattice as a result of a large number of UCM traces still exists. A solution for this issue will be discussed in the future work section.

6. Conclusion and Future Work

In this research, we introduced a methodology to support the evolution analysis of requirements by applying FCA to UCM. We introduced the concept of UCM scenarios and different types of UCM dependencies and provided a complete methodology including algorithms and steps involved to apply FCA for UCM change impact, regression test selection, feature interaction and bad smell analysis. A proof of concept prototype environment was implemented to support our methodologies and its automation. The applicability of the presented methodologies and its tool support was demonstrated and discussed using existing published UCM examples.

The major contributions of this thesis can be restated as follows:

(1) We presented a novel approach by combining Formal Concept Analysis with formalized UCM traces to support requirement specification evolution analysis;

(2) We introduced different evolution analysis techniques, including change impact, regression testing, feature interaction and bad smells detection at the requirement specification level;

(3) We implemented UREAF_tool as a proof of concept, which implements the introduced algorithms and methodologies and automates the analysis process.

As part of future investigation, finding new techniques to enrich FCA for detecting direct and indirect impact analysis in a way that would reduce the number of false positive cases would be interesting.

Moreover, there is a need to further evaluate and validate our approach by applying our techniques on larger case studies, especially cases studies from industries other than telecommunications and phone industry.

Also scalability issues need to be addressed, for example, by hiding unrelated parts in the concept lattice representation.

Furthermore, when it comes to regression test selection, it would be interesting to implement other techniques such as prioritizing techniques and compare those results with our current approach. In respect to feature interaction analysis, a way to integrate domain knowledge in order to determine FI cases would be interesting. Finally, in this research we are not considering synchronization and the notion of time. It would be interesting to investigate if our methodologies can fully support timed UCMs or not.

7. References

1. S. Anderson and M. Felici, "Requirements changes risk/cost analyses: An avionics case study," *Foresight and Precaution, Proceedings of ESREL,* vol. 2, pp. 921–925, 2000.

2. R.S. Arnold. "Software Change Impact Analysis." *IEEE Computer Society Press,*Los Alamitos, CA, USA, 1996

3. A. Aho, S. Gallagher, N. Griffeth, C. Scheel, and D. Swayne. " Sculptor with Chisel: Requirements Engineering for Communications Services". *Fifth International Workshop on Feature Interactions in Telecommunications and Software Systems (FIW'98),* Lund, Sweden, September, 1998, 45-63.

4. D. Amyot. "Use Case Maps as a Feature Description Notation." *FIREwork Feature Constructs Workshop*, 2000.

5. D. Amyot. "*Formalization of Timethreads Using LOTOS*". Master Thesis, Department of Computer Science, University of Ottawa, Canada, 1994.

6. D. Amyot , R.J.A Buhr., T. Gray. and L. Logrippo, "Use Case Maps for the Capture and Validation of Distributed Systems Requirements". RE'99, Fourth IEEE International Symposium on Requirements Engineering, Limerick, Ireland, June 1999,44-53. http://www.UseCaseMaps.org/pub/re99.pdf

7. D. Amyot , R. and Andrade, "Description of wireless intelligent network services with Use Case Maps," SBRC'99, 17th Simp´osio Brasileiro de Redes de Computadores, Salvador, Brazil, May 1999, pp. 418-433.

8. D. Amyot, L. Charfi, N. Gorse, T. Gray, L. Logrippo, J. Sincennes, B. Stepien T. Ware. "Feature Description and Feature Interaction Analysis with Use Case Maps and LOTOS." I*n: M.Calder and E. Magill, Sixth International Workshop on Feature Interactions in Telecommunications and Software Systems*, IOS Press, Glasgow 2000, 274-289.

9. D.Amyot, and L. Logrippo. "Directions in feature interaction research." Computer Networks, vol. 45, pp. 563-567, 2004.

10. D. Amyot and G. Mussbacher, "Bridging the Requirements/Design Gap in Dynamic Systems with Use Case Maps (UCMs)," *Proce.s 23rd Int. Conf. on Software Engineering (ICSE)*, pp. 743-744, 2001.

11. Arnold, R. S., and Bohner, S. A., "Impact Analysis -Towards A Framework for Comparison," *Proc. of the Conf. onSoftware Maint.*, pp. 292-301, Sept. 1993.

12. AT&T Labs-Research, Graphviz, http://www.graphviz.org. Last accessed, October 2007.

13. L. Badri, M. Badri and D. St-Yves, "Supporting Predictive Change Impact Analysis: A Control Call Graph Based Technique," *Proc. 12th Asia-Pacific SE Con. (APSEC'05)*-Volume 00, pp. 167-175, 2005.

14. G. Birkhoff. Lattice theory, Providence, Rhode Island: Amer.Math.Soc, 1967.

15. G. V.Bochmann, "Finite State Description of Communication Protocols." *In: Computer Networks,* Vol. 2 (1978), 361-372.

16. S. Bohner, "Software change impacts-an evolving perspective," *Software Maintenance, 2002.Proceedings.International Conference on,* pp. 263-272, 2002.

17. S. A. Bohner and R. S. Arnold, "An Introduction to Software Change Impact Analysis," *Software Change Impact Analysis*, pp. 1–26, 1996.

18. Bohner, S. A., "A Graph Traceability Approach to Software Change Impact Analysis," Ph.D. Dissertation George Mason University, Fairfax, VA, 1995.

19. L. Briand, Y. Labiche, L. O'Sullivan and M. Sówka, "Automated impact analysis of UML models," The J. of Systems & Software, vol. 79, pp. 339-352, 2006. (8)

20. L. Briand, Y. Labiche and G. Soccar, "Automating impact analysis and regression test selection based on UML designs," *Proc. of the Int. Software Maintenance Conference (ICSM),* pp. 252-261, 2002.

21. F. P. Brooks, "No Silver Bullet Essence and Accidents of Software Engineering," *Computer,* vol. 20, pp. 10-19, 1987.

22. R. J. A. Buhr, D. Amyot, M. Elammari, D. Quesnel, T. Gray, and S. Mankovski, "Feature-Interaction Visualization and Resolution in an Agent Environment," *Feature Interactions in Telecommunications and Software Systems V,* 1998.

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

24. R.J.A. Buhr, R.S. Casselman, T.W. Pearce, "Design Patterns with Use Case Maps: A Case Study in Reengineering an Object-Oriented Framework", SCE 95-17, http://ftp.sce.carleton.ca/UseCaseMaps/ dpwucm.ps.

25. R. J. A.Buhr, M. Elammari, T. Gray and S. Mankovski , "Applying Use Case Maps to multiagent systems: A feature interaction example". *In 31st Annual Hawaii International Conference on System Sciences*, 1998.

26. N. Nakamura, T. Kikuno, J. Hassine, and L. Logrippo, "Feature Interaction Filtering with Use Case Maps at Requirements Stage". *In: Sixth International Workshop on*

Feature Interactions in Telecommunications and Software Systems (FIW'00), Glasgow, Scotland, UK, May 2000.

27. M. Calder, M. Kolberg , E. H. Magill, and S. Reiff-Marganiec. "Feature interaction: a critical review and considered forecast." *Computer Networks*, vol. 41, pp. 115-141, 2003.

28. E. Cameron and H. Velthuijsen, "Feature interactions in telecommunications systems," *Communications Magazine, IEEE,* vol. 31, pp. 18-23, 1993

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

30. E.J. Chikofsky and J.H. Cross, "Reverse Engineering and Design Recovery: A Taxonomy," IEEE Software, vol. 7, no. 1, pp. 13-17, 1990.

31. A. Ciemniewska, J. Jurkiewicz, Ł Olek and J. Nawrocki, "Supporting Use-Case Reviews⋆," *LECTURE NOTES IN COMPUTER SCIENCE,* vol. 4439, pp. 424, 2007.

32. R G..Crespo, M.Carvalho, and L. Logrippo, "Distributed resolution of feature interactions for internet applications". Computer Networks, vol. 51, pp. 382-397, 2007.

33. Ł Dobrzański and L. Kuźniarz, "An approach to refactoring of executable UML models," *Proceedings of the 2006 ACM Symposium on Applied Computing,* pp. 1273-1279, 2006.

34. R. Duke, P. King, G. Rose and G. Smith, "The Object-Z specification language," *Technology of Object-Oriented Languages and Systems: TOOLS,* vol. 5, pp. 465-483, 1991.

35. A. Egyed, "Fixing Inconsistencies in UML Design Models," *Proceedings of the 29th International Conference on Software Engineering,* pp. 292-301, 2007.

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

37. H. C. Estler, T. Ruhroth and H. Wehrheim, "Modelchecking Correctness of Refactorings-Some Experiments," *Electronic Notes in Theoretical Computer Science,* vol. 187, pp. 3-17, 2007.

38. M. Faci, L. Logrippo and B. Stepien. "Formal Specifications of telephone Systems in LOTOS." P*rotocol Specification, Testing and Validation IX*, eds. E. Brinksma, G. Scolo, and C. Vissers,1990.

39. M. Fowler and K. Beck, "Bad Smells in Code," *in Refactoring:Improving the Design of Existing Code*," Addison-Wesley,2000, pp. 75-88.

40. A.Gammelgaard and J. E. Kristensen." Interaction Detection, a logical approach. In: L.G. Bouma and H. Velthuijsen (eds.) Feature Interactions in Telecommunications Systems. IOS Press, 1994 (Proc. of the 2nd International Workshop on Feature Interactions in Telecommunications Systems, Amsterdam) 178-196.

41. B. Ganter and R. Wille, "Formal Concept Analysis: Mathematical Foundations." Springer-Verlag NY, 1997.

42. B. Ganter and R. Wille, "Applied Lattice Theory: Formal Concept Analysis".Preprints, 1997.

43. T. Goradia, "Dynamic impact analysis: a cost-effective technique to enforce error-propagation," *Proceedings of the 1993 International Symposium on Software Testing and Analysis,* pp. 171-181, 1993.

44. T. L. Graves, M. J. Harrold, J. M. Kim, A. Porter and G. Rothermel, "An empirical study of regression test selection techniques," *ACM Transactions on Software Engineering and Methodology* ,Vol.10,2001 pp. 184-208, 2001.

45. N. Griffeth , R. Blumenthal, J. Gregoire , and T. Ohta. "Feature interaction detection contest." Feature Interactions in Telecommunications Systems V, pp. 327–359.

46. R. Gupta, M.J. Harrold, and M.L. So®a." An approach to regression testing using slicing." *In Proceedings of the Conference on Software Maintenance*, pages 299-308, November 1992.

47. D. Harada, H. Fujiwara and T. Ohta, "Avoidance of Feature Interactions at Run-Time," *Software Engineering Advances, International Conference on,* pp. 6-6, 2006.

48. J. Hartmann, D. Robson, " Techniques for Selective Revalidation." *IEEE Software, Jan* 1990, p.31-38.

49. J. Hassine. "Feature Interaction Filtering and Detection with Use Case Maps and LOTOS." Master's thesis, University of Ottawa, Ottawa, Ontario, Canada, 2001.

50. J. Hassine, J. Rilling and R. Dssouli, "An ASM Operational Semantics for Use Case Maps," *.Proc.13th IEEE Inter. Conf. on Requirements Engineering*, pp.467-468, 2005.

51. J. Hassine, J. Rilling, and R. Dssouli. Abstract operational semantics for use case maps. *In Formal Techniques for Networked and Distributed Systems - FORTE 2005, 25th IFIP WG 6.1 International Conferen*ce, Taipei, Taiwan, October 2-5, pages 366–380, 2005.

52. J. Hassine, J. Rilling, J. Hewitt and R. Dssouli, "Change Impact Analysis for Requirement Evolution using Use Case Maps," *Proc. of the 8th Int. Workshop on Principles of Software Evolution*, pp. 81-90, 2005.

53. J. Hassine, R. Dssouli and J. Rilling, "Applying Reduction Techniques to Software Functional Requirement Specifications," 4th SDL and MSC Workshop (SAM'04), 2004.

54. J. Hewitt and J. Rilling, "A light-weight proactive software change impact analysis using use case maps," *IEEE Int. Workshop on Software,Evolvability* pp. 41-46, 2005 .

55. S. Ibrahim, N. B. Idris, M. Munro and A. Deraman, "A Requirements Traceability to Support Change Impact Analysis," *Asian Journal of Information Technology, Pakistan,* vol. 4, pp. 345-355, 2005.

56. ITU-T, Recommendation Z.150, User Requirements Notation (URN), Geneva, Switzerland.

57. International Standard - ISO/IEC 14764 IEEE Std 14764-2006 Software Engineering, Software Life Cycle Processes, Maintenance, ISBN: 0-7381-4961-6, 2006 (33_1

58. ITU-T, URN Focus Group (2003), Draft Rec. Z.152 - UCM: Use Case Map Notation (UCM), Sept. 2003.

59. jUCMNav. jucmnav project (tool, documentation, and meta-model). http://jucmnav.softwareengineering.ca/twiki/bin/view/ProjetSEG/WebHome, 2006.Last accessed, October 2007.

60. D. O. Keck, and P. J. Kuehn. "The feature and service interaction problem in telecommunications systems: a survey". *IEEE Trans. Software Eng.*, vol. 24, pp. 779-796, 1998.

61. A. v. Knethen,"Change-oriented requirements traceability. Support for evolution of embedded systems," *Proceedings International Conference on Software Maintenance (ICSM).* pp. 482-485, 2002.

62. M. Kolberg, E. H. Magill, D. Marples, and S. Reiff. Second feature interaction contest. *Proc.of Sixth Int'l.Workshop on Feature Interactions in Telecommunication Networks and Distributed Systems (FIW'00)*, pp. 293-310, 2000 (35) pp. 246-250.

63. B. Korel and J. Laski, "Dynamic slicing of computer programs," J. Syst. Software, vol.13, pp.187-195, 1990.

64. B. Korel, L. H. Taha, "Understanding Modifications in State-Based Models," *12th International Workshop on Program Comprehension (IWPC)*, 2004, Italy, pp. 246-250.

65. W. Lam, Martin Loomes, V. Shankararaman, “Managing Requirements Change Using Metrics and Action Planning,” *Conference on Software Maintenance and Reengineering,* 1999, pp. 122-129.

66. T.F. La Porta, D. Lee, Y. J. Lin, and M.Yannakakis. “Protocol Feature Interactions.” *Proceedings of the FIP TC6 WG6.1 Joint International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols (FORTE XI) and Protocol Specification, Testing and Verification* (PSTV XVIII), pp. 59-74, 1998.

67. J. Law and G. Rothermel, "Whole program path-based dynamic impact analysis," Proceedings of the International Conference on Software Engineering, pp. 308–318, 2003.

68. P. Leelaprute, N. Nakamura, K. Matsumoto, and T. Kikuno, “Design and Evaluation of Feature Interaction Filtering with Use Case Maps”. NECTEC Technical Journal, pp. 581-597, 2005.

69. M M. Lehman, J F. Ramil, P D. Wernick, D E. Perry, and W M. Turski. “Metrics and laws of software evolution” - the nineties view. In METRICS ’97: *Proceedings of the 4th International Symposium on Software Metrics*, page 20, Washington, DC, USA, 1997. IEEE Computer Society.

70. M. Lehman and L. Belady, Program Evolution: Processes of Software Change. Academic Press Professional, Inc. San Diego, CA, USA, 1985.

71. C. Lindig.” *Introduction to Formal Concept Analysis”*, Harvard University, 2000, http://www.st.cs.uni-sb.de/~lindig/talks/ Last Accessed 2006.

72. C. Lindig. “Concept-based component retrieval.” *IJCAI95 Workshop on Formal Approaches to the Reuse of Plans, Proofs,* and Programs, 1995.

73. C. Lindig; G. Snelting. “ Assessing Modular Structure of Legacy Code Based on Mathematical Concept Analysis.” Software Engineering, , ICSE, 1997.

74. M. Lindvall and K. Sandahl, "How well do experienced software developers predict software change?" *The Journal of Systems & Software*, vol. 43, pp. 19-27, 1998.

75. A. Maedche, G. Neumann and S. Staab, "Bootstrapping an Ontology-Based Information Extraction System," Studies in Fuzziness and Soft Computing, Intelligent Exploration of the Web.Springer, 2002.

76. M. Mantyla, J. Vanhanen and C. Lassenius, "Bad smells-humans as code critics," Software Maintenance, 2004.Proceedings.20th IEEE International Conference on, pp. 399-408, 2004.

77. M. Mantyla, J. Vanhanen and C. Lassenius, "A taxonomy and an initial empirical study of bad smells in code," *Software Maintenance, 2003.ICSM 2003.Proceedings.International Conference on*, pp. 381-384, 2003.

78. T. McComb, "Refactoring Object-Z specifications," *FASE'04: Fundamental Approaches to Software Engineering*, 2004.

79. T. McComb and G. Smith.Architectural ."Design in Object-Z." *In Australian Software Engineering, Conference (ASWEC'04),* pages 77 – 86. IEEE Computer Society Press, 2004.

80. T. McComb and G. Smith. "Refactoring object-oriented specifications: A process for deriving designs."*Technical Report SSE-2006-01, School of Information Technology and Electrical Engineering*, University of Queensland, Australia, May 2006.

81. A. Miga, D. Amyot, F. Bordeleau, C. Cameron and M. Woodside, "Deriving Message Sequence Charts from Use Case Maps Scenario Specifications," *Tenth SDL Forum (SDL'01),* pp. 268-287, 2001.

82. Microsoft, "AsmL for Microsoft.Net," research.microsoft.com/foundations/asml, April/2007.

83. T. Mens and T. Tourwe, "A survey of software refactoring," Software Engineering, IEEE Transactions on, vol. 30, pp. 126-139, 2004.

84. A. Miga. "Application of use case maps to system design with tool support." Master's thesis, Dept. of Systems and Computer Engineering, Carleton University,Ottawa, Canada, 1998.

85. N. Moha, J. Rezgui, Y. Guéhéneuc, P. Valtchev, G. El Boussaidi. "Using FCA to Suggest Refactorings to Correct Design Defects". *Proceedings of the 4th International Conference On Concept Lattices and Their Applications* (CLA 2006), pp. 297-302 , In S. Ben Yahia & E. Mephu Nguifo Eds, October 30-November 1st, 2006, Hammamet, Tunisia.

86. N. Moha, S. Bouden and Y. Guéhéneuc, "Correction of High-Level Design Defects with Refactorings," *In Proc.of the ECOOP Workshop on Object-Oriented Reengineering WOOR,* 2006.

87. V. Nanda, N. H. Madhavji, "The Impact of Environmental Evolution on Requirements Changes," *Int'l Conference on Software Maintenance 2002,* pp. 452-461.

88. W.F. Opdyke, "Refactoring: A Program Restructuring Aid in Designing Object-Oriented Application Frameworks," PhD thesis, Univ. of Illinois at Urbana-Champaign, 1992.

89. D. Petriu, M. Woodside, "Software Performance Models from System Scenarios in Use Case Maps", Carleton Univ. Canada, Proc. Performance TOOLS 2002, London.

90. S. L. Pfleeger, “Software Engineering: Theory and Practice”. Prentice-Hall, Inc. Upper Saddle River, NJ, USA, 1998.

91. O. Pilskalns and A. Andrews, "Regression Testing UML Designs," *Proc. IEEE International Conference on Software Maintenance (ICSM'06)*-Volume 00, pp. 254-264, 2006.

92. J. Ratzinger, M. Fischer and H. Gall, "Improving evolvability through refactoring," *ACM SIGSOFT Software Engineering Notes*, vol. 30, pp. 1-5, 2005.

93. B. Regnell, M. Andersson, J. Bergstrand,” A hierarchical use case model with graphical representation”, *In Proc IEEE Symposium and Workshop on Requirements,1996*, pp.270 – 277.

94. Reps, T.; Siff, M.; Identifying Modules via Concept Analysis, In IEEE ICSM, Italy, Oct 1997, p.170-179

95. S. W. K. Rochimah, W. M. N. Abdullah and H. Abdul, "An Evaluation of Traceability Approaches to Support Software Evolution," *Software Engineering Advances, 2007.ICSEA 2007.International Conference on*, pp. 19-19, 2007

96. G. Rothermel and M. Harrold, "Analyzing regression test selection techniques," Software Engineering, IEEE Transactions on, vol. 22, pp. 529-551, 1996

97. G. Rothermel and M. J. Harrold, "A safe, efficient regression test selection technique," *ACM Transactions on Software Engineering and Methodology (TOSEM),* vol. 6, pp. 173-210, 1997

98. G. Rothermel, S. Elbaum, A. G. Malishevsky, P. Kallakuri and X. Qiu, "On test suite composition and cost-effective regression testing," ACM Transactions on Software Engineering and Methodology (TOSEM), vol. 13, pp. 277-331, 2004.

99. G. Rothermel, R. Untch, C. Chu, and M.J. Harrold. Test case prioritization. IEEE Transactions on Software Engineering, October 2001

100. J. Roy, J. Kealey, and D. Amyot. Towards integrated tool support for the user requirements notation. In *SAM 2006*, pages 198-215, 2006.

101. A. Russo, B. Nuseibeh, and J. Kramer, “Restructuring Requirements Specifications for Managing Inconsistency and Change: A Case Study,” Proc. Int’l Conf. 57

102. M. Ruth and S. Tu, "A Safe Regression Test Selection Technique for Web Services," *Internet and Web Applications and Services, 2007.ICIW'07.Second International Conference on*, pp. 47-47, 2007.

103. T. Ruhroth, "Refactoring Object-Z Specifications," 18th Nordic Workshop on Programming Theory, 2006.

104. R. Settimi, J. Cleland-Huang, O. Ben Khadra, J. Mody, W. Lukasik, C. DePalma, “Supporting Software Evolution through Dynamically Retrieving Traces to UML Artifacts,” *7th International Workshop on Principles of Software Evolution (IWPSE)*, 2004, pp. 49-54

105. M. Shiri, J. Hassine, J. Rilling, "A Requirement Level Modification Analysis Support Framework," *software-evolvability*, pp. 67-74, *Third International IEEE Workshop on Software Evolvability* 2007.

106. M. Shiri, J. Hassine, and J. Rilling. “ Modification analysis support at the requirements level.” In *Ninth international Workshop on Principles of Software Evolution: in Conjunction with the 6th ESEC/FSE Joint Meeting* (Dubrovnik, Croatia, September 03 - 04, 2007). IWPSE '07.

107. M. Shiri, J. Hassine, and J. Rilling. Feature interaction analysis: a maintenance perspective. In *Proceedings of the Twenty-Second IEEE/ACM international Conference on Automated Software Engineering* (Atlanta, Georgia, USA, November 05 - 09, 2007). ASE '07.

108. G. Snelting,” TOSEM Re-engineering of Configurations Based on Mathematical Concept Analysis”, 1996.

109. G. Snelting, F. Tip,“Reengineering Class Hierarchies Using Concept Analysis “,*ACM Transactions on Prog. Languages and Systems* 5/ 2000, p.540-582.

110. I. Sommerville, "Integrated requirements engineering: a tutorial," Software, IEEE, vol. 22, pp. 16-23, 2005.

111. A. Srivastava, and J. Thiagarajan., "Effectively prioritizing tests in development environment," *ACM SIGSOFT International Symposium on Software Testing and Analysis*, Roma, Italy, 2002, pp. 97-106.

112. S. Tallam and N. Gupta, "A concept analysis inspired greedy algorithm for test suite minimization," *Program Analysis for Software Tools and Eng.* pp. 35-42, 2005.

113. T. Tilley, R. Cole, P. Becker and P. Eklund, "A Survey of Formal Concept Analysis Support for Software Engineering Activities," *Proc.of 1st Int. Conference on Formal Concept Analysis*, 2003.

114. T. Tilley, W. Hesse and R. Duke, "A software modelling exercise using FCA," *Proceedings of the 11th International Conference on Conceptual Structures (ICCS'03)*, Springer LNAI 2746, Dresden, Germany, 2003.

115. P. Tonella, "Using a concept lattice of decomposition slices for program understanding and impact analysis," *IEEE Trans. Software Eng.*, vol. 29, pp. 495-509, 2003.

116. P. Tonella and G. Antoniol, "Inference of object-oriented design patterns," *Journal of Software Maintenance and Evolution Research and Practice,* vol. 13, pp. 309-330, 2001

117. UCM, « Use Case Maps Web Page and UCM User Group », http://jucmnav.softwareengineering.ca/twiki/bin/view/UCM/WebHome, 2006.

118. UCMNav. ucmnav project (tool, features, documentation, example, and meta-model). http://jucmnav.softwareengineering.ca/twiki/bin/view/UCM/UcmNav, 2005. Last accessed, December 2007.

119. M. Weiser, "Program slicing," *Proceedings of the 5th International Conference on Software Engineering*, pp. 439-449, 1981.

120. M. Weiss and D. Amyot, "Business Process Modeling with URN," *International Journal of E-Business Research*, vol. 1, pp. 63-90, 2005.

121. Weiss, M. and Amyot, D.: Designing and Evolving Business Models with URN.*Montreal Conference on eTechnologies (MCeTech)*, Montréal, Canada, January 2005, 149-162.

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

123. J. Woodcock and J. Davies, *Using Z: Specification, Refinement, and Proof.* Prentice-Hall, Inc. Upper Saddle River, NJ, USA, 1996.