Specification-Level Change Impact Analysis with Use Case Maps

Jacqueline Hewitt

A thesis
in
The Department
of
Computer Science and Software Engineering

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

April, 2006

Your file Votre référence
ISBN: 0-494-14321-5
Our file Notre référence
ISBN: 0-494-14321-5

Abstract

Specification-Level Change Impact Analysis with Use Case Maps

Jacqueline Hewitt

Changes in both customer needs and technology are driving factors influencing software evolution. Consequently, there is a need to assess the impact of these changes on existing software systems. Currently, the majority of change impact analyses approaches focus on determining changes at the source code level, requiring already an understanding of the source code and the system. There exists a need to raise the level of abstraction to be able to analyze and predict the potential impact of changes on a system without the need to comprehend the underlying source code.

In this research, we present a lightweight approach to identify the impact of requirement changes at the specification level. We use specification information included in Use Case Maps to analyze the potential impact of requirement changes on a system. We propose dependency definitions and algorithms to identify Use Case Map scenario and component relationships. Also, techniques for ripple effect analysis at the scenario, component and element levels of abstraction are presented. Further, we present our tool that implements the proposed approaches showing the possibility of automation. A simple case study makes use of this tool to analyze an existing Use Case Map to show the information that is returned by our approach and its applicability in change impact analysis.

Acknowledgements

I would like to thank my supervisor, Dr. Rilling, for his guidance, patience, support, and confidence in my work. I am grateful that he provided me the opportunity to pursue this research.

I would also like to thank my family and friends for their dedicated support. Their much needed advice and encouragement supplied throughout this thesis has made this accomplishment possible. I would like them to know that they are loved.

Dedication

To my mother, Cynthia Stephen-Hewitt, who made possible the life that I know.

Table of Contents

List of Figures

List of Tables

1. Introduction

Currently, the majority of change impact analysis approaches focus on determining changes at the source code level, requiring an understanding of the source code and the system. There exists a need to raise the level of abstraction to be able to analyze and predict the potential impact of changes on a system without the need to comprehend the underlying source code. In this chapter we will introduce the present state of change impact analysis and highlight the need for impact analysis to be performed at a higher level of abstraction. This will signify the relevance of the presented research that seeks to promote Use Case Maps as a viable model on which to perform change impact analysis.

1.1 The Current State of Change Impact Analysis

Changes in both customer needs and technology are driving factors influencing software evolution. As such, there is a need to assess the impact of these changes on existing software systems. For many users, technology is no longer the main problem, and it is likely to become a progressively smaller problem as standard solutions are provided by technology vendors. Instead, there is a need for research to start focusing on the interface of the software with business practices.

Currently, the majority of change impact analysis approaches focus on determining changes at the source code level [GOR93, LAW03, TON03]. Although this often results in an accurate analysis of change impact (since source code represents the final implementation of the design), the analysis itself is extremely time consuming. Also the information provided by the source code analysis can be overwhelming, especially if one is only interested in an overall assessment of the affected components related to potential requirements changes [ARN93].

For large systems, whether confined or distributed, research ([SET04][BOH02]) claims that it is difficult to use a low-level change impact methodology to analyse the effects of a change because those techniques are usually focused on instances where the code has been changed. To use low-level change impact analysis techniques to determine all possible ripples effects of a large system would be inefficient and time consuming.

Therefore a need exists to analyze, predict, and assess the impact of a requirement or specification change at the design or requirement level, "this is due to the increasing trend in Software Engineering towards Model Driven Development" [SET04]. As well, the proactive methodology has been shown to be more beneficial than the traditional reactive one; predictive impact analysis where software maintainers complete a change implementation plan is better than dealing with the consequences of the change after it is actualized [LAW03]. Predictive impact analysis typically requires a complete understanding of the system in order to assess the scope of the change.

1.2 The Use Case Maps Solution

The ability to assess the effect of a change at the level of a design or specification requires a notation that describes the relationships of the system. We propose the use of Use Case Maps (UCMs) as a model for performing change impact analysis. To our knowledge change impact analysis approaches have not yet been applied to the UCM specification model.

The application of UCMs to maintenance activities allows comprehension and change impact analysis to be performed on the same model. Specifically, UCMs incorporate use case scenarios and system components into its design that provides the ability to trace user requirements described in use case scenarios to the relevant system components. Also, the system structure coupled with behavioural information has the advantage of providing a high level and dynamic view of the system functionality. Finally, UCMs encompass both the software and non-software requirements needed to provide an architectural view of all entities involved in a system's functionality. These features make UCMs a desirable model for performing change impact analysis.

1.3 Research Hypothesis and Goals

The goal of this thesis is to address the issue of change impact analysis at the architectural level. We suggest 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 – before the underlying source code has to be (fully) comprehended or an actual modification is performed.

Our research hypothesis, therefore, is that change impact analysis techniques can be applied to UCMs. Specifically, we conjecture that UCMs provide enough information about the structure and behaviour of the system under analysis that they can be used to help scope the impact of a change through the use of dependency analysis. The Null-Hypothesis is that UCMs do not provide sufficient information to be useful for change impact analysis.

In order to prove our hypothesis, information provided by UCMs will be used for the application of current change impact analysis techniques. The following are the intended sub-goals of this research:

1. Apply existing dependency definitions to UCM components and scenarios to generate dependency relationships for the purpose of system comprehension.
2. For a changed UCM element, generate an impact set for ripple effects.
3. Automate the creation of dependency relationships and extraction of impact sets by means of UCMs.

If all these can be implemented then we will conclude that our hypothesis has been proven and the Null-Hypothesis can be rejected.

Our proposed research goals have the capability of providing an efficient, industry-viable change impact analysis approach at a time when it is greatly need – a time when software systems are large and widely distributed and the difficulty of comprehending these systems at the source code level becomes an ever growing factor.

1.4 Organization of Thesis

Chapter 2 provides the necessary background related to software evolution, specifically change impact analysis and its current approaches. The definitions and algorithms that will be applied to UCMs are introduced in order to provide the basis for their use. Background discussion is continued in Chapter 3 where the focus is on UCMs. We propose our approach of applying change impact analysis techniques to UCMs in Chapter 4. To show that indeed automation is possible, our developed tool that implements our presented theories is detailed in Chapter 5. This is followed in Chapter 6 with the application of the tool to a simple case study. Finally, conclusions and future works are discussed in Chapter 7.

2. Software Evolution

In developing software systems, it is rare that the initial design is the final design or implementation. In fact, software is used to implement solutions that are expected to change periodically to adapt to environmental changes [BOH96]. In general, after the system is complete and in use change usually occurs. Lehman's first law of software evolution [LEH80] states that "a program that is used in a real-world environment necessarily must change or become progressively less useful in that environment"; any changes made after the initial deployment of the software are considered evolutionary in nature [BEN00]. Evolution is critical in the life cycle of all software systems and the efficient management and execution of these changes are essential to software quality [LEH80].

This chapter details the related areas of software system evolution, including requirements evolution, software maintenance, software comprehension techniques, and finally change impact analysis.

2.1 Requirements Evolution

Although requirements changes are not the only cause of system evolution, they do account for approximately 80% of software maintenance activities [LAM99]. When requirements evolve the design that specifies the system's implementation needs to also evolve to reflect any changes [NAN02]. In this section we discuss requirements evolution since this research is based to some extent on requirements.

A requirements change is a modification or deletion of an existing requirement or the addition of a new one. Requirements changes occur for both social and technical reasons including: users needs change leading to a new or modified feature [JAV04][LAM98]; production constraints [JAV04]; environment changes [NAN02]; or redefinitions of non-functional requirements to increase quality [NAN02] or performance [LAM98]. Figure 2-1 depicts the EVE cycle proposed in [LAM98] to define the cyclic process of requirements evolution. This cycle clearly shows the link between environment changes and changes to the system.

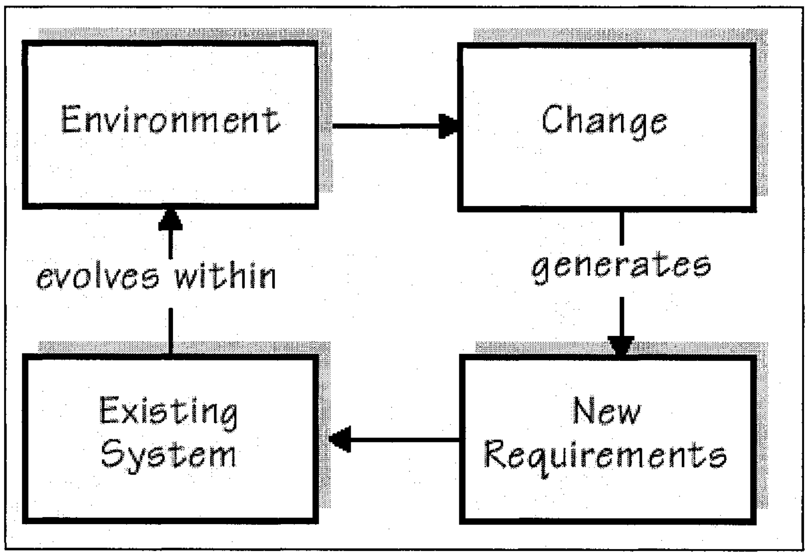


Figure 2-1: Requirements Evolution Cyclic Process

Any of the aforementioned changes to requirements, during the development cycle or after the system is installed, are a large source of risk [STR96]. In particular, those changes made after the requirements have been specified are typically the driving factors for cost and schedule overruns and may cause more defects [MAL98]. Therefore, it is important that requirement changes are managed in order to reduce the cost of implementing the change [LAM98][NAN02]. In order to manage requirements completely, Lam et al. in [LAM99] specify 4 main areas of concern that must be addressed: 1) planning the requirements change; 2) assessing the impact of the change; 3) determining the volatility of the change; and finally assessing the maintenance team's ability to handle the change [LAM99]. Furthermore, Strens et al. in [STR96] claim that both sensitivity analysis and impact analysis are needed in a pro-active approach to requirement change analysis.

2.2 Software Maintenance

The IEEE Standard Glossary of Software Engineering Terminology [IEE90] defines software maintenance as "the process of modifying a software system or component after delivery to correct faults, improve performance or other attributes, or adapt to a changed environment." This definition further describes three types of maintenance: 1) *Corrective Maintenance* that modifies the software to fix defects; 2) *Adaptive Maintenance* that consists of modifications made to keep the system usable after changed or changing environment; and 3) *Perfective Maintenance* that improves performance or maintainability.

The IEEE Standard for Software Maintenance [IEE98] clearly defines 7 key steps that are involved in maintaining software (these steps are shown in Figure 2-2 and are discussed in [PIG05]). The first step is to identify, classify and prioritize the problem. This assumes an understanding of the system to be maintained [PIG05]. This step is followed by a feasibility analysis that will, among other things, determine the impact of the change. Steps three to seven of the maintenance process involve implementing the change and ensuring the quality of the changed system.

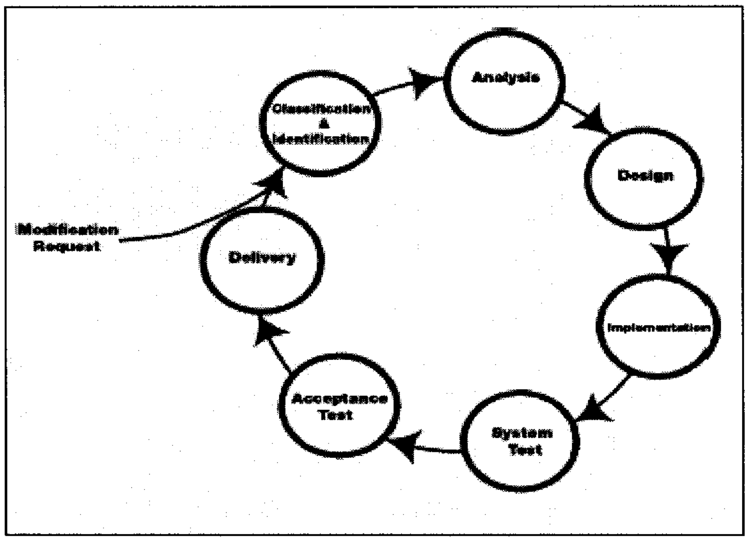


Figure 2-2: IEEE Maintenance Process

The focus of this research is on step two of the maintenance process – determination of the impact of the change. This step relies on step one that requires an understanding of the system. The next two sections will detail how software systems are understood and then how change impacts are determined.

2.3 Software Comprehension Techniques

Once it is determined that a change to the system is required, that system must first be understood before it can be modified [BEN00][BOH02]. This is generally termed software comprehension and has been defined in [RUG95] as "the process of acquiring knowledge about a computer program". This step consumes approximately 50% of time during software maintenance [NEL96]. We elaborate on some techniques that can be used in the process of software comprehension. Further details and references can be found in [RUG95].

At the most basic code level, syntactic analysis is performed by a parser to create a parse tree to show how the program code is broken down into its constituents. The abstract syntax tree (AST) removes details unrelated to program understanding leaving a meaningful tree that can be traversed and is the basis for most sophisticated program analysis tools. An example parse tree and the corresponding abstract syntax tree are depicted in Figure 2-3 [CSWPI]. Abstract syntax trees are used as the system representation in [FIE95] and [CHA98].

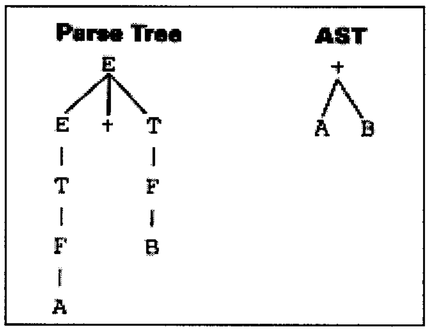


Figure 2-3: Example Parse Tree and Corresponding AST

Pattern recognition involves searching program code for instances of common programming patterns. A detailed listing of such standard patterns may be found in [GAM94]. An understanding of how the patterns work and their intent can contribute to the understanding of relationships within the source code [RIL03].

Also at the source code level, control flow analysis generally takes on two forms: intra-procedural and inter-procedural. Intra-procedural control analysis determines the order of statement execution. It creates a control flow graph from the AST that can give an abstract view of how the program executes using blocks (statements executed) and arcs that represent flows of control between the blocks. Inter-procedural analysis determines which blocks invoke others, effectively creating a call graph. Figure 2-4 shows a control flow graph where the solid arrows represent a node (labelled circles) calling another node and dashed arrow represents recursive calls [LAW03].

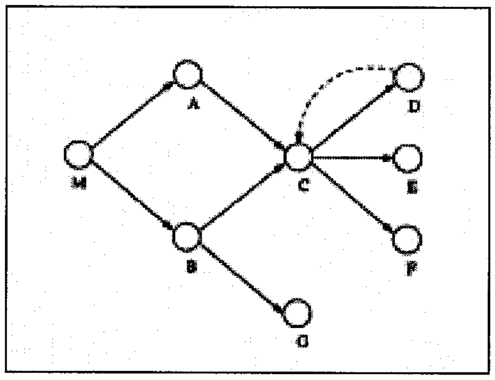


Figure 2-4: Example Call Graph

Data flow analysis is concerned with identifying how data definitions flow and where they are used in the programs. Data flow analysis focuses on describing what happens to the variable's contents during the execution of the program, not just where it is used. Program Dependence Graphs (PDGs) can be used to represent either control or data dependencies or both [BOH96]. Control flow or data flow analysis has been used as the basis for comprehension techniques in [HAR98], [WU01], [CAR99], and [ZHA01].

Dynamic analysis obtains information for comprehension purposes by systematically executing the program. This technique is generally used for performance checking, ensuring correctness in system execution, or system reengineering to support understanding [ZHA01][BOJ00][LHA05][SNE00].

Program slicing is a well researched software comprehension technique ([BEC93][RUS02][FIE95][AGR91][TIP96][HAR98][HAS04][KOR03]) that has been used to aid in understanding of both system source code and models. The next section is devoted to the explanation of slicing.

2.3.1 Program Slicing

Program slicing is program comprehension technique that was developed to reduce the amount of source code that needed to be understood; it achieves this by removing parts of the program that are not relevant to the analysis [WEI81]. Generally, a program dependence graph is formed from the program source code to show data and control dependencies. This graph is then used to extract the program slice. As an example, Figure 2-5 shows a program dependence graph (provided in [HAR98]) with data dependencies denoted by solid arrows and control dependencies by dotted arrows.

First introduced in [WEI81], the technique has been modified and customized over the years that have led to distinctions between static and dynamic slicing as well as forward and backward slicing.

Static Slicing

Static slicing uses data and control flow analysis to find the parts of the program relevant to a given input. The input to the slicing tool is a program and a variable *v* at some point of interest (called the slicing criterion). Given this input, the goal of static slicing is to find those parts of the program that are responsible for the computation of *v* at the point of interest. The output is a *slice* that consists of the parts of the program that could affect *v* for all inputs to the program. This is the technique that was introduced in [WEI81] and is currently known as backward static slicing.

Dynamic Slicing

Dynamic slicing, introduced by Korel and Laski [KOR90], creates an executable part of the program *P* with respect to some variable *v*, for some input *x*. The behaviour of the output slice is the same for input *x* as in the original program. The basis of the approach is to record an execution trace of the program for input *x* and then trace the execution backwards to collect dynamic data and control dependencies to create the PDG. Similarly, the dynamic slice could be created during run-time without requiring recording of an execution trace.

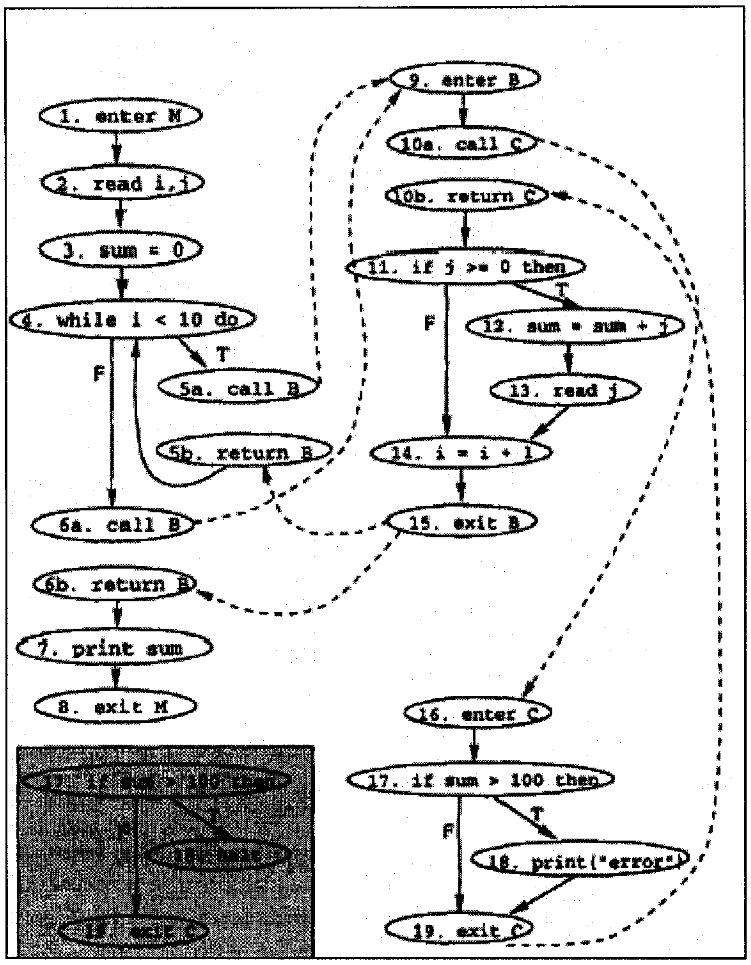


Figure 2-5: Program Dependence Graph

Forward Slicing

While a backward slice includes with it all the program statements and variables that may affect (static slicing) or actually affect (dynamic slicing) the variable *v* at some point *P*, a forward slice, distinguished by Horwitz et al. in [BRO83], is generated using the same slicing criterion as a backward slice but instead consists of all statements and predicates of the program that might be affected by the value of *v* at point *P*.

2.4 Change Impact Analysis

Once the software is understood, the proposed change must be analyzed. Since it is unlikely that the entire software will change at once, change impact analysis in some form is necessary to determine what other parts of the software may be affected if a change is implemented. Ideally, impact analysis is an iterative process that is performed as early as possible in the change cycle in order to determine the scope, the cost, and the risk of the change [QUE94] – essentially step two of the previously defined maintenance process. The actual impacts of a change in this case, are not known until the change implementation is complete [LAW03][BOH96].

An *impact* is a part of the software system determined to be affected by the change, thus requiring further inspection [BOH96]. *Impact analysis* then is "the activity of identifying the potential consequences of a change, or estimating what needs to be modified to accomplish a change" [BOH96]. We adopt this definition because it emphasizes the estimation of the impacts of a change.

Change impact analysis is a process that may involve several steps before coming to a conclusion. The IEEE Standard Glossary of Software Engineering Terminology [IEE90] does not provide a definition for impact analysis, but Standard 1219-1998: IEEE Standard for Software Maintenance [IEE98] provides a guideline for what software change impact analysis should entail during the analysis phase of software maintenance.

The standard states that impact analysis should:

- Identify potential ripple effects;
- Allow trade-offs between suggested software change approaches to be considered;
- Make use of documentation abstracted from the source code; and
- Consider the history of prior changes, both successful and unsuccessful.

These guidelines can serve as a basis for determining what should be accomplished during the impact analysis process, but they do not necessarily define a process. Bohner and Arnold in [BOH96] and [ARN93] attempt to define the sequential steps of an ideal impact analysis process. Figure 2-6 provides a visual of a generic impact analysis process discussed in [ARN93], showing the inputs and outputs of the impact analysis approach. The process begins with a change proposal that is analyzed to plan its implementation. After this, the change is scoped out by determining what parts of the system are initially affected by the change implementation chosen. Once the initial impacts of the change are determined, the potential ripple effects may be determined. These ripple effects may require further investigation while implementing the change. The accumulation of the impacts and their ripple effects provide the potential impact set of the change. This impact set can then be used to plan, predict, and accomplish that change task.

Change impact analysis aims primarily to identify these software parts that are affected by proposed changes with the goal to minimize unexpected side effects found during regression testing [BOH96][QUE94]. This includes both determining impacts within the software (*dependency analysis*) and between different software products used during the development process (*traceability analysis*).

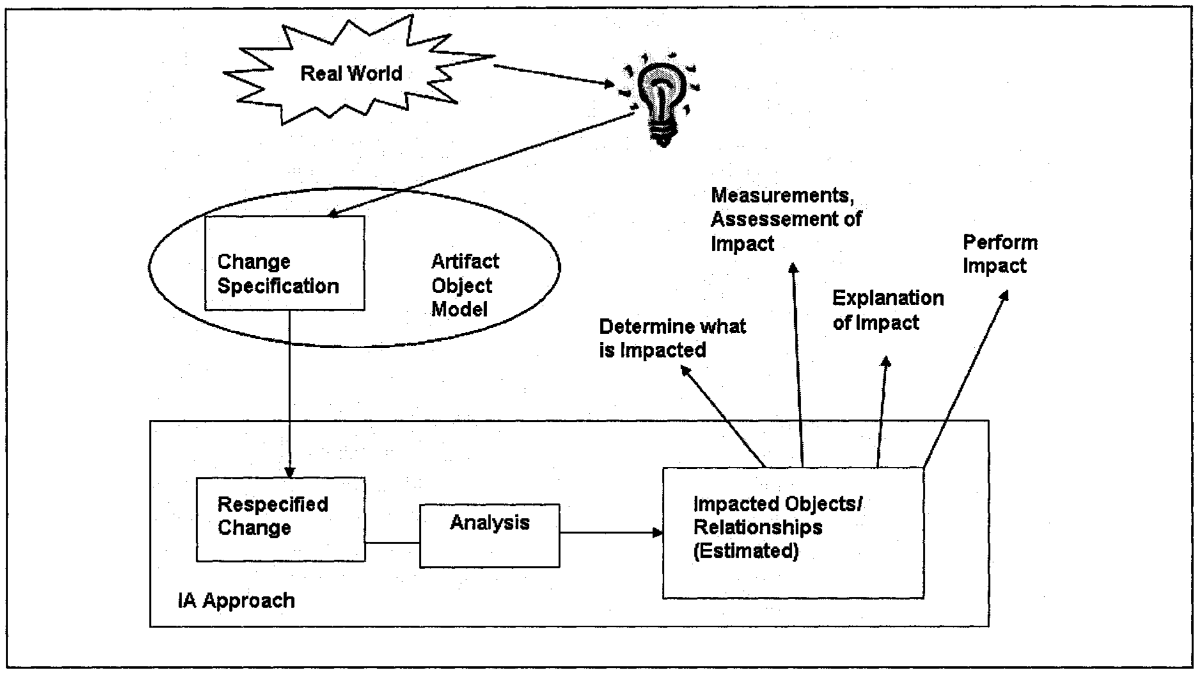


Figure 2-6: Generic Impact Analysis Process

2.4.1 Traceability Analysis

Traceability analysis is one of the two major areas of change impact analysis. Requirements traceability focuses on determining if the requirement has been implemented and if so, where? Once the requirement has been traced to the desired level of abstraction, a more specific impact analysis technique can be used to determine how the change will affect the system [BOH96]. It is said that requirements should be traceable to different design products (*vertical traceability*), within these design products (*horizontal traceability*), and also to the code that implements them [STR96]. Some say requirements traceability provides critical support for managing change in an evolving software system [SET04].

Software life cycle objects (SLOs) generated or modified during the development process are traceable if they provide the ability to associate the information between them [BOH96]. This is important because each software life cycle object, while essentially detailing the same system, provides a different view of the system at varying levels of abstraction. Thus, *traceability analysis* involves "examining dependency relationships among all types of SLOs" [BOH96].

Traceability information relies heavily on software documentation [BOH96], including but not limited to requirements specifications, designs, and user documentation [LAM98b]. The relationships between the various documentations can be illustrated in a graph structure. An example of such a graph is shown in Figure 2-7 [LAM98b] where the relationships between different artefacts (rectangles) within the same document (CSS Requirements Document) and then between different documents are labelled with specific identifiers (d*i*). This benefits impact analysis by providing information on what other life cycle products are affected by the proposed changes, allowing effective navigation for more detailed impact analysis [LAM98b].

There exists a body of research ([STR96][SET04][KNE03]) on approaches to effectively perform traceability analysis for change impact analysis. The mentioned references focus on how to trace requirements to design documents and models.

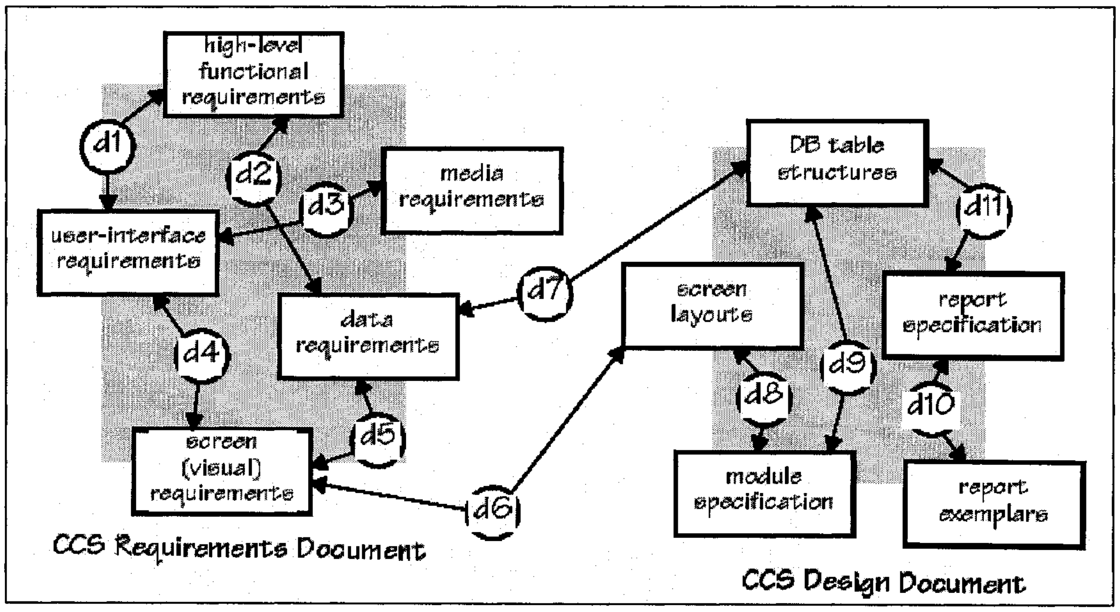


Figure 2-7: The Documentation Architecture

2.4.2 Dependency Analysis

Dependency analysis represents the second major branch of impact analysis and is used for determining the impact of changes within the system. *Dependency analysis* involves "examining detailed dependency relationships among program entities" [BOH96]. In [BOH96], it is stated that this type of analysis is narrowly focused, providing detailed evaluations at the code level but not at higher levels of abstraction. At the time of their assessment this may have been true, but over the years dependency analysis approaches have been developed for higher levels of system abstraction [KOR04][ZHA02].

A general definition for a *dependency* provided in [COX01] is "relations, D, between some number of entities wherein a change to one of the entities implies a potential change to others". Furthermore, this research also defines directed dependencies and categorizes them into unidirectional and bidirectional: a *unidirectional* dependency d(A,B) exists if node A depends on node B, while a *bidirectional* dependency exists if A depends on B and B depends on A (as in a recursive call). This work also details a set of attributes that are common to all dependencies. Some of these include sensitivity, stability, importance, and impact. Impact meaning in what ways is the dependent entity's functionality compromised by failure of this dependency.

Dependency analysis can be performed manually or automatically using techniques like program slicing, control- and data-flow analysis, test-coverage analysis, and cross referencing to evaluate the data, control, and component dependencies between system entities [BOH96]. Additionally, dependency analysis may be performed on static system information, such as a class diagram, or on dynamic information, such as actual execution traces of the system.

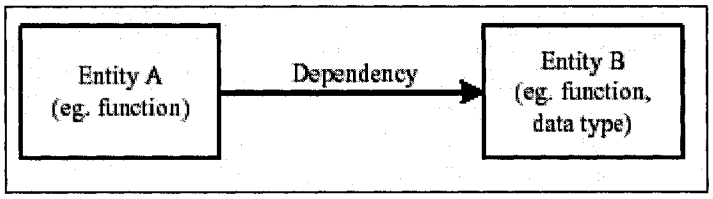


Figure 2-8: Schema for a Traditional Dependency Graph

Dependency information about the system under analysis is usually stored in a dependency graph that includes system entities and the respective relationships between them. The schema for this is shown in Figure 2-8 [HAS03]. Bohner et al. in [BOH96] define three main categories of system dependencies: data dependencies, which are depicted in data dependency graphs, show relationships among system entities that define and use data; control dependencies are "relationships among program statements that control program execution"; and component dependencies are "general relationships among source code components such as modules files, and test runs".

Dependencies and dependency graphs have also been defined for abstraction levels higher than that of source code. Most relevant to our research are the scenario dependencies that have been defined. The next section is dedicated to their detailed discussion.

2.4.2.1 Scenario Dependency Analysis

Scenarios, defined in [RADZI] and [BRE00], have been used to describe the execution sequences of system functionality for a long time because they provide a good compromise between informal use cases and formal designs [AMY01c]. The use of scenarios allows advantages such as: validation of requirements and the comparison of requirement alternatives [KAZ96]; reduction of complexity in requirements understanding [WEI98]; and aiding in requirements agreement among different stakeholders [WEI98].

Scenario dependencies have been detailed in [TSA03], [PAU01], [TSA01], [BAI02], [BOR01] and [BRE00]. Each describes a slightly different set of scenario relationships and/or dependencies as they relate to their own research area. We summarize these research findings here because scenarios are vital to this research.

Functional Dependency

Bai et al. in [BAI02] describe how to define scenario groups to create test scenarios. They discuss working from the requirements and decomposing each functional feature in order to define the groups that the scenarios may fit into. They conclude that functional dependency among scenarios exists if the scenarios belong to the same group that represents common system functionality. Figure 2-9 [BAI02] shows an example of the scenario model in which the functions of a Banking System are decomposed progressively into multiple levels of scenario groups, scenarios, and sub-scenarios.

A predecessor of this research is [TSA01] where discussion pertains to arranging thin threads hierarchically for the purposes of generating test cases, performing risk analysis, and accomplishing ripple effect analysis. In their approach, a thin thread tree is created and the root of the tree is the system under test. Each branch of the tree represents a group of scenarios related by functionality, while each leaf represents a concrete scenario. This, they claim, effectively creates a functional decomposition of the system.

Containment Dependency

In [PAU01] and [BAI02] a "path-contained relationship" for a scenario execution path is defined. For two scenarios A and B, if the complete path of A is part of the path of B, then A is contained in B and B depends on A. Similarly, in [BRE00] a subset relationship is discussed. They define a scenario to be a subset of another if one scenario shares the context of the other scenario.

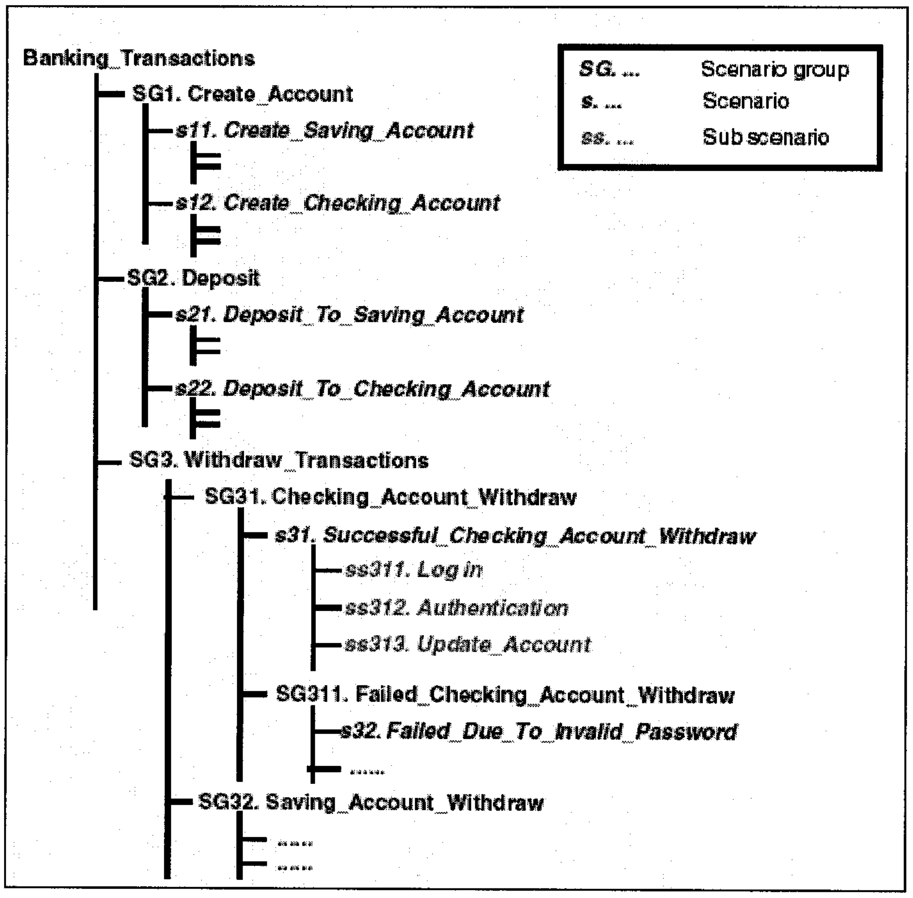

Banking_Transactions
SG1. Create_Account
s11. Create_Saving_Account
s12. Create_Checking_Account
SG2. Deposit
s21. Deposit_To_Saving_Account
s22. Deposit_To_Checking_Account
SG3. Withdraw_Transactions
SG31. Checking_Account_Withdraw
s31. Successful_Checking_Account_Withdraw
ss311. Log in
ss312. Authentication
ss313. Update_Account
SG311. Failed_Checking_Account_Withdraw
s32. Failed_Due_To_Invalid_Password
......
SG32. Saving_Account_Withdraw
......
......
SG. ... Scenario group
s. ... Scenario
ss. ... Sub scenario

Figure 2-9: Example of Modeling Scenarios

Condition Dependency

In [BAI02], scenarios are said to be condition dependent if they are affected by the same conditions. Comparably, [PAU01] specifies that scenarios are condition dependent if they share pre- and/ or post-conditions. Additionally, in these works, the relationships that exist between the conditions themselves are defined. Some of the relationships defined for conditions include independent - two conditions can happen irrespective of the other, mutually exclusive – two conditions cannot exist at the same time, and related – the two conditions are used in the same thread.

Execution Dependency

Execution dependency is discussed by Paul in [PAU01] and Tsai et al. in [BAI02] within the context of generating test scenarios. They briefly define execution dependency to exist between scenarios that interact through their execution paths and share common software components such as code modules or interfaces. In [PAU01], Paul claims that scenarios may be dependent on each other in 3 ways: identical paths, covered paths, and crossing path, but provide no further definition of these dependencies.

2.4.3 Ripple Effect Analysis

Ripple effect analysis is a sub-process within impact analysis. A *ripple effect* is "the effect caused by making a small change to a system which affects many other parts of a system" [BOH96][ARN93]. This definition is not limited to the source code level and can be extended to include design and specification models as well [WAN96].

Figure 2-10 visualizes the iterative generic ripple effect analysis process defined in [WAN96] that includes: 1) Making the initial change; 2) Identify potentially affected areas due to that change; 3) Determine which of these areas needs further changes; and 4) Determine how to make that change.

Ripple effect analysis (REA) focuses on determining what parts of the system may be affected by a change. Using a chosen approach, the location of the initial change is identified and the effects of that change are recorded to create the impact set. How the impact set is identified depends on the approach itself that either uses dependency analysis or traceability analysis as its basis.

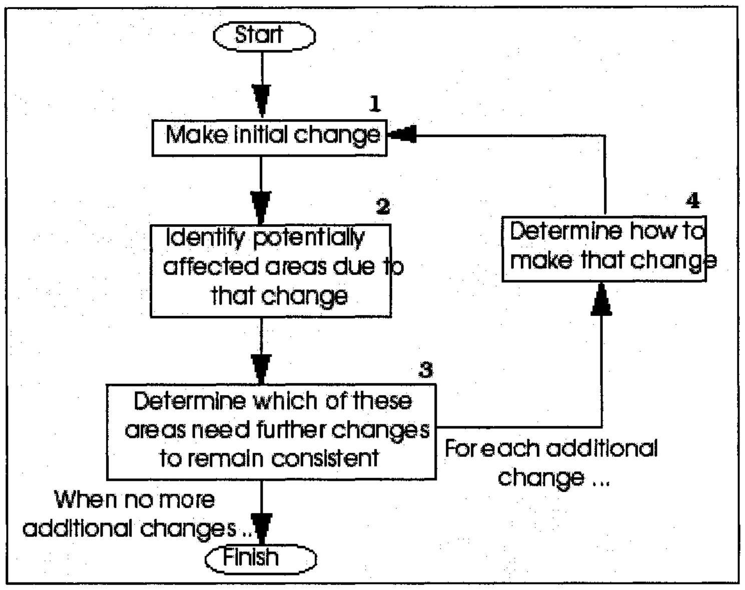

Start
1
Make initial change
2
Identify potentially affected areas due to that change
3
Determine which of these areas need further changes to remain consistent
For each additional change ...
4
Determine how to make that change
When no more additional changes ..
Finish

Figure 2-10: Generic Ripple Effect Analysis Process

To determine the impact associated with a change, the system model must be searched to identify relationships among system entities. In [BOH96] and [BOH02], the following summary of several search algorithms which form the basis of many ripple effect analysis approaches is provided. They state that some search algorithms may be semantically guided where the impacts are obtained from a predetermined semantic network of objects; others may be heuristically guided in which a predetermined set of rules suggest possible paths that may contain impacts. Stochastically guided searches use a given situation as the basis for determining the probabilities of impact. Hybrid searches combine the aforementioned algorithms, while unguided searches attempt to find impacts in a brute force manner. Transitive closure used on call graphs is used as the basis for many of these algorithms to determine the impact sets of a change [LEE00][BRI03][LAW03].

2.4.4 Change Impact Analysis Approaches

A large amount of research exists that details change impact approaches that range from analysis at the source code level up to the system architecture level. Queille et al. in [QUE94] identify three issues that need to be resolved when performing impact analysis: First, the system must be represented to show the system objects and the relationships between these objects. A dependency graph is an example of such a system representation. Next, the data to populate this model must be collected. Finally, a method of tracing through this representation, such as a call graph to determine the impact set must be created.

From our research, we ascertained that most impact analysis approaches use either dependency or traceability information as their basis for impact determination. In the following section we provide a survey of some existing approaches that use dependency analysis.

2.4.4.1 Source Code-based Approaches

Source-code change impact analysis uses the source code of the system as its basis for determining relationships within the system. These approaches have the advantage of being very accurate in the analysis since they identify impacts in the final product, however, they have the disadvantage of being extremely time consuming, limited in scope, and they require implementation of the change before the impact can be determined [BRI03].

Kung et al. in [KUN94] discusses the type of code changes that can occur in object-oriented classes (class changes, method changes, etc.) and provide a solution to determine their impacts. Further, Lee et al. in [LEE00] define object-oriented data dependency graphs (OODDG) that emphasize data relationships relevant to object-oriented systems between such items as classes, class members, and constants. The OODDG actually consists of three graphs specific to method, inter-method, and inter-classes relationships in the system. This work provides algorithms to evaluate changes and metrics to quantitatively evaluate change impacts.

Similarly, Kim et al. in [KIM99] investigate the impact of changes to object-oriented software in distributed environments and propose a distributed program dependency graph (DPDG) that shows relationships that include data, design documents, servers, and classes. They argue that conventional change impact analysis is hard to apply to distributed systems because their characteristics are limited to centralized system environments.

In [BLA01], Black proposes using a developed approximation algorithm to completely automate the computation of ripple effects for C programs. In [REN04], Ren et al. detail their tool *Chianti* that analyzes two versions of an application and decomposes their differences into a set of atomic changes.

Other methodologies focus on determining impacts from a dynamic representation of the system where a representation is created based on execution traces of the system. Law and Rothernel in [LAW03] propose a novel approach for impact analysis using "whole path profiling" based on dependency analysis at the procedure level. It uses low cost instrumentation to retrieve dynamic information about system execution and then builds a representation from that collected information. This approach incorporates call-order and call-returns in the impact set. Similarly, work in [ORS03] take this one step further and provides algorithms for impact analysis using instrumentation information from deployed software. They claim that this provides better results for impact sets than using fabricated system executions.

2.4.4.2 Slicing Based Approaches

Various implementations of program slicing algorithms exist for all the previously mentioned categories of slicing (see section 2.3). With respect to impact analysis, program slicing can be used to determine the potential effects of making a change [WAN96]. Static slicing determines dependencies for all program inputs while dynamic slicing searches the dependency graph for dependency based on the input given [LAW03]. The following discussion summarizes some of the major approaches that make use of slicing algorithms to compute impact sets.

In [TON03] a novel approach is introduced that combines slicing and concept analysis, called "Concept Lattice of Decomposition Slices". The authors claim that the approach is an extension of the decomposition slice graph in that the graph is obtained from concept analysis and it can be used to assess change impacts at given program points. Similarly, in [CHE96], researchers discuss how to create a dependency graph from a C++ program (termed "C++ Program Dependence Graph") to capture declaration, message, and class dependencies relevant to object-oriented systems. Impacts on this representation are determined by the application of developed C++ specific slicing algorithms to slice classes, messages, and programs.

Some other works have applied slicing algorithms to system models instead of a dependency graph created from the source code. For example, in [KOR03], an approach is presented that applies slicing to extended finite state machines (ESFM) in effort to analyze the system with respect to a particular functionality. They create an ESFM dependency graph using the data and control dependencies of the model and suggested that this approach can help in understanding how the model will interact with changes made to the system. Similarly, in [HEI98], slicing is applied to hierarchical state machines with the goal of performing impact analysis on the Requirement State Machine Language (RSML).

2.4.4.3 Model-based Approaches

There is a growing trend towards model-driven development. It is creating a need to perform change impact analysis on a representation of the system that is at a higher level of abstraction than source code [SET04]. The benefit of these model-based approaches is their ability to determine impacts without implementation of the change, although they may provide less precise results [BRI03].

In [BRI03], Briand et al. propose a static, UML model-based approach to impact analysis that can be applied before changes are implemented to help in the planning process. They first check for consistency between the diagrams and the implemented system. Then, changes between the two models (the original and the changed) are identified using a “change taxonomy” to associate their formally defined (using Object Constraint Language) impact rules with types of changes. Finally, impacts are determined using the defined impact rules and transitive closure.

A dynamic impact analysis approach that supports understanding the effects of changes on an ESFM is presented in [KOR04]. Using both the original model and the modified one, they provide an algorithm for automatically determining the differences between the two models. The algorithm identifies parts of the model that may be affected by the change using model-based dependency analysis.

Finally, Zhao et al. in [ZHA02] propose an approach for change impact analysis that is at the architectural level. Based on an architectural slicing and chopping technique, they use a formal description of the architecture of the system modeled in WRIGHT Architectural Description Language (ADL) and automate the determination of change impact using architectural slicing and a new technique called architectural chopping.

2.4.5 Change Impact Approaches Discussion

In an attempt to provide software maintainers with a way of assessing impact analysis (IA) approaches for effectiveness, [ARN93] proposes a framework for comparing different impact analysis approaches. This framework focuses on three main areas that can be used to compare and evaluate various impact analysis approaches regardless of the abstraction level. The areas can be summarized as follows:

- IA Application examines how the IA approach is used to accomplish impact analysis by assessing the inputs and outputs of the approach.
- IA Parts is concerned with the functional parts of the IA approach – the internal model used to represent the system and its dependencies, the rules and semantics of relationships between the model entities, and the algorithms for determining the impact set.
- IA Effectiveness is concerned with how well the approach accomplishes impact analysis. The estimated impact set is compared to the actual impact set to assess its effectiveness.

These general guidelines of how impact analysis can be accomplished provide us with an understanding of what the goals of an approach should be. We highlight the main features of the distinct approaches discussed in Section 2.4.4 in Table 2-1 using some of the sections of the mentioned framework as headings.

As discussed in Section 2.4, change impact analysis at each level of abstraction has its advantages. These include impact set accuracy at the source code level and ability to scope the impact of the change at the design model level. We extend that discussion and outline some limitations that exist in impact analysis techniques at both the source code and model levels.

Reference	IA Application	IA Parts		
	Object Domain	Internal Model	Impact Model	Impact Tracing Algorithm
[KIM99]	Distributed object-oriented source code objects	methods, data members, classes, servers, documents	Distributed Program Dependency Graph (DPDG) with data and control dependencies	semantically guided searching
[LEE00]	object-oriented source code objects	data members, classes, methods	Object-oriented system dependency graph with data and control dependencies	transitive closure on graph
[LAW03]	source code methods	executed class methods	whole path directed acyclic graph	forward and backward searching guided by execution traces
[TON03]	source code objects	statements, data members	concept lattice of decomposition slices	forward slicing on lattice nodes
[CHE96]	object-oriented source code objects	methods, classes, objects	C++ dependency graph with message, class, and declaration dependencies	forward and backward slicing
[KOR04]	EFSM system model	system states and transitions	data and control dependence graph	transitive closure on graph
[BRI03]	UML conceptual and class models	classes, interfaces, sequence messages, state machines	UML models (sequence, state or class diagrams)	transitive closure
[ZHA02]	system architecture objects	architecture components	WRIGHT architectural structure	architectural backward and forward slicing and chopping

Table 2-1: Change Impact Approaches Comparison

The mentioned source code-based approaches all require the source code to understand and identify impacts of a change. As such, they return impact sets that are very specific and possibly too detailed for assessing change impacts in large systems. Further, source code based approaches do not support scoping the impact of a change; they require a change implementation before impacts can be identified.

Conversely, model-based impact analysis approaches require a representation of the system to identify change impacts. When impact analysis is performed on an abstraction of the system, the assumption is that the model is up to date and consistent with the code. Although this can be verified using traceability tools [BRI03], there is still the possibility of inaccurate representation. Furthermore, not all developed systems are modelled using a design notation, thus change impact analysis at this level of abstraction may require the extra step of reverse engineering the system to the required model.

From the perspective of this research impact analysis approaches (at any level of abstraction) can be categorized into two main categories termed *static* and *dynamic*, as defined in [BOH96]. Static impact analysis analyzes the source code structure without executing the code. Since it is independent of any particular program execution, static analysis considers all potential system executions and all entities that are related to the changed entity are added to the impact set. The disadvantage of this method is that a large, possibly inaccurate impact set may be returned [BOH96].

Dynamic impact analysis, on the contrary, relies on an execution trace of the system [BOH96]. Dynamic analysis has the benefit of returning an impact set that is determined from tracing through an execution of the system, implying that the results returned are almost certainly impacted [LAW03]. The disadvantage is that the execution trace(s) used may not account for all the possible executions that involve the changed entity. This results in an impact set that may not contain all the impacted parts of the system [BOH96].

This concludes the first required background of our research study. We now focus on the second aspect that we need to have an understanding about in order to fully achieve our research goals.

3. Use Case Maps

Use Case Maps (UCMs) are a high-level design notation developed by R.J. Buhr and his colleagues [BUH96b]. It is intended for use at the requirements specification level to help in the reasoning of "large-grained behaviour patterns in systems" [BUH96b]. The goal of the notation is to link system behaviour and structure in a lightweight and visual way.

This chapter describes UCMs including its features, its notation, and how it defines and makes use of scenarios. We also provide a brief overview of the existing tool that supports the creation of UCM models before concluding with UCM scenario and formal definitions that have been developed.

3.1 UCM Features

UCMs have been developed to help provide an understanding of the scope of a system that is defined as a set of collaborating components [BUH96b]. For UCMs, a system includes both software and non-software entities. The intended application of UCMs is for the specification of systems whose behaviour can be detailed in terms of paths with minimal concurrency [BUH96b]. As an example of such a system, Figure 3-1 depicts a simple point-to-point fax system UCM. The scenarios of the system are defined by the paths that go from one point to another with no concurrent paths.

According to the creators of UCMs [BUH96b], the advantages that UCMs provide include:

- Ability to reason about system behaviour
- Ability to show intended coupling between large-grained behaviour patterns
- Bridging the gap between requirements and detailed designs
- Provide a behavioural framework for reasoning about architectural issues
- Document high level decisions
- Combine real-time and object-oriented issues in one design.

The following subsections elaborate on how UCMs have been shown to provide these advantages.

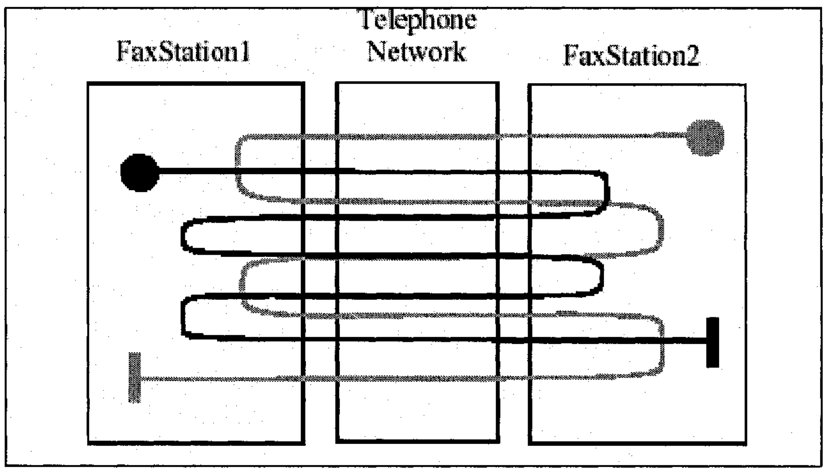


Figure 3-1: UCM Fax Machine Example

3.1.1 Ability to Reason about System Behaviour

The UCM notation is said to be lightweight because it is at the architectural level, aiding in high-level understanding, designing, and reengineering that require this attribution of behaviour to architecture [BUH96].

Scenarios have been shown to be useful for modeling behaviour. However, the common UML notation for expressing scenarios, message sequence charts, reduces their scalability with their dependence on detail [BUH96b]. The detailed information in these diagrams that include the specification of operations, assembly, and inheritance lowers the level of abstraction below a level that is appropriate for architectural specification. UCMs, on the other hand, describe the behaviour at the system architecture level without including message passing details and therefore allows for high-level reasoning of the architecture [BUH96].

3.1.2 Ability to Show Intended Coupling Between Large-Grained Behaviour Patterns

UCMs show the interaction of entities for a single use case in a map-like diagram. Each UCM shows several scenarios of the use case together in one map with paths showing the progression of each scenario within the system [BUH99].

Scenarios in the UCM sense correspond to something more abstract than the traditional message sequence charts used in UML [MIG01]. UCM scenarios are continuous paths which are superimposed on organizational structures whose sequence is determined by component responsibilities.

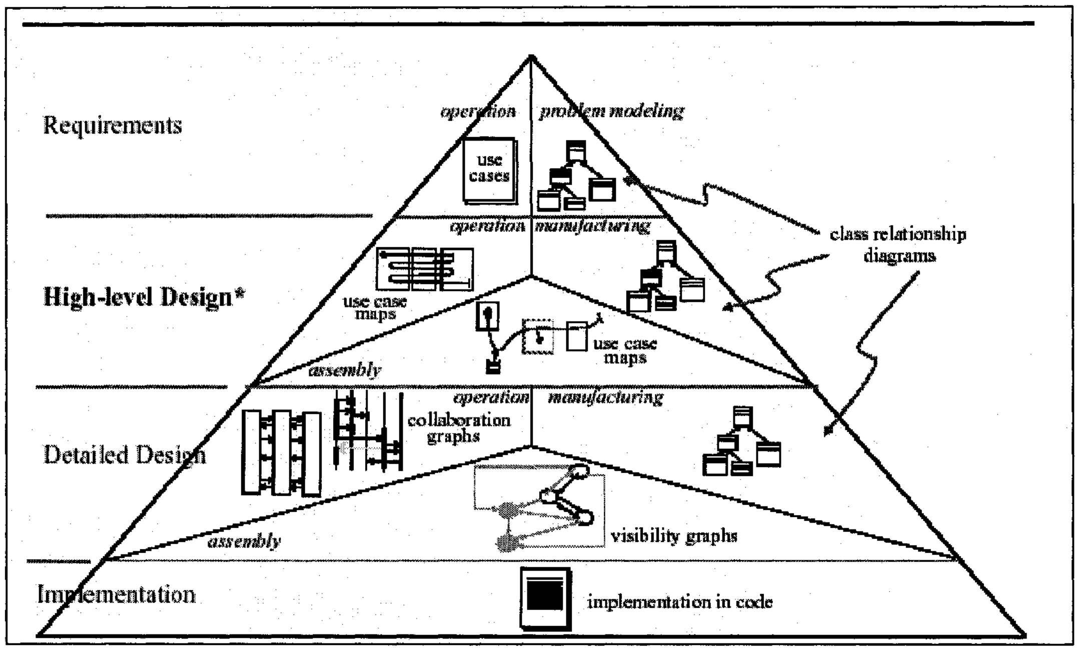


Figure 3-2: UCMs in Context of Other Models

3.1.3 Bridging the Gap Between Requirements and Detailed Designs

UCMs are developed from use cases that are formed directly from the requirements. This raises UCMs above the level of message passing diagrams, allowing the design model to be scalable for the specification of large systems [AMY01b].

The goal of the introduction of UCMs was not to replace existing design models, but rather complement them by filling the conceptual gap that currently exists when transitioning from requirements to design [AMY01b][MIG01]. Figure 3-2 shows UCMs in the context of the development cycle, while [BUH96b] describes how UCMs complement and are compatible with other notations like Object Oriented Software Engineering (OOSE), Real-time Object Oriented Modeling (ROOM), and Object Modeling Technique (OMT).

These complementary aspects of UCMs were further extended by Amyot et al. in [MIG01] with their approach for converting UCMs into detailed UML message sequence charts. An example output of their approach is shown in Figure 3-3. According to [AMY01] the UCM notation is simple, intuitive, and has a low learning curve so using it in conjunction with other models is a benefit.

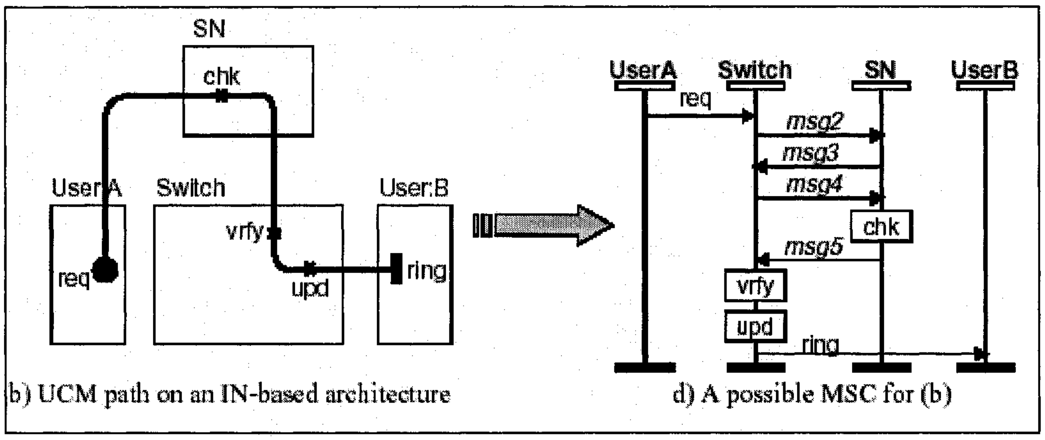


Figure 3-3: UCM Transformation to MSC

3.1.4 Provide a Behavioural Framework for Reasoning about Architectural Issues

UCMs are powerful for expressing and understanding important macroscopic aspects of behaviour in relation to architecture. Such aspects are very difficult to deal with at the level of message passing [BUH96]. In [BUH99] the following benefits of the abstraction level provided by UCMs are outlined:

- Path interactions in concurrent systems are visible at a glance
- Performance becomes a property of paths, rather than a non-functional property of a whole system, as it is usually considered to be
- Large scale dynamic situations can be made visible at a glance
- Paths directly indicate how the architecture satisfies use case requirements

The fact that scenario definitions are the basis for the visualization means that UCMs show the dynamic functionality of the system. This is a benefit when attempting to model dynamic systems where scenarios and structures may change at run-time [AMY01].

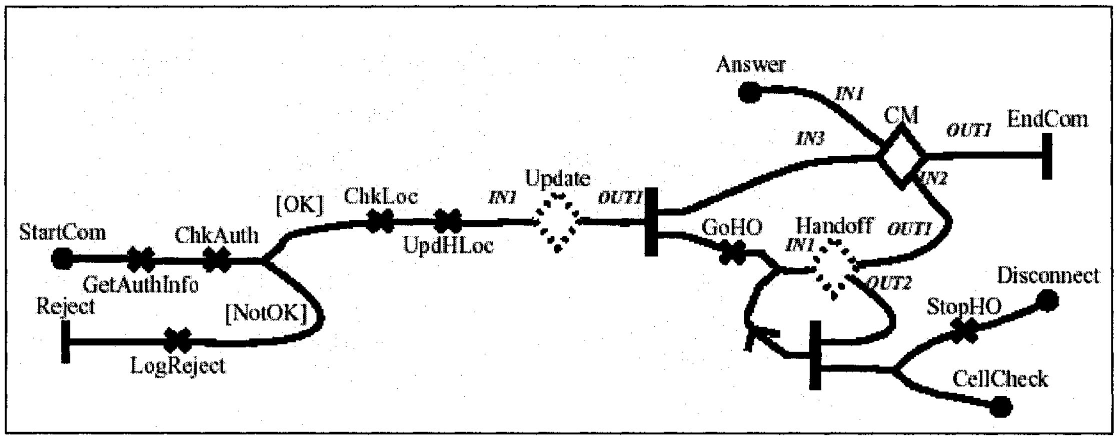


Figure 3-4: Root UCM for Simplified Wireless System

UCMs are intended to be used to guide system architectural design [BRU00][AMY01] and the notation has been used in the describing the design of a wide range of systems including: operating systems, agent systems, Wireless ATMs, Intelligent Networks, and GPRS ([UCM] provides the necessary references). As an example, Figure 3-4 shows a proposed UCM for a wireless system [AMY03]. As well, UCMs have been used to help capture, understand, analyze, reuse, and change high level behaviour patterns [BUH95][NAK00].

3.1.5 Document High Level Decisions

UCMs can also be used to document the high-level decision [BUH96b]. Proof of their usefulness in documenting design decisions is the fact that UCMs is part of the set of User Requirements Notation (URN) standards proposed to the International Telecommunications Union (ITU-T) [AMY03] for describing functional requirements as causal scenarios.

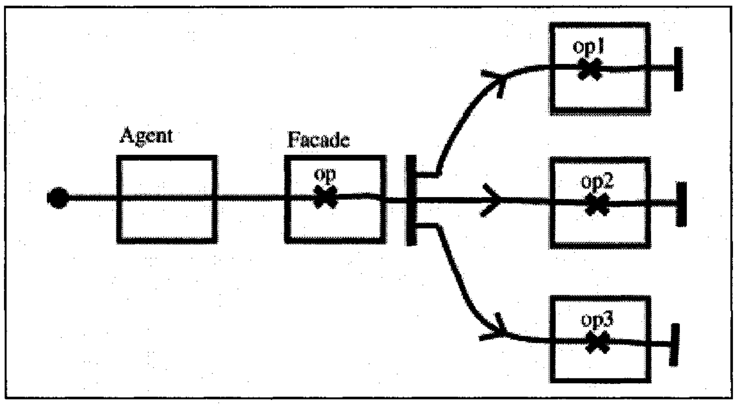


Figure 3-5: UCM Showing Concurrent Paths

3.1.6 Combine Real-Time and Object-Oriented Issues in One Design

UCMs incorporate several different aspects into their notation. Real-time issues such as concurrency and parallelism, modeled using UCM AND-joins and forks (an AND-fork is visible in Figure 3-5 [BIL04]), are included and represented in the notation [BUH99].

Furthermore, scenario paths assume that real time is required to progress from start to finish [BUH96b]. Object-oriented issues are included in the model through the support of different instances of the same component that are differentiated using component role names. Also polymorphism can be modeled by allowing the context to provide different meanings for responsibilities with the same name.

We have discussed in detail the benefits that UCMs provide in an effort to substantiate the claims of its usefulness. In the next section we focus on the notational aspects of UCMs.

3.2 The UCM Notation

The UCM notation contains three main elements that are the basis for all maps: *paths* that trace scenarios through the components of the system; *responsibilities* that link paths to components; and *components* that perform responsibilities [BUH96b]. Paths, responsibility points, and component boxes, labelled in Figure 3-6 [UCM], are all formal elements of the notation. These formal elements all have name labels to aid in the understanding process. According to Buhr in [BUH99], this lack of formality in the details is what makes the notation lightweight.

3.2.1 UCM Scenario Paths

UCM scenario paths begin at the *start points* (represented by a filled circle) that indicate triggering events and/or pre-conditions for the commencement of the scenario. Scenario *paths* (wiggly lines) are progressed to *end points* (short vertical bars) that represent terminating events or post-conditions of the scenario execution. In between, component responsibilities are executed showing how the initial stimulus affects the system. System scenario paths are end-to-end as they show complete system behaviour [BUH96b].

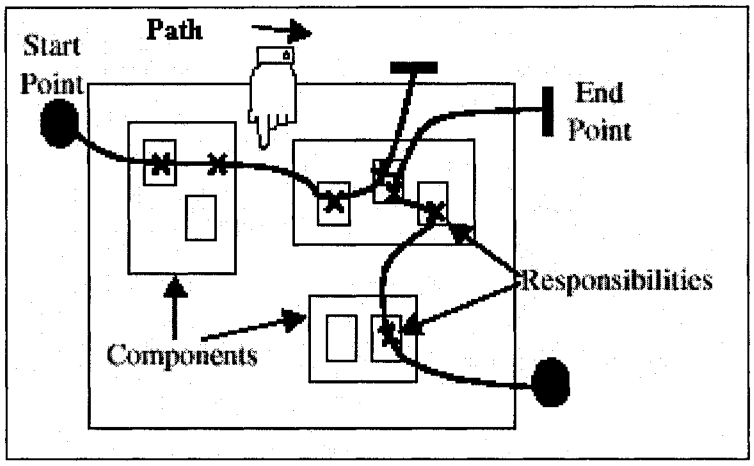


Figure 3-6: UCM Main Notational Elements

A path is defined by the conditions that govern its execution. This forms the context of the scenario that consists of the state of the system at the time the scenario is executed and the data that triggers the start points. Scenario variables (called *global variables)* have been introduced to represent system conditions and control the choice of alternative paths [MIG01].

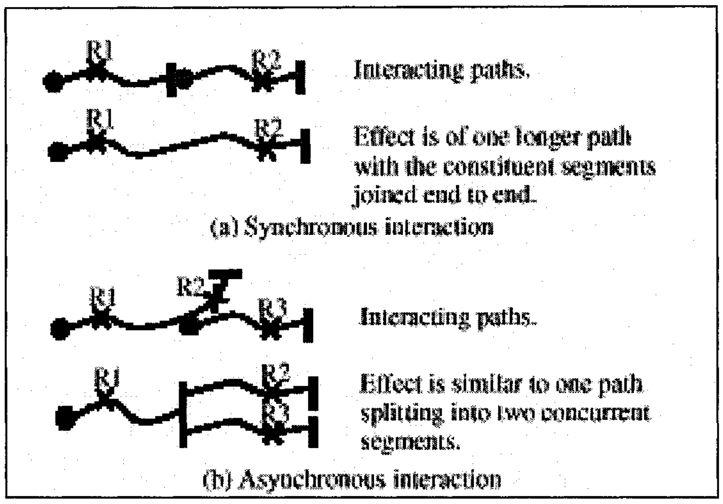

R1
R2
Interacting paths.
R1
R2
Effect is of one longer path with the constituent segments joined end to end.
(a) Synchronous interaction
R1
R2
R3
Interacting paths.
R1
R2
R3
Effect is similar to one path splitting into two concurrent segments.
(b) Asynchronous interaction

Figure 3-7: Interacting Paths

Scenario paths do not occur in isolation - a benefit of UCMs is the ability to show various logical interactions of different scenario paths [BUH96b]. Figure 3-7 depicts the different types of interactions that may occur between scenario paths [UCM].

3.2.2 UCM Responsibilities

As previously mentioned, UCM paths execute responsibilities that are bound by components. Thus, responsibilities link paths and components. UCM responsibilities are defined as events or tasks that the component must be able to perform. It is important to note that UCMs can be created without any component structure in which case responsibilities are not confined. The interactions between responsibilities form scenario paths, however responsibility relationships are not static; the cause-effect relationship between two responsibilities is determined by the path that creates the context [BUH96b].

3.2.3 UCM Components

UCM components are system objects or processes that are self-contained with internal states and interfaces [BUH96b]. Components can either be *fixed* (solid outlines) or *slots* (dotted outlines). Fixed components are components that are persistent in the architecture of the application, while slots indicate where dynamic components (along with their dynamic responsibilities) are created, moved (change visibility), or destroyed. A *pool* holds the dynamic components that are available to be moved into the slot where a single component is active [BUH96][UCM].

As an example, Figure 3-8, which can be found in [BUH96b], shows a model-view-control (MVC) design that makes use of the two component types. The map shows an MVC triad where the *model* component is a slot to allow different drawings (from the pool of drawings) to be in its position at different times.

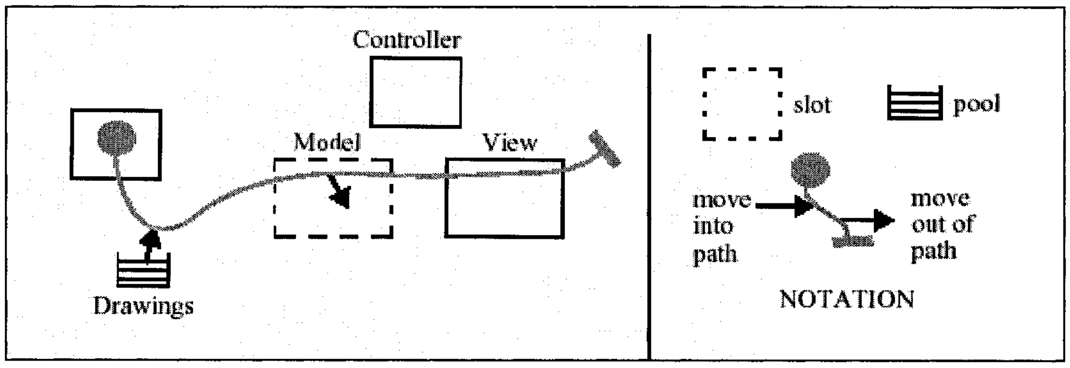


Figure 3-8: Example of Structural Dynamics Expressed with a UCM

If components exist within the UCM design, then the responsibilities displayed in each component are bound by that component. When a path links two components together, it implies that during that scenario execution some type of interaction exists between the two connected responsibilities. Those pairs of components that bind the connected responsibilities may actually involve several interactions in order to complete the execution, thus the responsibilities are considered as shared [BUH96].

3.2.4 UCM Stubs

UCMs provide the ability to layer the diagrams to show different levels of abstraction. A *stub* is a placeholder for the insertion of a sub-map, called a *plug-in*. Stubs can be static or dynamic and are bound to the parent map by *input* and *output* segments of the root map to the start and end points, respectively, of the sub-map. This ensures the continuity of the scenario sequence.

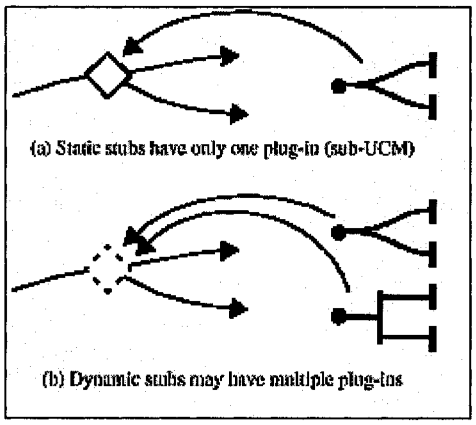

(a) Static stubs have only one plug-in (sub-UCM)
(b) Dynamic stubs may have multiple plug-ins

Figure 3-9: Stubs and Plug-ins

Static stubs only have one *plug-in* that can be inserted into the map. They are represented by solid diamonds in the location of the root map where they are to be inserted. Contrarily, dynamic stubs (represented by dotted diamonds) have multiple *plug-ins* that are chosen dynamically at run-time based on their selection policy and the system state [AMY03b]. Figure 3-9 [UCM] shows the notational difference between static and dynamic stubs.

3.2.5 UCM Concurrency

Concurrency can be shown in UCMs by allowing more than one scenario path to be traced at the same time [BUH96]. The UCM elements that show concurrency include *AND-Forks* and *AND-Joins* (Figure 3-10 [UCM]). Alternative paths that may be chosen depending on *guard conditions* are shown on UCM using *OR-Forks* and *OR-Joins* (Figure 3-11 [UCM]). The joins effectively show merging paths, while the forks show alternatives that can be taken depending on the preconditions of the scenario. OR-Forks have no logic attached to them and OR-joins are not synchronized; they are simply the result of several scenario paths being contained in one UCM [BUH96b].

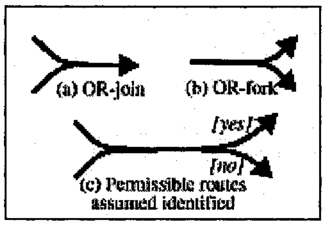


Figure 3-10: OR-forks and OR-joins

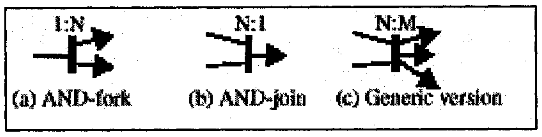


Figure 3-11: AND-forks and AND-joins

This section introduced the major elements of the UCM notation. Several notational details have been omitted from this overview in the interest of space and relevance. For a complete overview of the subtleties of the UCM notation, readers are advised to consult [UCM].

3.3 UCM Navigator

In an effort to support the use of UCMs, a tool has been developed to facilitate their creation. UCM Navigator (UCMNav), developed at Carleton University, aids in the development of properly structured UCMs and is freely available from the UCM website [UCM].

UCMNav provides basic features for creating UCMs that are syntactically correct by ensuring proper bindings of plug-ins to stubs and of responsibilities to components and paths. Figure 3-12 displays the user interface of UCMNav.

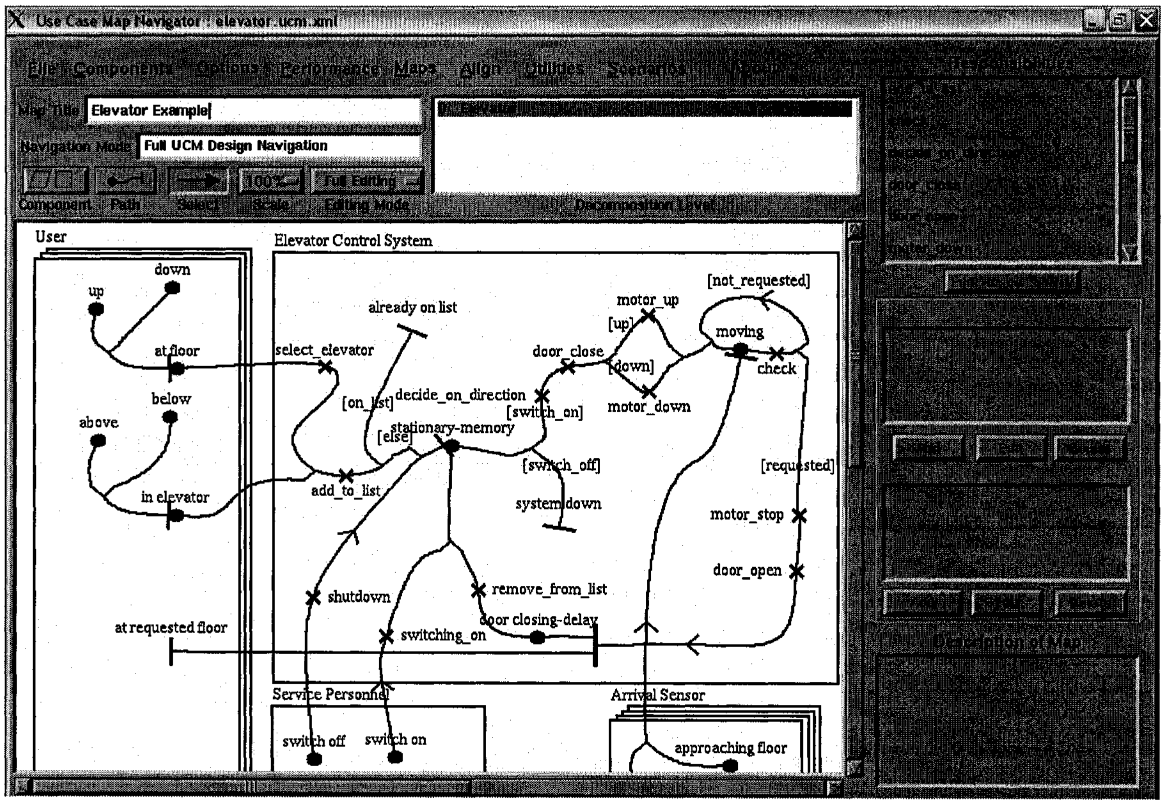


Figure 3-12: UCMNav User Interface

The user interface allows the creation of UCMs with all the previously discussed notational elements. It also supports the definition of UCM scenarios and global variables that create the context for the scenario. Once the maps are created, they can be saved as an XML (*.ucm.xml) file or exported to several visual formats. The *.ucm.xml files that save the created UCM are valid according to a defined UCM Document Type Definition (DTD) that is available at [UCM]. This DTD describes the formal definition of UCMs with respect to their notation and how these definitions can be applied. UCMNav uses this DTD to ensure that syntactic and static semantic rules are satisfied [MIG01].

Furthermore, UCMNav provides the ability to convert created (or imported) UCMs to other models such as the performance model Layered Queuing Networks (LQN) and message sequence charts (details on how these transformations are accomplished can be found in [PET02] and [MIG01] respectively). As well, the simple path data model supports scenario definitions that permit the export of individual scenarios to an XML file [AMY03b].

In what follows, we provide an in-depth discussion on UCM definitions that are particularly relevant to this research. The scenario definition specifies how scenarios are outlined in UCM models while the formal definition of UCM specifications allows for an unambiguous interpretation of the developed map.

3.4 UCM Scenario Definition

The methodology of extracting individual scenario details from UCMs is discussed in [AMY03b]. In that research, algorithms were developed to extract the details of a single scenario from a UCM and output it to a format that would be amiable to further transformations. XML was chosen as the output format. We concern ourselves with the specifics of this extraction because these XML files will be used as input for our developed tool that will automate the approaches we later define.

```
<!ELEMENT scenario-list (scenario-group)* >
<!ELEMENT scenario-group (scenario-definition)* >
<!ATTLIST scenario-group
            name        NMTOKEN   #REQUIRED
            description CDATA     #IMPLIED >
<!ELEMENT scenario-definition ((scenario-start)*,
                               (variable-init)*, (postcondition)*) >
<!ATTLIST scenario-definition
            name        NMTOKEN   #REQUIRED
            description CDATA     #IMPLIED >
<!ELEMENT scenario-start EMPTY >
<!ATTLIST scenario-start
            map-id      IDREF     #REQUIRED
            start-id    IDREF     #REQUIRED >
<!ELEMENT variable-init EMPTY >
<!ATTLIST variable-init
            variable-id IDREF     #REQUIRED
            value       (T|F)     #REQUIRED >
<!ELEMENT postcondition EMPTY >
<!ATTLIST postcondition
            variable-id IDREF     #REQUIRED
            value       (T|F)     #REQUIRED >
```

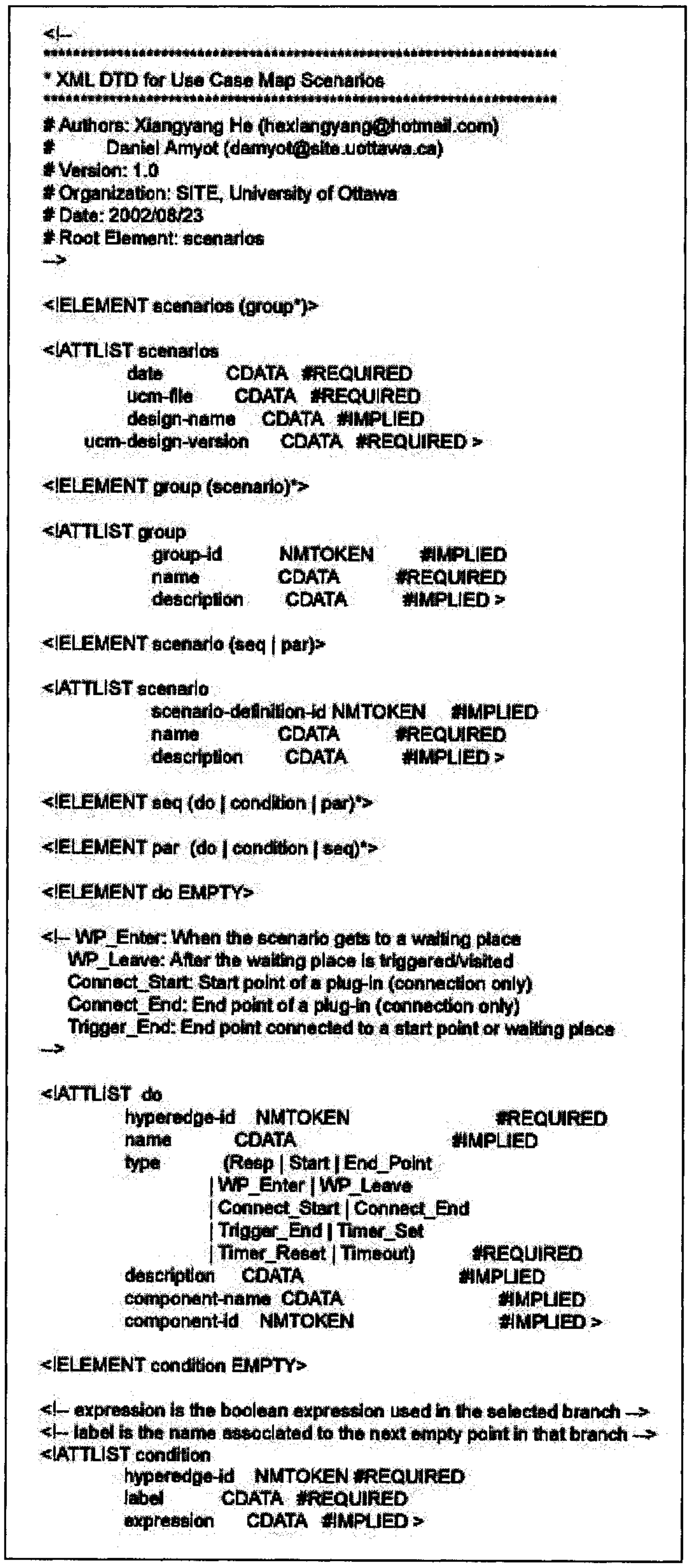
Figure 3-13: Scenario Definition DTD

According to [AMY03b], system scenarios must be defined in UCMNav before information about them can be extracted. Scenario definitions describe scenarios represented in the UCM in terms of their initial values for the system global variables and their start points triggered. Figure 3-13 [AMY02] presents the DTD that formally defines the structure of scenario definitions. Definitively, each scenario definition must contain a name, the list of start points to be triggered, the initial values of the global variables (either true or false), and (optionally) a post-condition used to assert the validity of a scenario once the traversal has completed [AMY03b].

Furthermore, individual scenarios are separated from UCMs by applying a scenario traversal algorithm that was developed as part of the research of [AMY03b]. This algorithm traverse the UCM with respect to the chosen scenario and ensures that the contents of the outputted XML file (valid with respect to the Document Type Definition shown in Figure 3-14) contains all the relevant information about path concurrency, responsibilities, components, and plug-ins,. The XML elements that specify the extracted scenario contain attributes that preserve the traceability information to the original UCM [AMY03b].

The research presented in [AMY03b] has been integrated into UCMNav. The tool provides the interfaces for defining scenarios and organizing them into groups and for selecting global variable values that form scenario conditions. UCMNav can also produce an XML file containing individual scenarios or groups of scenarios.

The XML DTD (Figure 3-14 [UCM]) for extracted scenarios specifies that scenarios can be part of one or more group and that a scenario supports the recursive use of sequence (<seq>) and parallel (<par>) XML elements to detail its sequences and concurrent paths. The extracted scenarios are partial orders and, as such, contain information about sequences and concurrency, but not alternatives [AMY03b]. Plug-ins within UCMs are resolved as follows: the appropriate plug-in for the scenario is selected and plug-in components and elements are allocated to the component containing the parent stub [AMY03b].



```
<!–
*******************************************************************************
* XML DTD for Use Case Map Scenarios
*******************************************************************************
# Authors: Xiangyang He (hexiangyang@hotmail.com)
#        Daniel Amyot (damyot@site.uottawa.ca)
# Version: 1.0
# Organization: SITE, University of Ottawa
# Date: 2002/08/23
# Root Element: scenarios
–>

<!ELEMENT scenarios (group*)>

<!ATTLIST scenarios
           date          CDATA   #REQUIRED
           ucm-file      CDATA   #REQUIRED
           design-name     CDATA   #IMPLIED
      ucm-design-version     CDATA   #REQUIRED >

<!ELEMENT group (scenario)*>

<!ATTLIST group
              group-id        NMTOKEN       #IMPLIED
              name            CDATA         #REQUIRED
              description      CDATA         #IMPLIED >

<!ELEMENT scenario (seq | par)>

<!ATTLIST scenario
              scenario-definition-id NMTOKEN     #IMPLIED
              name            CDATA         #REQUIRED
              description      CDATA         #IMPLIED >

<!ELEMENT seq (do | condition | par)*>

<!ELEMENT par  (do | condition | seq)*>

<!ELEMENT do EMPTY>

<!– WP_Enter: When the scenario gets to a waiting place
   WP_Leave: After the waiting place is triggered/visited
   Connect_Start: Start point of a plug-in (connection only)
   Connect_End: End point of a plug-in (connection only)
   Trigger_End: End point connected to a start point or waiting place
–>

<!ATTLIST  do
          hyperedge-id   NMTOKEN                     #REQUIRED
          name          CDATA                    #IMPLIED
          type          (Resp | Start | End_Point
                        | WP_Enter | WP_Leave
                        | Connect_Start | Connect_End
                        | Trigger_End | Timer_Set
                        | Timer_Reset | Timeout)        #REQUIRED
          description     CDATA                     #IMPLIED
          component-name  CDATA                       #IMPLIED
          component-id    NMTOKEN                     #IMPLIED >

<!ELEMENT condition EMPTY>

<!– expression is the boolean expression used in the selected branch –>
<!– label is the name associated to the next empty point in that branch –>
<!ATTLIST condition
          hyperedge-id   NMTOKEN #REQUIRED
          label         CDATA  #REQUIRED
          expression     CDATA  #IMPLIED >
```

Figure 3-14: Scenario1.DTD

3.5 UCM Formal Definition

As previously mentioned, UCMs are intended to be used for specifying requirements. The UCM notation itself and its underlying semantics can be regarded as informal. There is an ongoing research effort to formalize the semantics of UCMs. Hassine et al. in [HAS04] provide a formal definition for a UCM specification; it defines what a UCM consists of and how its elements are related.

Hassine assumes a UCM Requirement Specification, RS, is denoted by (D, C, V, λ, B) where:

- D is the UCM domain that includes all possible UCM domain elements (start points, responsibilities, end points, etc.)
- C is the set of components in RS (C = ∅ for unbound UCM)
- V is the set of global variables in RS
- λ is a transition relation defined as: $\lambda = D \times D \times E$, where E is the set of the logical formula over V.
- B is a Binding relation, defined as: $B = D \times C$. B defines which element of D is associated with which component of C.

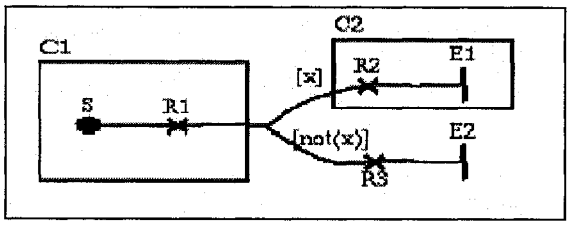


Figure 3-15: Simple UCM

The following example, also provided in [HAS04], serves to clarify the meaning of this formal definition. The UCM in Figure 3-15 can be described as follows:

- D = {S, R1, R2, R3, E1, E2, OF1} where OF1 is the OR-Fork.
- C = {C1, C2}
- V = {x}
- λ= {(S, R1, true),(R1, OF1, true),(OF1, R2, x),(OF1, R3, $\neg$x),(R2, E1, true),(R3, E2, true) }
- B = {(S, C1),(R1, C1),(R2, C2),(E1, C2)}

The defined grammar of this formal definition of UCMs provides the basis our developed definitions that are relevant to change impact analysis. We make use of this grammar in the chapters to follow.

This concludes the background section of this research. In the first part we discussed software evolution, specifically change impact analysis and current approaches used for identifying the impact of a change. This was followed by a detailed review of UCMs – their notation, their benefits and current applications, and the definitions that have been applied to them. In what follows the contributions of this research are presented with focus on applying current change impact approaches to UCMs.

4. Applying Change Impact Analysis Techniques to UCMs

The goal of this research is to show that UCMs can be used during change impact analysis by applying existing dependency-based impact analysis techniques to them. The stated goal is based on two major factors: defining the approaches that we will apply to UCMs as well as defining the algorithms necessary to produce the desired output. We use the formal definition of UCM specifications discussed in section 3.5 to support our approaches. In this section details on the dependency analysis approaches that we applied to UCMs are presented. This is followed by a section discussing how the impact sets for changed UCM elements are identified.

4.1 UCM Scenario Dependencies

Identification of scenario dependencies can support the understanding of system operations and their inter-relationships. In what follows, we describe in detail how scenario dependencies (discussed earlier in section 2.4.2.1) can be applied to UCMs. We ascertain that these dependencies can be applied to UCM scenarios to understand the relationships among system scenarios.

4.1.1 Functional Dependency

Scenario functional dependency, as discussed in [BAI02] and [TSA01], captures the coexistence of two or more scenarios inside the same conceptual (or logical) cluster. At the UCM level, one can apply the idea of functional system decomposition via a scenario tree, in which the functional dependencies for each scenario are determined.

The thin threads used to form the basis of the system functional decomposition in [TSA01] are comparable to UCM scenarios. A thin thread is defined as a minimum usage scenario that describes a complete system scenario from the end users' point of view. This effectively defines UCM system-level scenarios (assumed henceforth and simply referred to as system scenarios) which require external stimulation and propagate through the system at the root level. These system scenarios include within their paths plug-in scenarios.

Before one can create a functional hierarchy of a system, functional groups within the UCM have to be defined. Functional groups are sets that contain system scenarios that carry out the same goal and suggest functional dependency among its elements with respect to the requirement specification.

This functional dependency among scenarios can then be arranged hierarchically to form a functional decomposition tree. The structure, shown in Figure 4-1 can be described as follows: The UCM for which this composition applies is set as the root of the tree, while the defined groups form the tree's internal branches. The system scenarios of the UCM form the functional decomposition tree's leaf nodes. Plug-ins, which may be contained within scenarios, are not included within this hierarchy since the same plug-in can be included in multiple scenarios, thus multiple groups.

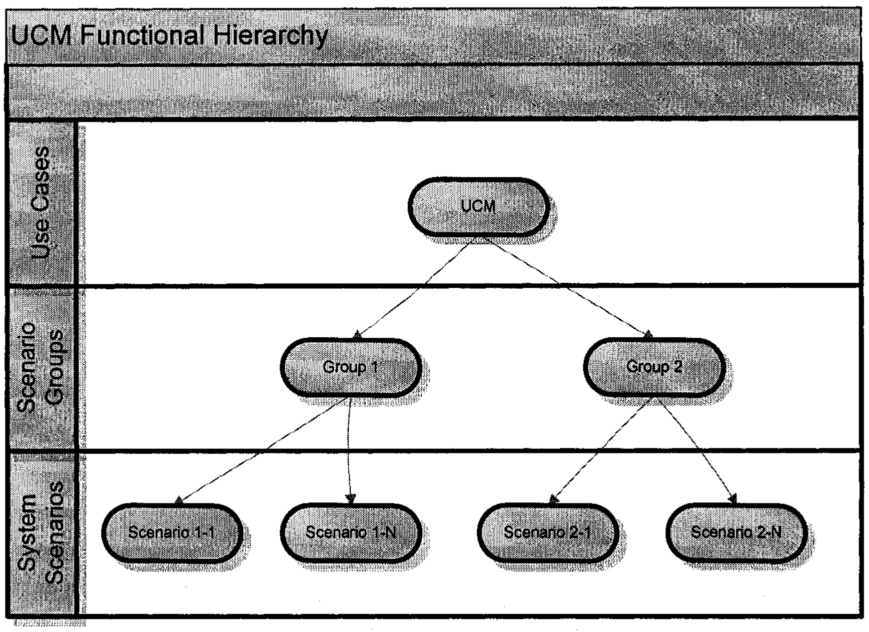


Figure 4-1: UCM Functional Decomposition Tree

UCM scenarios describe the functionality of the system in the context of the system's components. System behaviour may consist of large numbers of scenarios that describe each functional execution as well as its alternatives. This functional decomposition aims to provide some structure to the UCMs' defined scenarios to support system comprehension.

4.1.2 Containment Dependency

As discussed earlier, UCMs support model abstraction by allowing some scenario details to be hidden from the root map through the use of stubs and plug-ins. At run-time, the stubs are filled with the appropriate plug-in map that contains the hidden execution details, endorsing the usage of plug-in maps in multiple scenarios. We aim to identify which parts of a UCM system scenario is actually part of a plug-in map through the extension of previously defined notions of containment dependency.

For UCMs, we define a containment dependency to exist between two scenarios, *A* and *B*, if the path of *A* is fully contained within the path of *B*, but does not equal *B*. A UCM scenario is actually a sequence of executed responsibilities, a subset of the transition relations (λ) in the domain of the UCM. We then denote the sequence of a scenario, *X*, by λ_X.

> ***Definition:*** *(Containment Dependency)*
>
> *For two scenarios A and B and their sequences* λ_A *and* λ_B*, respectively, A is contained in B if* $\lambda_A \subset \lambda_B$.

Since the export feature in UCMNav does not provide plug-in paths outside the context of the system scenarios in which they are contained, a work around has to be provided. We must resolve the UCM domain elements within the scenario traversal that are bounded by a plug-in map. Plug-in start and end points are actually UCM domain elements called *connect_start* and *connect_end*, respectively, so that they can be differentiated from start and end points that require external stimulation. Thus, we identify scenarios that contain plug-in path segments by searching the UCM domain elements within its sequence for one or more domain elements of type *connect_start*. If a plug-in start point is encountered, then all the succeeding elements up to and including the corresponding *connect_end* domain element are deemed to be those enclosed by the bounds of the plug-in map.

Once a plug-in path is identified, we separate the sequence of domain elements that constitute the plug-in scenario traversal and create a relationship between this contained path and the scenario that makes use of it. Subsequent discoveries of the same path in other scenarios also create a relationship between these scenarios and the plug-in path.

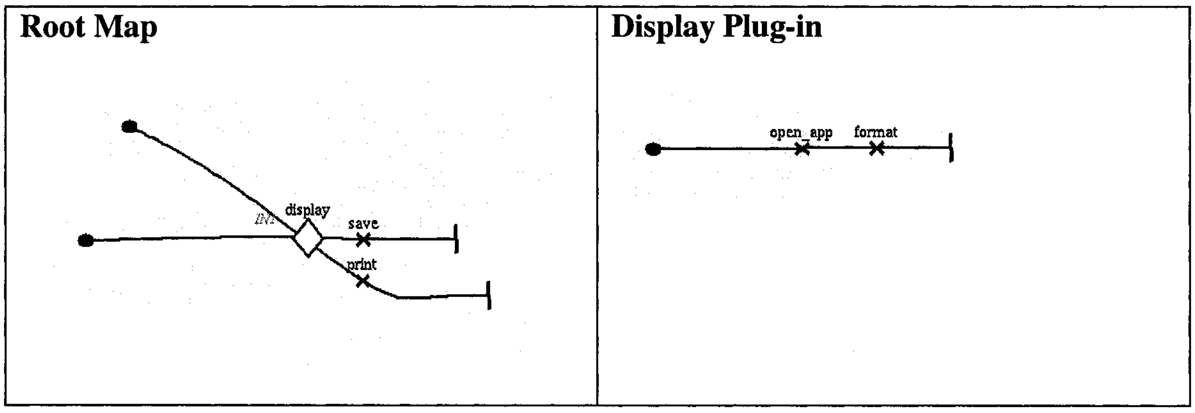


Figure 4-2: UCM Root Map and Plug-ins

After the isolation of plug-in paths, it is then possible to produce containment dependency information. For a given system scenario, hierarchical list of all its contained paths (hierarchical because plug-in paths may in turn contain their own plug-in segments) can be generated to show all the sub-scenarios upon which the scenario depends. Similarly, all the scenarios that contain a specific plug-in path can also be generated to show these scenarios that are related through their dependency on the plug-in path. This dependency information can then be used to provide an understanding of all the scenarios that may be affected if a plug-in path is changed.

As an example, Figure 4-2 displays a root UCM that contains a stub, called *display*, which is required by both scenarios. The plug-in for the stub is shown on the right. The start and end point of the plug-in path would be defined in UCMNav and the scenario XML file as *connect_start* and *connect_end* respectively, allowing for the identification of the plug-in within both system scenario paths. Thus, an analysis of either scenario would confirm a containment dependency between it and the plug-in sequence on the right. Furthermore, an analysis of the scenarios that make use of the shown plug-in scenario would produce a set containing both of the scenarios in the root map.

4.1.3 Global Variable Dependency

Conditions are used to define the context for when a scenario executes and (implicitly) when it does not. UCMs make use of Boolean global variables to represent system state. A *True* value for the global variable implies the existence of the system state, while a *False* value represents the opposite. That is, each variable $v \in V$ has two values that it can be assigned which we denote as vt and vf, for True and False, respectively. At the time of scenario definition, a value for each global variable that is required for scenario execution must be specified. Figure 4-3 shows the user interface in UCMNav that provides the means for accomplishing this.

Since scenarios within a UCM make use of the same global variables, we propose definitions of relationships between these scenarios based on their conditions (global variable values). To do this we exploit the condition relationship definitions presented in

[PAU01]. We first identify relationships between conditions as described in [PAU01] and then relate the scenarios that execute based on these conditions to generate dependency information at the scenario level. We have defined four different types of dependencies between scenarios and their global variable values: *Related*, *Value Related*, *Value Alternate*, and *Independent*.

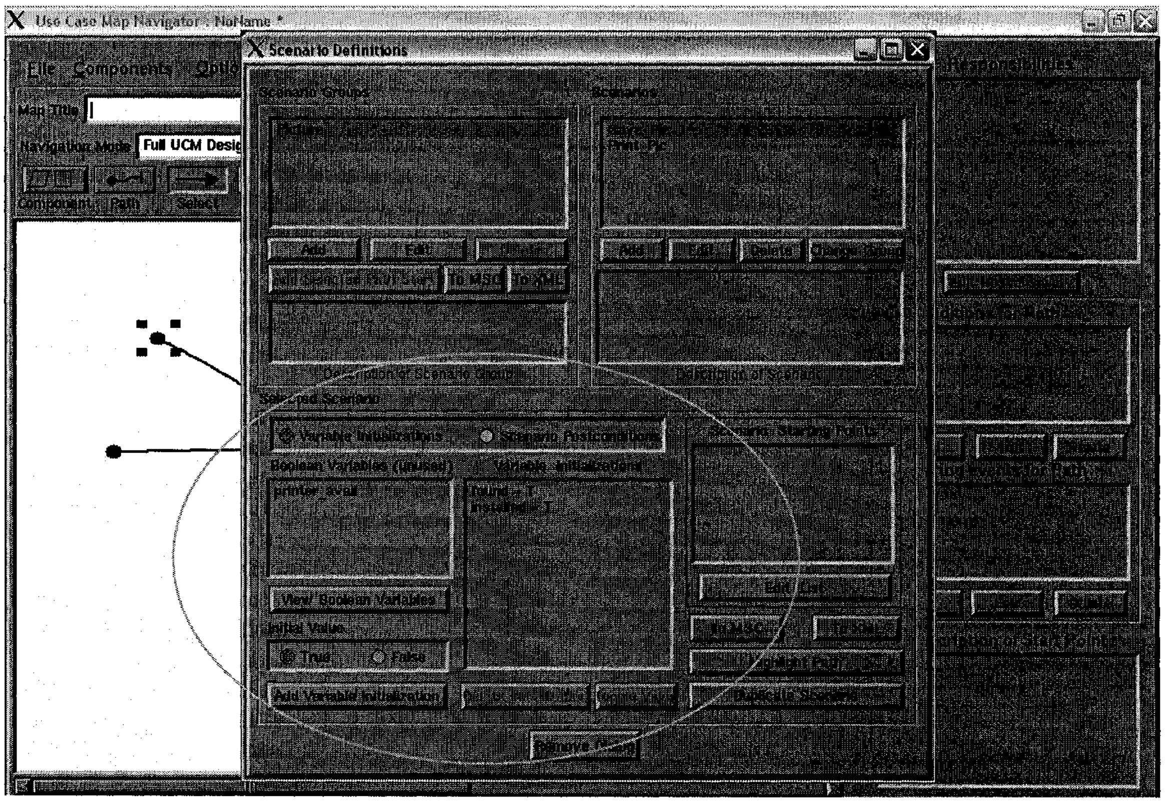


Figure 4-3: UCMNav Global Variable Value Selection

For a given scenario, we define a *Global Variable Related* set as one containing those scenarios that are dependent with the input scenario through the requirement of the same global variable(s), regardless of the value of the variable. We suggest a further reduction of this set by introducing the notion of a *Global Variable Value Related* set – a set that includes only scenarios that use the same value for any of the global variables required by the input scenario. The formal definitions of these two dependencies are as follows:

Definition: *(Global Variable Related)*

Let a value for a global variable be vx | vx=vt or vx=vf. Also let the set of global variable values for a specific scenario be Vx. Two scenarios, C and D are condition related if for a value vx of v (v∈ V), vx ∈ Vx_C and vx ∈ Vx_D

Definition: *(Global Variable Value Related)*

Let a value for a global variable be vx | vx=vt or vx=vf. Also let the set of global variable values for a specific scenario be Vx. Two scenarios, C and D are condition value related if for a value vx of v (v ∈ V), (vx=vt ∈ Vx_C and vx=vt ∈ Vx_D) or (vx=vf ∈ Vx_C and vx=vf ∈ Vx_D)

Table 4-1 depicts an example of scenario initializations for a UCM (where "/" corresponds to a variable that has not been initialized for that particular scenario). Given scenario *S1* as the input for the analysis, scenarios *S2* and *S3* would be included in the *Global Variable Related* set, while only *S2* would be included from the *Global Variable Value* set.

Scenario Name	Global Variables		
	Found	On	Opened
S1	T	F	T
S2	T	/	/
S3	/	T	F

Table 4-1: UCM Scenario Initializations

Furthermore, scenarios that are related by the same global variable but use the opposite value are part of a more specific set, so-called *Global Variable Value Alternates*. Scenarios that are deemed to be alternates have opposing condition values for the same global variable implying that they cannot occur at the same time. Referencing Table 4-1 with the *S1* as input, *S3* would be an alternate scenario.

Definition: *(Global Variable Value Alternates)*

Let a value for a global variable be vx | vx=vt or vx=vf. Also let the set of global variable values for a specific scenario be Vx. Two scenarios, C and D are global variable alternates if for a value vx of v (v∈ V), $vx=vt \in Vx_C$ *and* $vx=vf \in Vx_D$

The last global variable-based dependency we define actually refers to a lack of dependency; these scenarios do not share any of the same global variables with the given scenario. This definition is the exact opposite of that of *Global Variable Related* and is termed *Global Variable Independent.*

Definition: *(Global Variable Independent)*

Let the set of global variable values for a specific scenario be Vx. Two scenarios, C and D are global variable independent if for any condition v (v∈ V) such that v $\in Vx_{C,}$ $v \notin Vx_D$

As shown in Table 4-1, for the input scenario *S1*, the *Global Variable Independent* set would be empty since the other two scenarios do share its conditions.

4.1.4 Execution Dependency

UCMs depict behaviour over a static system using scenario paths. Since the structure of the system within the model is fixed, the scenarios must share existing system elements (domain elements and/or components). To provide information on how scenarios relate through these system elements we make use of execution dependencies defined in [PAU01] and [BAI02].

For UCMs, we propose that scenarios are execution dependent if they share common elements. That is, these scenarios execute the same components or domain elements along their paths. We have also defined the inverse of execution dependent, execution independent, as two scenarios that do not execute any common UCM elements.

Furthermore, we can analyze scenario execution dependencies at two levels of abstraction: component and domain element. At the component level we ascertain that a dependency exists between two scenarios if they both contain the same component in their execution sequence (termed *Component Execution Dependency).* At the more specific level, two scenarios are domain element execution dependent if they both execute the same UCM domain elements (termed *Domain Element Execution Dependency).* Scenarios that are domain element execution dependent are also component execution dependent since responsibilities are contained within components, however, the inverse does not hold. The formal definitions are as follows:

Definition: *(Component Execution Dependency)*

For two scenarios A and B and their respective sequences λ_A and λ_B, A and B are component execution dependent if a component c ($c \in C$) exists such that $c \in \lambda_A$ and $c \in \lambda_B$.

Definition: *(Domain Element Execution Dependency)*

For two scenarios A and B and their respective sequences λ_A and λ_B, A and B are domain element execution dependent if a domain element r ($r \in D$) exists such that $r \in \lambda_A$ and $r \in \lambda_B$.

Figure 4-4 illustrates a UCM that contains three scenarios, referred to as A, B, and C whose start and end points are respectively: (*b*,*x*), (*b*, *y*), and (*c*,*z*). Notice that scenarios A and B share the same starting point, *b*. Given scenario A as input for execution dependency analysis, scenarios B and C would be deemed component execution dependent, while only scenario B would be considered domain element execution dependent.

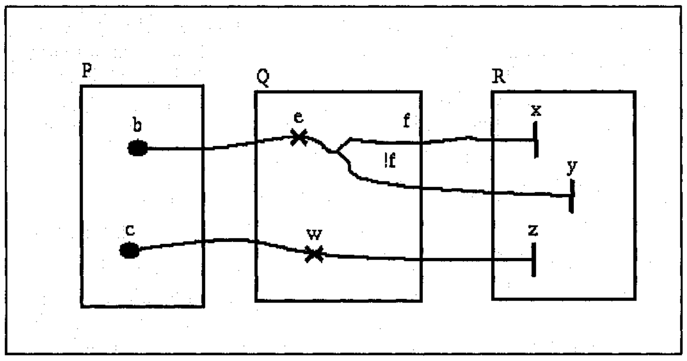


Figure 4-4: UCM for Execution Dependency

For a given scenario, its execution dependent scenarios are determined by analyzing all the domain elements executed by the scenario. Iteratively, for each domain element in the scenario sequence, we obtain all the other scenarios that execute that domain element and add them to the set of dependent scenarios. To identify component execution dependencies among scenarios, the component by which each executed domain element is bound are identified. These components are then further analyzed to discover the other domain elements they bind. The scenarios that execute these other domain elements are then added to the dependency set. Therefore, the component execution dependency set for a given scenario will include the scenarios that share the same components.

4.2 Component Dependencies

UCMs support the comprehension of the components that make up a modeled system through the analysis of the scenario's paths that inter-connect these components. The scenarios explicitly create relationships between the components of the system, implying that component relationships depend on the system scenarios to provide the semantic information about their dependencies [WEI98]. Since UCMs do not provide information on the type of dependency (control or data), we can only determine that a dependency exist between components.

In this section we elaborate on the current definition of UCM component interfaces and detail our component dependency definition.

4.2.1 Component Interfaces

UCM scenario paths traverse the system and along its way responsibilities that are bound to components are executed. If any path traverses one component and enters another then the two components must interact to ensure the causality required by the scenario occurs between the two components [HAS04]. In [WU01], an interface is defined as access points of the component that invoke events within that component when it is accessed. Further, Hassine in [HAS04] defines a UCM interface as follows:

Definition: *(Component Interface) [HAS04]*

Let RS = (D, C, V, λ , B) be a UCM, R is the set of responsibilities, start/end points of RS (R $\subseteq$ D) and c $\in$ C a component. A component Interface Ic is defined as a subset of R (Ic=R' and R' $\subseteq$ R) where R' is the set of responsibilities, start/end points that defines the interaction between c and other components in C.

The above definition specifies that a component interface is a responsibility, a start point, or an end point that defines interaction with other components. Start and end points must be included in the component interface set since the start point is where the component receives stimuli and the end point is where it returns control. We extend this definition to clarify the component interface responsibility in an effort to make future automatic extraction more specific.

Definition: *(Component Interface) Revised*

Let RS = (D, C, V, λ , B) be a UCM. Let R be the set of responsibilities, start/end points of RS (R $\subseteq$ D) and c $\in$ C a component. A component Interface Ic is a subset of R (Ic=R' and R' $\subseteq$ R) where R' are the start/end points and responsibilities bound to c ((R', c) $\subseteq$B') such that for each domain element, r, in Ic there exists a transition to a domain element w (w $\in$ R and (r, w)$\in$ λ) and w is not bound to c ((w, c) $\notin$ B).

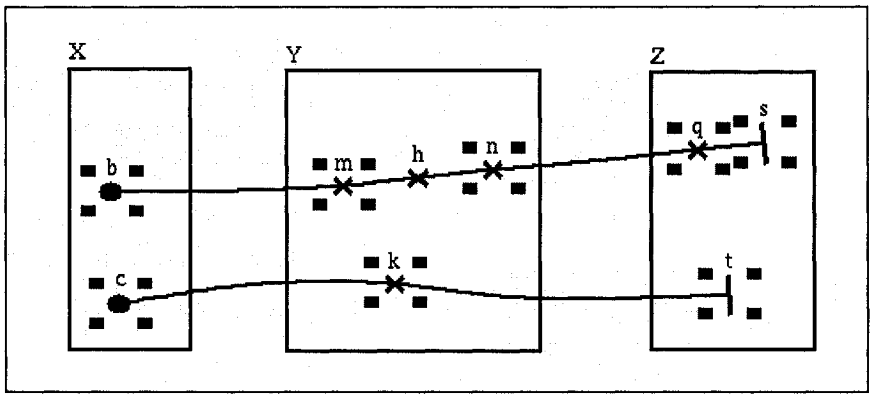


Figure 4-5: UCM with Component Interfaces Highlighted

Figure 4-5 shows a UCM with the interfaces of each component highlighted. Specifically, component *X* has domain elements *b* and *c* as interfaces; component *Y* has *m*, *n*, and *k* as interfaces; and component *Z* has *q*, *s*, and *t* as its interfaces. As defined, all start points, end points and responsibilities which have paths that lead to other components are included in these sets.

We introduce the concept of component interface domain elements but refrain from describing any operational details of the interface. The general definition provided simply implies that the identified interface domain elements are used to support the necessary collaboration between it and other components.

4.2.2 Component Dependency Identification

UCM scenario paths create relationships among the components by which the domain elements they execute are bound. For this reason, a UCM can easily be viewed as a component dependency graph. We make this assumption by applying to UCMs the following definition of a component dependency graph presented by Yacoub et al. in [YAC04]:

> ***Definition:*** *(Component Dependency Graph) [YAC04]*
>
> *A tuple* $(\mathrm{N}, \mathrm{E}, s, t)$*, where* (N, E) *is a directed graph and* s *is the start node and* t *is the end node.* N *is the set of nodes in the graph and* E *is the set of directed edges.*

A UCM corresponds to a graph with directed edges created by the causal nature of its scenarios – scenario paths begin at start points and progressively continue to end points, explicitly indicating direction. This concept is visualized in Figure 4-6 where the UCM includes directional arrows to further specify scenario directions. The components within the UCM correspond to the nodes in the graph, while the path segments between components (implying component interaction) can be regarded as the directed edges. The start and end nodes where the scenarios begin and end, are contained within components, thus these components form the sources and sinks of the graph. A UCM graph, due to the possible containment of multiple scenarios may contain multiple start and end points.

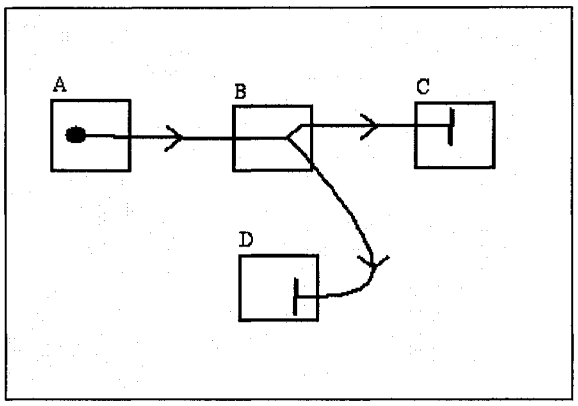


Figure 4-6: UCM Showing Scenario Direction

The dependencies of a component are identified by analyzing the scenarios that execute its domain elements. A component dependency exists between two components if there is a directed edge between them.

Definition: *(Component Dependency)*

For a given component x (x ∈ C), let its interface set be Ic_x. A component y (y ∈ C) has a dependency with x if there exists some scenario A and its execution sequence, λ_A, such that an interface of x, r (r ∈ Ic_x) executes before or after an interface, q of y (q ∈ Ic_y).

Due to the sequential nature of a scenario execution, we can further refine our defined component dependencies to specify whether one component depends on another or whether another component depends on it. We term these two component dependency specifications *backward* and *forward* dependency, respectively. This notion is derived from work in [KOR04] where Korel et al. define affecting and affected state chart transitions.

Components are *Forward Dependent* on a specific component *c*, if within at least one scenario sequence their interface domain elements are executed after that of *c*. That is, there exists a directed edge from *c* to these other components. Similarly, *c* is *Backward Dependent* on other components if directed edges exist which emanate from them and lead to *c*. In Figure 4-7 the dependencies of component *Y* are highlighted; component *X* is backward dependent with *Y* where as *Z* is forward dependent.

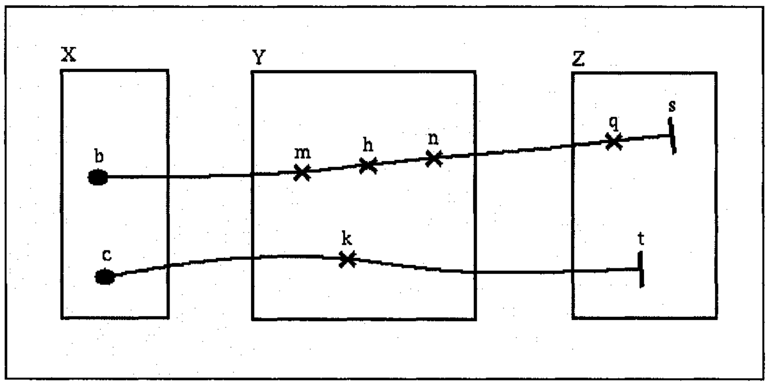


Figure 4-7: UCM with Component Dependencies of *Y* Highlighted

Viewing a UCM as a component dependency graph allows us to perform component dependency analysis using UCM. Component dependency information extracted from

UCM is useful in change impact analysis for the same reasons a general dependency graph is useful – as an indicator of component interactions and its level of coupling.

4.3 UCM Impact Analysis

Impact analysis is the process of identifying the other parts of the system that may be affected by a change. In order to advocate UCMs as a system model for performing change impact analysis, we realize that the identification of impacts must be supported. In this section we present our approach for generating the impact sets with respect to UCMs. Initially we specify our assumptions about the initial change set and follow with discussions on how we identify ripple effects of UCM elements.

4.3.1 The Initial Change Set

The initial change set consists of elements known to be affected by the planned change. The identification of the elements that need to be modified first to accomplish the change does require some effort to analyze the change and the system. We assume that these initial elements are known and therefore focus our approach on identifying what may be affected by these initial changes.

Since all aspects of the UCM notation are related (scenarios, components, and domain elements), it is possible to identify the impact set for a change at all three levels of granularity; scenarios are the highest level of abstraction, while the impact set at the domain element level would provide the most specific analysis. Depending on the level of analysis desired, we require that the initial change element at that level be provided as input. For instance, if impact analysis is desired at the component level, but a change actually occurs to a domain element, then the component by which the changed domain element is bound is the requisite for determining the impacted components.

4.3.2 Scenario Impact Set

Scenario impact sets can be determined with the use of the scenario dependencies defined in section 4.1. This concept is based on a similar technique employed by Tsai et al. in [PAU01b] for the determination of impact sets for test scenarios. That work discusses a slicing method that requires a changed scenario and the dependency attribute as the slicing criterion. Based on this input the selected dependency is applied to scenarios to determine the impact set.

In this research we do not make use of a slicing algorithm but do adopt the idea of allowing users to choose the dependency to apply for the generation of an impact set. This is done to avoid making claims about the appropriate dependencies that should be applied to a given changed scenario. Making such claims would require empirical evidence to validate them, which is unavailable in the existing literature. Also, conducting an empirical analysis would go beyond the scope of this research. Consequently, for a given scenario and a selected dependency, the impact set will contain all scenarios dependent with the given scenario with respect to the selected dependency.

4.3.3 Ripple Effect Analysis

The behavioural specifications contained in UCMs support the identification of two types of impact sets that we term *static* and *dynamic*. A static impact set contains all the components or domain elements (depending on the desired level of analysis) that interact with the given input through any of the modeled scenarios. Similar to other approaches that have been defined as static, this impact set includes any UCM elements that could possibly be impacted by the change irrespective of any behavioural details. Alternatively, a dynamic impact set is specific to an execution path. It contains a (possibly) more distinct set of elements which are specific to a given scenario.

In order to generate the impact set that contains the ripple effects of the change, we implement a transitive closure algorithm that identifies all elements in the UCM that are reachable from the given input. We conclude that a UCM element is reachable if an element in the impact set contains a transition to it [PAU01b][TSA03].

> ***Definition:*** *(Forward Reachable Element)*
>
> *For two elements d and e, e is reachable from d if there exists a scenario sequence (λ_X) such that a transition relation containing d executes before a transition relation containing e.*

We again make use of the notion of ripple effect discussed in [KOR04] and identify two types of ripple effects using our transitive closure algorithm. Elements that are forward reachable from the given input are deemed to be *forward ripple effects* while elements from which the given input is reachable constitute *backward ripple effects.* The identification of backward ripple effects makes use of our developed definition of backward reachable elements.

> ***Definition:*** *(Backward Reachable Element)*
>
> *For two elements d and e, e is backward reachable from d if there exists a scenario sequence (λ_X) such that a transition relation containing d executes <u>after</u> a transition relation containing e.*

These definitions are applied to determine both component and domain element ripple effects as detailed in the following sections.

4.3.3.1 Component Ripple Effect Analysis

Our component ripple effect analysis technique aims to identify other components that may be affected by an initially changed component. Since UCMs may be created without components, we assert that component ripple effect analysis can only be performed on those UCM models that contain component specifications.

For a given component, the impact set is determined by applying our previously defined forward and backward component dependencies. For static ripple effect analysis, the forward and backward dependent components for the given input are identified and added to the respective impact set. The complete backward impact set is then generated by iteratively identifying backward dependencies for all components in that set, marking each as it is analyzed. Similarly, the forward impact set is produced by identifying the forward dependencies of the given component and each component is added to the set. The process ends when there are no further unprocessed elements in the impact set.

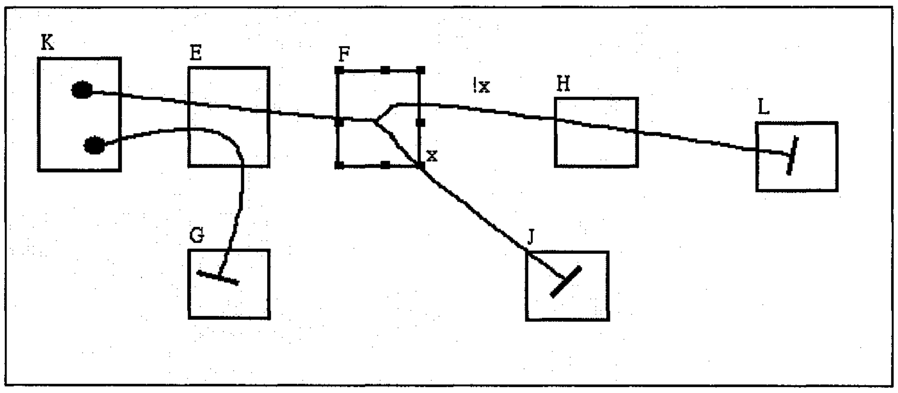


Figure 4-8: UCM Showing Ripple Effects of Component *F*

Dynamic ripple effect analysis requires the additional input of the scenario that should be used to identify the possible change impacts. Given this and the initially changed component, the analysis of the impact sets follows the same methodology as that of static analysis, except only the dependencies occurring with respect to the input scenario path are included in forward and backward ripple effect set.

As an example, Figure 4-8 shows the identified ripple effects for component *F*. Applying static impact analysis would produce a forward impact set containing components *H* and *L* and a backward impact set containing components *E* and *K*. Dynamic impact analysis with respect to the scenario that begins in component *K* and ends in component *J*, would have a forward impact set that includes only component *J* whereas the backward set would remain unchanged, containing both *E* and *K*.

4.3.3.2 Domain Element Ripple Effect Analysis

Once the initial changed domain element is known, forward and backward ripple effects for both static and dynamic analysis can be identified as defined below. Similar to components, a static change impact set is generated for a given domain element by analyzing all the scenarios that execute it. The forward impact set contains all the domain elements that are reachable from the input , while the backward impact set will be comprised of those domain elements that are backward reachable from the given domain element.

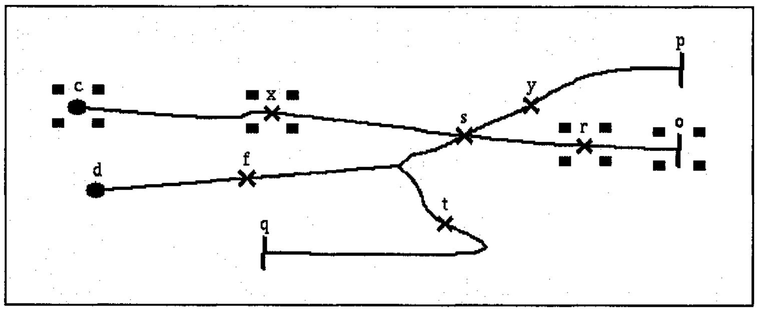


Figure 4-9: UCM Showing Path Specific Impact Set of *s*

A restricted dynamic impact is produced by requiring a scenario as an additional input. The forward and backward ripple effects are then identified only with respect to the given scenario. This effectively limits the forward impact set to those domain elements that are executed after the given input during the specified scenario's execution and the backward impact set to those domain elements that execute before it.

As shown in Figure 4-9, the dynamic impact set for responsibility *s* with respect to the scenario that begins at *c* and ends at *o* is highlighted; the forward impact set includes domain elements *r* and *o* while the backward impact set includes domain elements *x* and *c*. Static ripple effect analysis on this UCM, with *s* as input, identifies a forward impact set containing domain elements *y*, *p*, *r*, and *o* and a backward impact set containing domain elements *c*, *x*, *f*, *d*.

In this chapter definitions and algorithms were proposed that can be regarded as a framework for using UCMs for change impact analysis. Scenario and component dependencies were introduced as an aid to system comprehension, while the ripple effect analysis methods can be used to scope the impacts of a change at three levels of abstraction. The next chapter details how we automated our theories through tool support.

5. UCM Analyzer

This chapter describes in detail the tool that was developed to automate the ripple effect analysis at the UCM level, called UCM Analyzer. We provide a brief overview of its usage and then discuss some implementation details with respect to the tool design itself.

5.1 UCM Analyzer Usage

The goal of UCM Analyzer is to: 1) create the relationships among elements of a UCM to support further analysis, and 2) provide dependency and ripple effect information about the UCM. The tool's interface is divided into four main areas that support the analysis of the UCM.

Scenario Importer

To create an accurate representation of the UCM, we require the scenario XML files produced by UCMNav be available as input for our tool. In order to analyze all the relationships among the UCM elements it is necessary to import all the scenario XML files for the UCM into UCM Analyzer. The *Scenario Importer* interface allows users to select these XML files which are then automatically parsed by UCM Analyzer to derive the necessary relationships among the scenario elements.

UCM Details

The tool displays the details of the chosen UCM. Since the UCM Analyzer supports the storage of data for multiple UCMs, the user must select a UCM from the list that is populated from the database. Once selected, all scenarios, Boolean global variables, components, and domain elements that comprise the selected UCM are listed in the display area.

Dependency Analyzer

For dependency analysis, users select the desired dependency analysis level – either scenario or component. For the scenario dependency analysis it is necessary that the scenario and the type of dependency that should be applied are provided as input. Alternatively, for component dependency information, users must input a specific component from the list of components and select a scenario from the list of available scenarios if a path-specific dependency is desired.

Impact Analyzer

Users can choose among three different levels of granularity when requesting impact information. Scenario impact analysis makes use of scenario dependency information, and users are redirected to the Dependency Analyzer section to obtain information on dependent scenarios. Component and domain element impact analysis can produce ripple effect information for all scenarios or a specific scenario (if a scenario is also provided as input). The displayed information identifies the impact set for both forward and backward impacts indicating to the user which components or domain elements are affecting and are affected by the input.

5.2 Tool Design

5.2.1 Tool Architecture

UCM Analyzer is a Java implementation that was developed using the IBM Eclipse IDE and the MySQL database management system. The architecture of UCM Analyzer is depicted as a UML2 component diagram in Figure 5-1.

The user interface for the tool is referred to as *Change Analysis*. It handles all user input and produces text output to the display area. If the input is an XML file that is to be parsed the user interface accesses the *UCM Generator* component to handle this task. The tool is populated by each XML file individually so that the relationship information between the scenario and its elements is maintained. During parsing, *UCM Generator* accesses the *Object Manager* component to create the necessary objects and relationships for each UCM element found in the XML file. The objects in the *Object Manager* component access the database via the *DB Manager* component for the purpose of saving and retrieving information. The *DB Manager* component accesses the MySQL database using the Java Database Connectivity (JDBC) API standard SQL database access interface.

The *Component Dependency Manager* component applies the algorithms for determining other component dependencies for the given input. Similarly, the *Scenario Dependency Analyzer*, given the scenario and the desired dependency type to be analyzed, applies the necessary algorithms to produce the resulting output set.

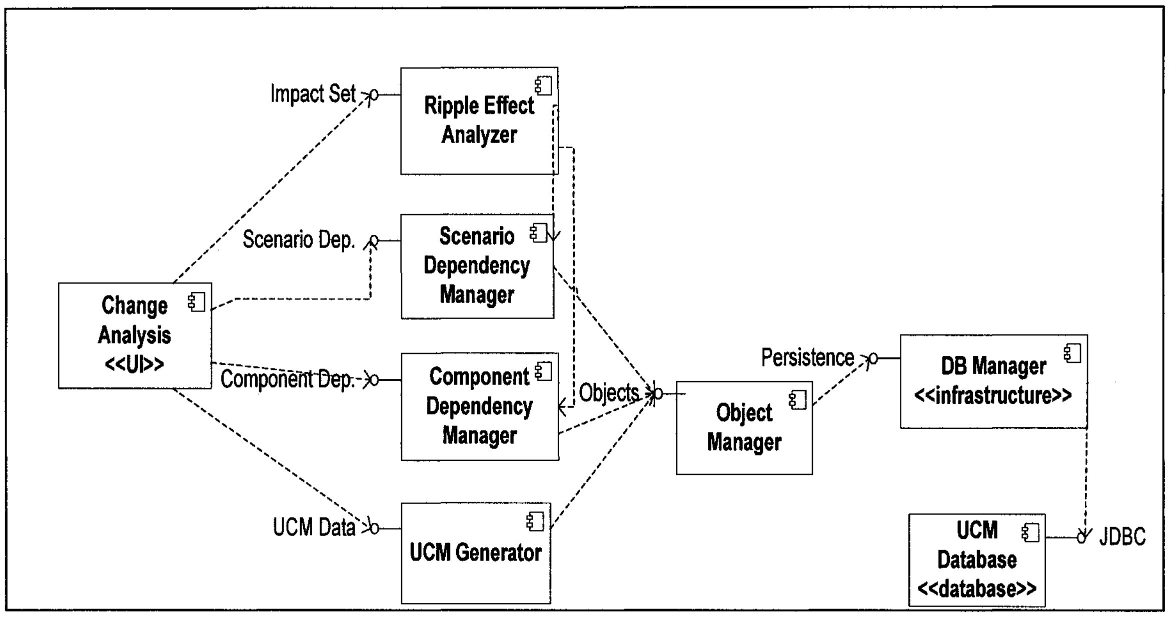


Figure 5-1: UCM Analyzer Component Diagram

The *Ripple Effect Analyzer* determines the impact set for a specific UCM element. If the impact set for a component is desired, given the initial component, it recursively makes use of the *Component Dependency Manager* unit to generate a complete impact set. The same concept is applied for the scenario impact set with interactions with the *Scenario Dependency Manager* unit. If the impact set for a UCM domain element is favored, then the *Ripple Effect Analyzer* handles the implementation of the necessary algorithms for generating the set.

5.2.2 Scenario XML File Parsing

Each scenario file contains a hierarchy of related elements as shown in Figure 5-2. A UCM element contains groups which in turn contains scenarios. Scenarios are comprised of conditions and executed UCM domain elements, with the latter containing the component by which it is bound. We parse each scenario file to extract the elements, their attribute values, and their relationship information using the following logic. The attribute values of each XML element are those provided by the UCM creator or UCMNav to identify the UCM element. (For clarity XML file elements and attributes referenced are written in italics).

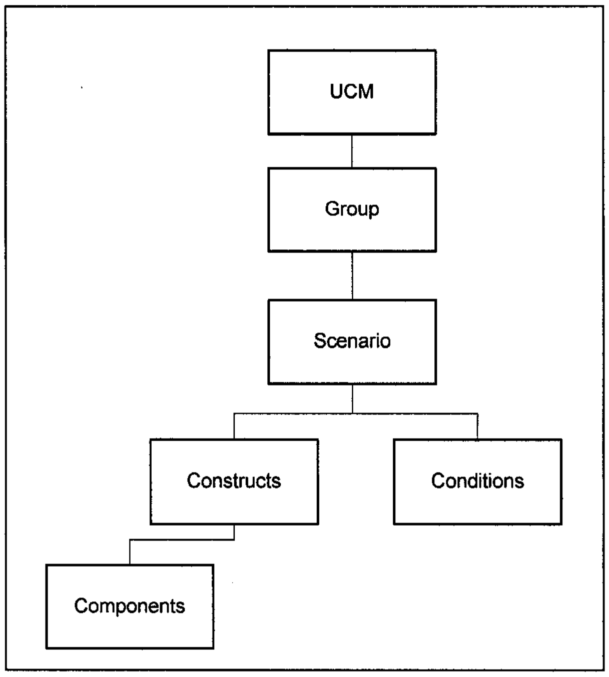


Figure 5-2: Scenario XML File Element Relationship Hierarchy

The UCM for the current scenario file is identified by its filename (*ucm-file* attribute) which we assume is unique among UCMs. A UCM can contain several defined groups; for each new group that is identified (distinguished by its unique *group-id* attribute), a relationship is created with its UCM. A scenario can only be a part of one group forcing a relationship between each scenario and its specified group. Individual scenarios are identified by their *scenario-definition-id* attribute.

The ordered UCM domain elements (*<do>* elements*)* form the scenario's execution sequence and allow for the creation of a relationship between each domain element and the scenario. Scenario sequence information is preserved by creating a previous-next relationship between the ordered UCM domain elements. Path concurrency is preserved with by creating objects to represent the *<par>* and *<seq>* elements that provide structure to the scenario sequence in the XML file (visible in the sample file in Figure 5-3). A parent-child relationship is created between each concurrency element and the UCM elements that it binds.

Due to the fact that UCM domain elements may be contained within several scenarios, it is necessary to avoid duplicates within the UCM by validating the uniqueness of the value of the *hyper-edge-id* attribute of each element. UCMNav applies a unique *hyper-edge-id* to each modeled element to identify the element within UCM. The first time an element is parsed, it is represented in UCM Analyzer while successive identifications simply cause the creation of a relationship between the existing representation and the scenario currently being parsed. Contrarily, domain elements are bound by only one component whose information is contained within an attribute of its XML element. Components are uniquely identified by both their component id and role name (multiple instances of the same component within a UCM are given the same component id but different role names). As with domain elements, each component is only represented once in UCM Analyzer, causing only relationships to be created between existing components and successive elements that they bind.

```xml
<?xml version='1.0' standalone='no'?>
<!DOCTYPE scenarios SYSTEM "scenarios1.dtd">

<scenarios date ="Thu Jun  9 08:15:26 2005" ucm-file = "SDLforum10.ucm" design-name = "SDLforum10" ucm-design-version = "134">
  <group name = "TeenLine"  group-id = "3" >
    <scenario name = "TLActivePINInvalid" scenario-definition-id = "4" >
      <seq>
        <par>
          <seq>
            <do hyperedge-id="0" name="req" type="Start" component-name = "User" component-id= "1" component-role= "Orig" />
            <do hyperedge-id="50" name="start" type="Connect_Start" component-name = "Agent" component-id= "0" component-role= "Orig" />
            <do hyperedge-id="55" name="InitFeatures" type="Resp" component-name = "Agent" component-id= "0" component-role= "Orig" />
            <condition label="TeenLine" expression ="chkTL&amp;amp;(!chkOCS)" hyperedge-id="65" />
            <do hyperedge-id="66" name="start" type="Connect_Start" component-name = "Agent" component-id= "0" component-role= "Orig" />
            <do hyperedge-id="68" name="checkTime" type="Resp" component-name = "Agent" component-id= "0" component-role= "Orig" />
            <condition hyperedge-id="70" label="[Active]" expression ="TLactive" />
            <do hyperedge-id="72" name="getPIN" type="Timer_Set" />
          </seq>
          <seq>
            <do hyperedge-id="91" name="PIN-entered" type="Start" component-name = "User" component-id= "1" component-role= "Orig" />
            <do hyperedge-id="93" type="Trigger_End" component-name = "Agent" component-id= "0" component-role= "Orig" />
          </seq>
        </par>
        <do hyperedge-id="72" name="getPIN" type="Timer_Reset" />
        <do hyperedge-id="89" name="checkPIN" type="Resp" component-name = "Agent" component-id= "0" component-role= "Orig" />
        <condition hyperedge-id="74" label="[notPINvalid]" expression ="!PINvalid" />
        <do hyperedge-id="82" name="deny" type="Resp" component-name = "Agent" component-id= "0" component-role= "Orig" />
        <do hyperedge-id="78" name="fail" type="Connect_End" component-name = "Agent" component-id= "0" component-role= "Orig" />
        <do hyperedge-id="54" name="fail" type="Connect_End" component-name = "Agent" component-id= "0" component-role= "Orig" />
        <do hyperedge-id="4" name="notify" type="End_Point" component-name = "User" component-id= "1" component-role= "Orig" />
      </seq>
    </scenario>
  </group>
</scenarios>
```

Figure 5-3: Example Scenario XML File Contents

Finally, scenario sequences can also contain global variable values that represent selection conditions during a path execution. These conditions are located at the point in the sequence where the selection occurs. Each detected condition (uniquely identified by its *label* and *expression* attributes) is represented once and a relationship is created between the condition and any scenarios that make use of these variables. Pre- and post-conditions are not included in the generated UCMNav scenario XML file. This information is manually extracted from the original UCM design (the *.xml.ucm file) file created by UCMNav. Relationships are then created between global variables and their values. As well, relationships are created between these global variable values and the scenarios that make use of them.

It should be noted that not all UCM domain elements as defined by Hassine et al. in [HAS04] are included in the UCM Analyzer representation of the model due to the limited contents of the scenario XML files generated by UCMNav. Specifically, any path selection domain elements such as OR-forks and OR-joins are not included in the UCM domain elements, since scenario alternatives are resolved before the scenario is exported to the XML file. AND-joins and AND-forks, although preserved in the scenario trace are not actual domain elements in the scenario XML file, thus, they are not included in the scope of this research either.

5.2.3 Object Manager Class Diagram

The static structure of the *Object Manager* component is shown in Figure 5-4 to provide a better understanding of the relationships between the base classes of UCM Analyzer. The properties and methods of the classes have been omitted in an effort to maintain the focus on the relationships.

As shown in Figure 5-4, the scenario class is the core part of the application. The *Scenario* class is used by the *Group* class to create all the scenarios that are part of a group, which, in turn, is related to the *UCM* class. The scenario instance relates to all the *BooleanVariableValue* objects that form its execution state. These values comprise one half of an instance of a *BooleanVariable* object, so the related value is either TRUE or FALSE.

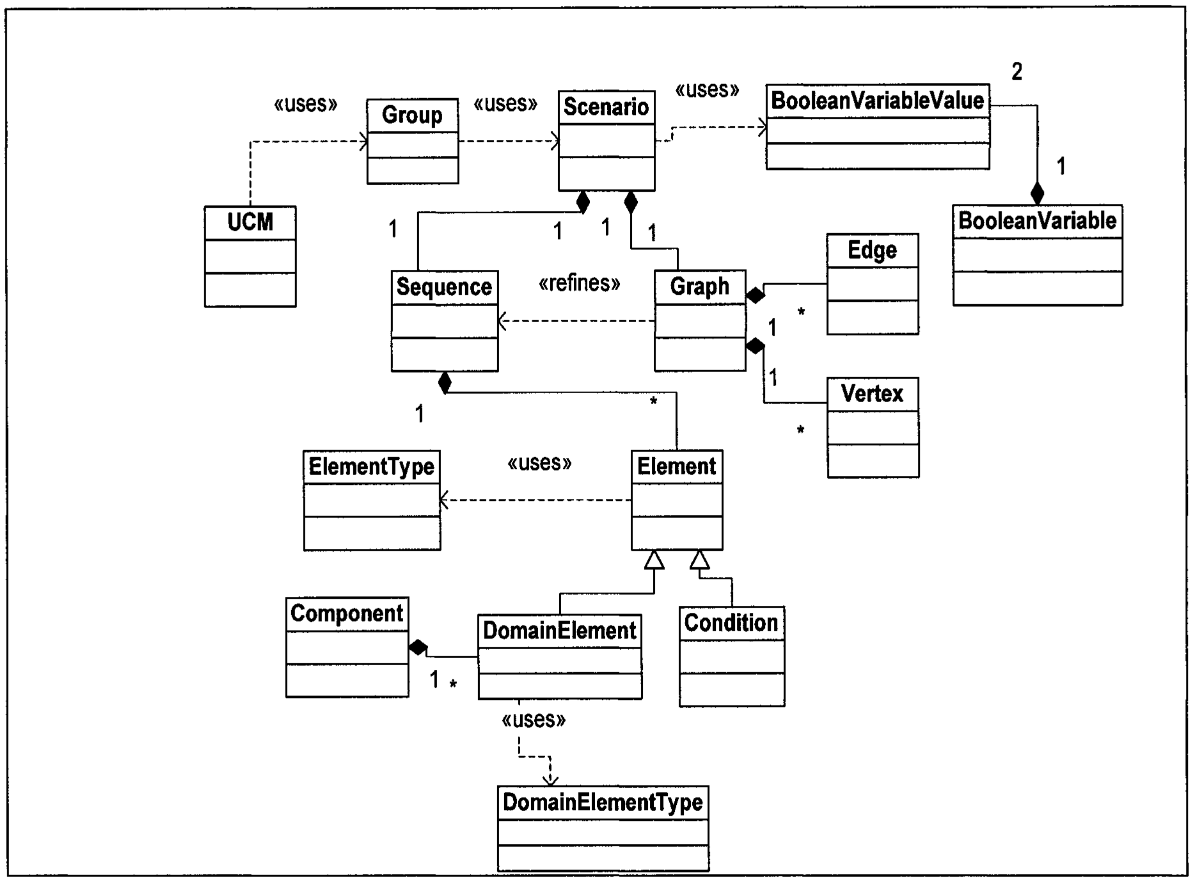


Figure 5-4: Class Structure of Objects Manager

A scenario is composed of exactly one sequence and one graph which is a refinement of that sequence. A *Sequence* object is a tree structure of the four types of elements that may be contained in a sequence: *domain element*, *condition*, *par*, and *seq*. The <*seq*> and <*par*> elements from the XML file are kept to maintain the structure for parallel sequences. In order to create this sequence tree, when the XML file is parsed, the database stores a relationship to its parent element for each element within the sequence.

The *Graph* instance of a given *Scenario* object refines the sequence by removing the *seq*, *par*, and *condition* element types from the sequence while creating a graph from the tree of (possibly) parallel sequences to facilitate forward and backward traversal along all the scenario's paths. The graph's vertices are the domain elements executed by the scenario and the edges are the transitions among these elements.

The *Element* class is refined for the domain element and condition elements types with the *Domain element* and *Condition* classes, respectively. A *Domain Element* is part of one *Component* and each domain element has a type defined by *DomainEelementType*.

This chapter described in detail the overall implementation of the UCM Analyzer and how the change impact analysis at the UCM level is performed using the tool. The next chapter describes an initial case study to illustrate the use of the UCM Analyzer for change impact analysis on an existing UCM.

6. Case Study: Simple Telephony System

The purpose of the presented case study is to provide an initial proof of concept of the presented approach. In what follows we will apply the approaches discussed to an existing UCM in order to show that it is possible to apply change impact analysis techniques to UCMs. In particular, this brief study will select various elements of the UCM and apply the different aspects of our approach to produce dependency and ripple effect information.

6.1 UCM Details

The Simple Telephony System UCM, presented in [MIG01] and referenced in [HAS04], describes the simple connection phase of an agent-based telephone system. The UCM describes the basic call request sequence with the features of call screening and call display. Figure 6-1 shows the root UCM that includes four components (originating and terminating users and agents) and two static stubs. Static stub *Sorig* contains the *Originating* plug-in of Figure 6-2a, while *Sterm* contains the *Terminating* plug-in of Figure 6-2c. Each stub also contains a dynamic stub, *Sscreen* and *Sdisplay*, respectively, which contain either the default plug-in (Figure 6-2e) or their corresponding feature plug-in shown in Figure 6-2b and Figure 6-2d.

Sscreen Plug-ins:

- *Originating Call Screening (OCS)* – implements a call screening feature that either denies or allows a call.

▪ *Default* – used in the case where the caller does not subscribe to any originating features.

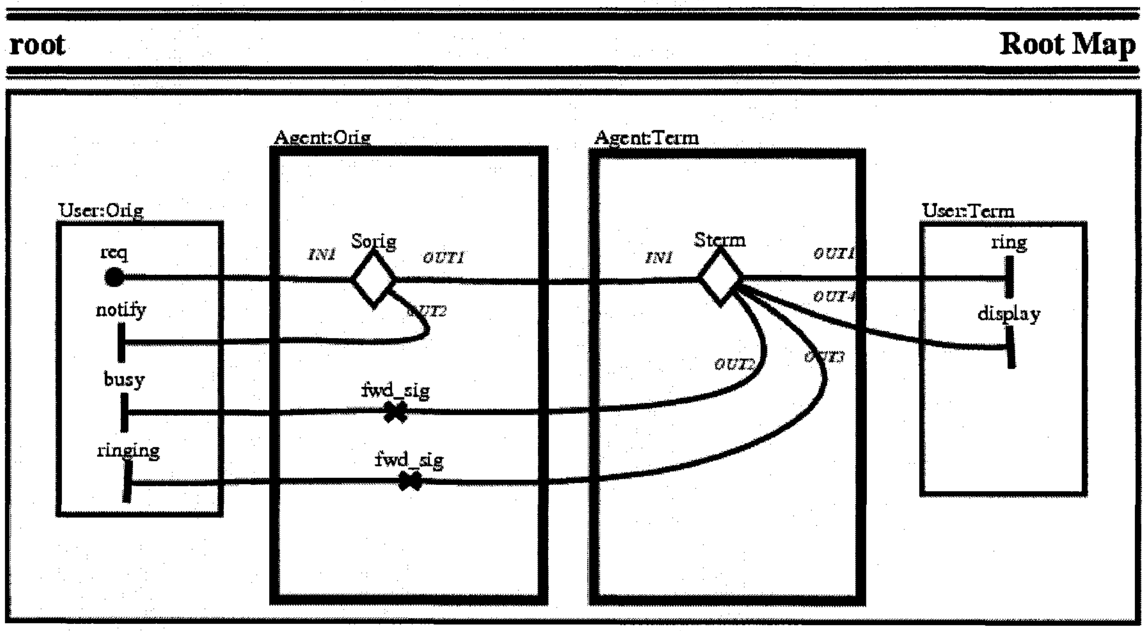


Figure 6-1: Simple Telephony System Root Map

Sdisplay Plug-ins:

▪ *Call Number Delivery (CND)* – implements a call display feature where the number of the originating caller is displayed while ringing.

▪ *Default* – used in the case where the user has not subscribed to any terminating features.

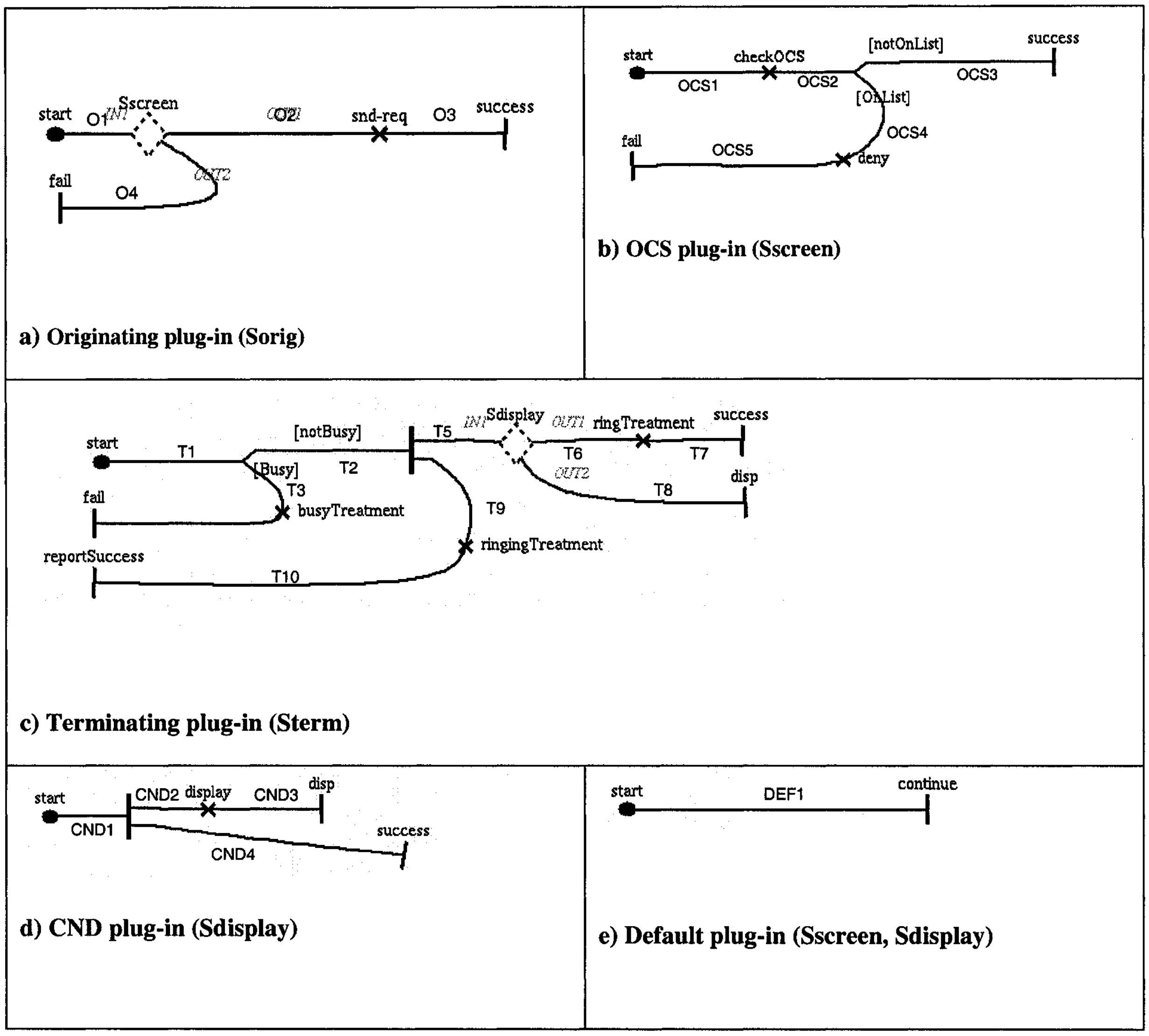


Figure 6-2: Simple Telephony System Plug-ins

Further, the start and end points of the plug-ins in Figure 6-2 are not triggered by external events but are connectors to the input/output segments of their parent stubs. The bindings between stubs and their plug-ins and the conditions that govern the selection of the plug-in map, described in [HAS04], are arranged in Table 6-1.

Stub	Plug-in Map	Condition(s)	Bindings
Sorig	Originating	TRUE	{(IN1, start), (OUT1, success), (OUT2, fail)}
Sscreen	OCS	subOCS	{(IN1, start), (OUT1, success), (OUT2, fail)}
Sscreen	Default	!subOCS	{(IN1, start), (OUT1, continue)}
Sterm	Terminating	TRUE	{(IN1, start), (OUT1, success), (OUT2, fail), (OUT3, reportSuccess, (OUT2, disp)}
Sdisplay	CND	subCND	{(IN1, start), (OUT1, success), (OUT2, display)}
Sdisplay	Default	!subCND	{(IN1, start), (OUT1, continue)}

Table 6-1: Plug-in Input/Output Bindings

6.2 Simple Telephony System in UCM Analyzer

In order to create relationships of the Simple Telephony System in UCM Analyzer, we populated the system's scenarios by parsing their respective XML files. Additionally, we manually added to UCM Analyzer information about the Boolean global variables obtained from the *Simpletelephonesystem.xml.ucm* file that UCMNav created to store the details of the created UCM. The Boolean values related to each scenario were obtained from the exported Encapsulated PostScript (.eps) file that provides meta-information about the UCM model and its scenarios.

Table 6-2 details the scenarios that were parsed by UCM Analyzer to support further analysis. The *Scenario ID* field contains the unique identifier for the scenario provided by UCM Analyzer whereas the group and scenario names are those that have been extracted from the scenario XML file as provided at the time of scenario definition in UCMNav. Each scenario has an associated value of true (T) or false (F) for each global variable or no defined value (/). Please note that the complete UCM for the Simple

Telephony System includes an additional plug-in and several scenarios that we have omitted from the model in order to simplify the case study.

System Scenarios							
	Scenario	Scenario	Global Variables				
Group Name	ID	Name	Busy	OnOCSList	subCND	subOCS	subTL
Basic Call	1	BCbusy	T	/	F	F	F
	2	BCsuccess	F	/	F	F	F
CND	3	CNDbusy	F	/	T	F	F
	4	CNDdisplay	T	/	T	F	F
OCS	7	OCSbusy	T	F	F	T	F
	8	OCSdenied	F	T	F	T	F
	9	OCSsuccess	F	F	F	T	F
FI_OCS_CND	5	OCS_CNDbusy	T	F	T	T	F
	6	OCS_CNDOnList	T	T	T	T	F
	10	OCS_CNDdisplay	F	F	T	T	F

Table 6-2: Simple Telephony System Scenarios

The UCM Analyzer representation of the UCM also contains the components and the domain elements they bind. The Simple Telephony System implements two instances of a component, causing the names of the components to be the same, but their role names to differ. This results in the representation by UCM Analyzer for all four components shown in the root map (Figure 6-1). Domain elements within the plug-in maps are bound to the components that contain them. For example, the elements of *Default* plug-in (Figure 6-2e) actually have two instances that are bound to different components since the plug-in may be substituted for either the *Sscreen* or the *Sdisplay* stub.

Component Name	Component Role	Number of Domain Elements	Number of Interfaces	Execution Scenarios
User	Orig	4	4	all 10
Agent	Orig	14	5	all 10
Agent	Term	14	5	1, 2, 3, 4, 5, 7, 9, 10
User	Term	2	2	2, 4, 9, 10

Table 6-3: Simple Telephone System Components

Table 6-3 summarizes each component's relationship information as follows:

- *User:Orig* – contains 4 domain elements (shown in Figure 6-1) and all of them act as interfaces for access to this component. Domain elements bound by this component are executed by all ten of the scenarios listed in Table 6-1.
- *Agent:Orig* – contains 14 domain elements where 5 act as interfaces. The domain elements bound by this component can be seen in the following plug-in maps: Figure 6-2a, Figure 6-2b, and Figure 6-2e. Domain elements bound by this component are executed by all ten of the scenarios listed in Table 6-1.
- *Agent:Term* - contains 14 domain elements where 5 act as interfaces. The domain elements bound by this component are shown in the following plug-in maps: Figure 6-2c, Figure 6-2d, and Figure 6-2e. The listed scenarios execute the domain elements bound by this component.
- *User:Term* – contains only 2 domain elements (shown in Figure 6-1) that both act as interfaces. The four scenarios listed execute these either or both of these domain elements.

Furthermore, each domain element has an associated domain element type (all possible types are shown in Table 6-4) that classifies its role in the UCM. As well, all the domain elements extracted from the scenario files are classified as *do* element types, while the selection condition elements contained within the file are classified as *condition* element types.

Type Name	UCM Name	Description
Start	Start	A start ucm domain element
Responsibility	Resp	A ucm responsibility
End Point	End_Point	A UCM scenario end point
Waiting Place Enter	WP_Enter	When PT gets to a waiting place
Waiting Place Leave	WP_Leave	After a waiting place is visited
Plugin Connect Start	Connect_Start	Start point of a plugin
Plugin Connect End	Connect_End	End point of a plug-in
Trigger End	Trigger_End	Connected end point
Timer Set	Timer_Set	A timer is set
Timer Reset	Timer_Reset	A timer is reset
Timeout	Timeout	A timeout domain element

Table 6-4: UCM Domain Element Types

6.3 Scenario Dependencies

In order to demonstrate the scenario dependency outputs provided by the UCM Analyzer, we selected the scenario *OCSsuccess* as input for dependency analysis.

6.3.1 Functional Dependency

Querying UCM Analyzer for the scenarios that are functionally dependent with *OCSsuccess* returns a list of all the scenarios that are in its group. Table 6-5 displays the information returned by UCM Analyzer – the other scenarios that are in the same UCM group as the given scenario.

Functional Dependency				
	Scenario ID	**Scenario Name**	**Scenario Group**	**UCM Group ID**
Input Scenario:	9	OCSsuccess	OCS	2
Functionally Dependent Scenarios:				
	8	OCSdenied	OCS	2
	7	OCSBusy	OCS	2

Table 6-5: Functional Dependencies for *OCSsuccess*

6.3.2 Global Variable Dependency

Global variable dependency analysis returns scenarios related to *OCSsuccess* with respect to its conditions. Previously we described four types of global variable dependencies but we limit our case study to improve readability to two types of variable dependency analysis for *OCSsuccess.* The results of the analysis are shown in Table 6-6.

The *Global Variable Value* dependent scenarios listed share the values of any of the input scenario's global variables, while the *Global Variable Value Alternate* scenarios use the opposite value of the global variable to that used by the input scenario.

Global Variable Dependency		
	Scenario ID	**Scenario Name**
Input Scenario:	9	OCSsuccess
Global Variable Value Dependent:		
	5	OCS_CNDbusy
	10	OCS_CNDdisplay
	2	BCsuccess
	1	BCbusy
	8	OCSdenied
	6	OCS_CNDOnList
	3	CNDbusy
	4	CNDdisplay
Global Variable Value Alternates		
	6	OCS_CNDOnList
	3	CNDbusy
	4	CNDdisplay
	10	OCS_CNDdisplay
	2	BCsuccess
	1	BCbusy
	7	OCSbusy

Table 6-6: Global Variable Dependencies for *OCSsuccess*

6.3.3 Execution Dependency

The notion of execution dependency aims to identify the scenarios that share component or domain elements (depending on the level of detail desired). Due to the fact that all the scenarios of the Simple Telephony System begin at the start point *req*, UCM Analyzer will compute all scenarios to be both component and domain element execution dependent with *OCSsuccess* based on the domain element *req*. So, in order to show some unique data, we requested execution dependent scenarios for a specific component and a specific domain element.

Component Specific Execution Dependent		
	ID	Name
Input Scenario:	9	OCSsuccess
Input Component:	4	User:Term
Dependent Scenarios:		
	2	BCsuccess
	4	CNDdisplay
	10	OCS_CNDdisplay

Table 6-7: Component Specific Execution Dependency for *OCSsuccess*

The returned results for execution dependency analysis with respect to component *User:Term* are shown in Table 6-7. The results indicate that the listed scenarios are component execution dependent with *OCSsuccess*. In Table 6-8 the scenarios that are domain execution dependent with OCSsuccess with respect to domain element *snd-req* are listed.

Responsibility Specific Execution Dependent		
	ID	Name
Input Scenario:	9	OCSsucess
Input Responsibility:	9	snd-req
Dependent Scenarios:		
	1	BCbusy
	2	BCsuccess
	3	CNDbusy
	4	CNDdisplay
	5	OCS_CNDbusy
	7	OCSbusy
	10	OCS_CNDdisplay

Table 6-8: Domain Element Execution Dependency for *OCSsuccess*

6.4 Component Dependency

Component dependency information is obtained by making use of the scenario sequences that execute the domain elements bound to components. The Simple Telephony System contains only four components whose interactions are shown in Figure 6-1. UCM Analyzer organizes these dependencies into forward and backward sets.

Table 6-9 details all the dependencies of input component *Agent:Orig*. Evidently, components *Agent:Term* and *User:Orig* are both forward and backward dependent with respect to *Agent:Orig*. For reference purposes, we added to the table the interfaces of the respective components that cause this dependency. These are specified in the *Interface Transitions* field. If a component is forward dependent with the given component then the transition (start, end) means that the "start" domain element is bound to the input component while the "end" domain element is bound to the dependent component. The inverse is true for the backward dependency.

Component Dependency				
	ID	**Name**	**Role**	**Interface Transitions (start, end)**
Input Component:	2	Agent	Orig	
Forward Dependent Components:				
	3	Agent	Term	{(success, start)}
	1	User	Orig	{(fwd_sig, ringing), (fail, notify), (fwd_sig, busy)}
Backward Dependent Components:				
	1	User	Orig	{(req, InitFeatures)}
	3	Agent	Term	{(reportSuccess,fwd_sig), (fail, fwd_sig)}

Table 6-9: Component Dependencies for *Agent:Orig*

Table 6-10 shows the dependencies for *Agent:Orig* with respect to the scenario *OCSdenied* in which the caller is on the screening list and has been denied. The additional input of the component limits the resulting dependency list, identifying only *User:Orig* as being both forward and backward dependent.

Scenario Specific Component Dependency				
	ID	**Name**	**Role Name**	**Interface Transitions (start,end)**
Input Component:	2	Agent	Orig	
Input Scenario:	8	OCSdenied		
Forward Dependent Components:				
	1	User	Orig	{(fail, notify)}
Backward Dependent Components:				
	1	User	Orig	{(req, InitFeatures)}

Table 6-10: Scenario Specific Component Dependency

6.5 Ripple Effect Analysis

Ripple effect analysis aims to determine what other parts of the system are affected by a change to the given component or domain element. In what follows we perform ripple effect analysis on different elements and at two different levels of abstraction for the Simple Telephony System.

6.5.1 Component Ripple Effect Analysis

Assuming a proposed change to component *Agent:Orig*, Table 6-11 lists the forward and backward ripple effects. The forward ripple set implies that from *Agent:Orig* there are paths that lead to the other three components (forward ripple effects). Conversely, only paths from *Agent:Term* and *User:Orig* lead to *Agent:Orig* (backward ripple effects).

Component Ripple Effect Analysis			
	ID	Name	Role Name
Input Component:	2	Agent	Orig
Forward Ripple Effects:			
	1	User	Orig
	4	User	Term
	3	Agent	Term
Backward Ripple Effects:			
	3	Agent	Term
	1	User	Orig

Table 6-11: Component REA for *Agent:Orig*

6.5.2 Domain Element Ripple Effect Analysis

For a more detailed view, we chose to obtain ripple effect information for the domain element *send-req* that is bound by the component *Agent:Orig*. The scope of the analysis if further refined to focus on the scenario *OCSsuccess*. Table 6-12 summarizes the forward and backward ripple effect sets for the provided criteria. The domain element name, its type, and the component to which it is bound are specified.

Scenario Specific Domain Element Ripple Effect Analysis				
	ID	Name	Domain Element Type	Component ID
Input Domain Element:	9	snd-req	Responsibility	2
Input Scenario:	9	OCSsuccess		
Forward Ripple Effects:				
	10	success	Plugin Connect End	2
	11	start	Plugin Connect Start	3
	20	start	Plugin Connect Start	3
	21	continue	Plugin Connect End	3

	ID	Name	Domain Element Type	Component ID
	22	ringTreatment	Responsibility	3
	23	success	Plugin Connect End	3
	24	ring	End Point	4
	25	ringingTreatment	Responsibility	3
	26	reportSuccess	Plugin Connect End	3
	27	fwd_sig	Responsibility	2
	28	ringing	End Point	1
Backward Ripple Effects:				
	40	success	Plugin Connect End	2
	38	checkOCS	Responsibility	2
	37	start	Plugin Connect Start	2
	4	InitFeatures	Responsibility	2
	3	start	Plugin Connect Start	2
	2	req	Start	1

Table 6-12: Domain Element REA for *send-req* and *OCSsuccess*

6.6 Discussion of the Results and Limitations

The presented results produced by UCM Analyzer for the Simple Telephony System show that an automatic implementation of our novel approaches for impact analysis at the UCM level is possible.

The results for our defined UCM scenario dependency analysis approach show clearly that for a given scenario and a selected dependency type, it is possible to determine the scenarios that are related to the given criteria. Furthermore, component dependency analysis also accurately (according to our definitions) identifies those components that are forward and backward dependent for the given input. Finally, ripple effect analysis allows the analysis and identification of those UCM elements (components or domain elements) that are reachable (affected) from the input (forward ripple effects) and those that lead to the specified input (backward ripple effects).

The results do not make any claims as to what extend the provided information is applicable in determining real change impacts. To validate this we would require actual changes and their implementation to compare the estimated dependencies and ripple effects determined by our approach to the actual impact sets [ARN93]. Due to the unavailability of this information, we cannot make any conclusions about the effectiveness of the presented approach. However, as mentioned, this particular goal was not the intent of our research.

In spite of our encouraging early results, our implementation does have some limitations. The following describes some of the issues that currently exist in UCM Analyzer.

Although it would have been ideal to relate the *Condition* and *BooleanVariableValue* classes, since they essentially are referring to the same UCM condition, the lack of consistency in the naming of conditions and the global variable they represent made this impossible. Global (Boolean) variables that apply to the UCM and their relationships to scenarios must be entered into UCM Analyzer manually in order to provide the tool with all the global variable values that apply to a specific scenario (the XML file for the scenario only contains selection conditions). The documents from which we obtain the global variable and their relationship details – the *.xml.ucm file and the *.ps file outputted by UCMNav, do not have the same names for the global variable values as that of the scenario XML file. So to avoid making incorrect associations we simply omitted the relationship between these two elements.

At the current stage of development, UCM Analyzer does not implement the scenario *Containment Relationship* defined as part of our approach. The reasons for this are discussed as part of our future investigation.

Finally, since our approach requires the input of the initial change set, some additional work is required to use our approach for change impact analysis. That is, some effort is still required to identify the initial impacts of the proposed change before our approach can be used to identify the dependencies and ripple effects of the changed element.

7. Conclusion and Future Investigations

In this research, we demonstrated how dependency and ripple effect analysis techniques can be applied to UCMs for use during change impact analysis. The proposed approach introduced the concept of UCM scenario and component dependencies, providing definitions and algorithms for their determination. As well, definitions and algorithms were presented for UCM ripple effect analysis of components and domain elements [HEW05]. Table 7-1 provides a summary of our proposed approach by making use of criteria from the change impact analysis comparison framework presented in [ARN93] and discussed in section 2.4.5. Further, we described our developed tool that implements our defined approach and supports automation. The applicability of the presented approach was exemplified using an existing UCM as a case study. The initial case study helped to validate both our theoretical and implementation approach.

A major contribution of this research is the novel idea of using UCM for change impact analysis. While many approaches focus on change impact analysis at the source code or detailed model level, our proposed approach raises the level of analysis to that of requirements specification, allowing the scope of the change to be analyzed. Furthermore, in our proposed approach, analysis may be performed at different levels of granularity, providing flexibility in analysis.

In [HAS05], we present an extension of our proposed approach. Specifically, the domain element ripple effect analysis technique is further developed through the application of the UCM slicing algorithm presented by Hassine et al. in [HAS04]. The produced impact set confines to the defined formal definition of the slicing approach.

IA Application	IA Parts				
Object Domain	**Internal Model**	**Impact Model**	**Impact Tracing Algorithm**	**Repository**	**Repository Features**
UCM requirements specifications	UCM scenarios, components, domain elements	UCM scenario dependency graph	transitive closure based on scenario specifications	relational database	load, browse

Table 7-1: Proposed Approach Summary

Another major contribution of this work is the developed tool. UCM Analyzer implements the discussed methodologies and effectively automates the analysis process. We have shown that this tool can be effectively used to extract dependency and ripple effect information from UCMs.

As part of future investigation a complete integration of the implementations of UCM Analyzer into UCMNav would be desirable. This would remove the need to recreate the UCM relationships from exported data.

Another issue for future work is the need to further evaluate and validate our approach. There is a need for further validation of the effectiveness in identifying the actual ripple effects and dependencies as well as their precision. One possible validation approach that we are currently exploring is the extension of our defined scenario containment dependency through the use of concept analysis.

An interesting research area would be to assess how our approach can be applied to feature interaction. That is, for a changed feature, assess whether our proposed approach provides sufficient information for change impact analysis.

Finally, synchronization and notion of time are not considered in this research. Though UCMs do support the modeling of time and concurrency, current research that is attempting to formalize UCMs has not modeled the notion of time.

8. References

[AGR91] Hiralal Agrawal, Richard A. DeMillo, Eugene H. Spafford, "Dynamic Slicing in the Presence of Unconstrained Pointers," *Symposium on Testing, Analysis, and Verification* 1991, pp. 60-73.

[AMY01] Daniel Amyot, Gunter Mussbacher, "MiniTutorial on UCMs Notation," http://www.usecasemaps.org/urn/cascon01/UCMintro.pdf.

[AMY01b] Daniel Amyot, Gunter Mussbacher, "Bridging the Requirements/Design Gap in Dynamic Systems with Use Case Maps (UCMs)," *International Conference on Software Engineering* 2001, pp. 743-744.

[AMY01c] D. Amyot, A. Eberlein, "An Evaluation of Scenario Notations for Telecommunication Systems Development," *9th Int. Conference on Telecommunication Systems (9ICTS)*, Dallas, USA (March 2001).

[AMY02] Daniel Amyot, "UCM Scenarios and Path Traversal," SG17, Geneva, 2002, www.itu.int/itudoc/itu-t/com17/urn/urnp2_pp7.ppt

[AMY03] Daniel Amyot, "Introduction to the User Requirements Notation: Learning by Example," *Computer Networks* 42, 3 (Jun. 2003), pp. 285-301.

[AMY03b] Daniel Amyot, Xiangyang He, Yong He, Dae Yong Cho, "Generating Scenarios from UCM Specifications," *International Conference on Quality Software (QSIC)*, 2003, pp.108-115.

[ARN93] R. S. Arnold, S. A. Bohner, "Impact Analysis - Towards a Framework for Comparison", *Proc. Conf. Software Maintenance*, 1993, pp. 292-301.

[BAI02] Xiaoying Bai, Wei-Tek Tsai, Ke Feng, Lian Yu, Ray J. Paul, "Scenario-Based Modeling and Its Applications," *WORDS* 2002: 253-260.

[BEC93] J. Beck and D. Eichmann, "Program and Interface Slicing for Reverse Engineering," *IEEE/ACM 15th Conference on Software Engineering (ICSE'93)*, 1993, pp. 509-518.

[BEN00] K. H. Bennett, V.T. Rajlich, "Software Maintenance and Evolution: a Roadmap," *In Proceedings of the Conference on the Future of Software Engineering (Limerick, Ireland, June 04 - 11, 2000), ICSE '00.* pp. 73-87.

[BIL04] Edward Billard, "Patterns of Agent Interaction Scenarios as UCMs," *IEEE Trans. Syst., Man, Cybern.*, pp. 1933-1939, 34B:4:2004.

[BLA01] Sue Black, "Computing Ripple Effect for Software Maintenance," *Journal of Software Maintenance* 13(4), 2001, pp. 263-279.

[BOH02] Shawn A. Bohner, "Software Change Impacts - An Evolving Perspective," *Int'l Conference on Software Maintenance* 2002, pp. 263-272.

[BOH96] S. Bohner and R. Arnold. Software Change Impact Analysis. IEEE Computer Society Press, 1996, pp. 1-26.

[BOJ00] Dragan Bojic, Dusan M. Velasevic, "A Use-Case Driven Method of Architecture Recovery for Program Understanding and Reuse Reengineering," *Proceedings of the Conference on Software Maintenance and Reengineering*, 2000, pp. 23-32.

[BOR01] Francis Bordeleau, Jean-Pierre Corriveau, "On the Importance of Inter-Scenario Relationships," www.usecasemaps.org/urn/cascon01/scenarioRelationships.pdf

[BRE00] K. Breitman, J. C. Sampaio do Prado, "Scenario Evolution: A Closer View on Relationships," *Proceedings. 4th International Conference on Requirements Engineering*, 2000, pp. 95 - 105.

[BRI03] Lionel C. Briand, Yvan Labiche, L. O'Sullivan, "Impact Analysis and Change Management of UML Models," *Int'l Conference on Software Maintenance* 2003, pp. 256-265.

[BRO83] Ruven Brooks, "Towards a Theory of the Comprehension of Computer Programs," *International Journal of Man-Machine Studies*, 18 (6), 1983, pp. 543-554.

[BRU00] Hans de Bruin, Vrije Universiteit, "UCMs a Technique for Communicating Validating Behavioral Aspects of Architectures," http://www.serc.nl/lac/LAC-2001/lac-2000/2-validatie/use%20case%20maps.pdf

[BUH95] R.J.A. Buhr, R.S. Casselman, T.W. Pearce, "Design Patterns with UCMs: A Case Study in Reengineering an Object-Oriented Framework", SCE 95-17, www.usecasemaps.org/pub/dpwucm.pdf

[BUH96] R.J.A. Buhr, "UCMs for Attributing Behavior to Architecture," *Proc. 4th Int'l Workshop on Parallel and Distributed Real Time Systems*, 1996.

[BUH96b] R. J. A. Buhr, R. S. Casselman, Use Case Maps for Object-Oriented Systems, Prentice-Hall, Inc. 1996.

[BUH99] R. J. A. Buhr, “Making Behaviour a Concrete Architectural Concept,” *32nd Annual Hawaii International Conference on System Sciences (HICSS-32)*, 1999.

[CAR99] S. Jeromy Carrière, Steven G. Woods, Rick Kazman, “Software Architectural Transformation,” *Working Conference on Reverse Engineering,* 1999, pp. 13-23.

[CHA98] Melissa P. Chase, Steven M. Christey, David R. Harris, Alexander S. Yeh, “Recovering Software Architecture from Multiple Source Code Analyses,” *Proceedings of the SIGPLAN/SIGSOFT Workshop on Program Analysis For Software Tools and Engineering (PASTE),* 1998, pp. 43-50.

[CHE96] Xiaoping Chen, Wei-Tek Tsai, Hai Huang, Mustafa H. Poonawala, Sanjai Rayadurgam, Yamin Wang, “Omega - an Integrated Environment for C++ Program Maintenance,” *Int’l Conference on Software Maintenance,* 1996, pp. 114-123.

[COX01] Lisa Cox, Harry S. Delugach, David Skipper, “Dependency Analysis Using Conceptual Graphs,” *Supplementary Proc. 9th Int’l. Conference. on Conceptual Structures*, 2001.

[CSWPI] http://web.cs.wpi.edu/~kal/courses/compilers/module4/mysa.html

[FIE95] J. Field, G. Ramalingam, and F. Tip, “Parametric Program Slicing,” *In Proceedings of the 22nd ACM SIGPLAN-SIGACT Principles of Programming Languages (POPL),* 1995, pp. 379-392.

[GAM94] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns – Elements of Reusable Object-Oriented Software, Addison-Wesley, 1994.

[GOR93] Tarak Goradia, “Dynamic Impact Analysis: A Costeffective Technique to Enforce Error-propagation”, *Proc. of the 1993 ACM SIGSOFT Intl. Symposium on Software Testing and Analysis*, 1993, pp. 171- 181.

[HAR98] Mary Jean Harrold, Ning Ci, “Reuse-Driven Interprocedural Slicing,” *Int’l Conference on Software Engineering,* 1998, pp. 74-83.

[HAS03] Ahmed E. Hassan, Richard C. Holt, “ADG: Annotated Dependency Graphs for Software Understanding,” *VISSOFT 2003: 2nd Annual DESIGNFEST On Visualizing Software For Understanding And Analysis*, Amsterdam, 2003.

[HAS04] Jameleddine Hassine, Rachida Dssouli, Juergen Rilling, “Applying Reduction Techniques to Software Functional Requirement Specifications,” *System Analysis and Modeling (SAM)*, 2004, pp.138-153.

[HAS05] Jameleddine Hassine, Juergen Rilling, Jacqueline Hewitt, “Change Impact Analysis for Requirement Evolution using UCMs,” *8th International Workshop on Principles of Software Evolution (IWPSE),* 2005, pp. 81-90.

[HEI98] M. P. Heimdahl, J. M. Thompson, and M. W. Whalen, “On the Effectiveness of Slicing Hierarchical State Machines: A Case Study,” *In Proceedings of the 24th Conference on EUROMICRO - Volume 1*, 1998.

[HEW05] Jacqueline Hewitt, Juergen Rilling, “A Light-Weight Proactive Software Change Impact Analysis Using UCMs,” *ICSM, Proceedings of the Special Session on Software Evolvability*, 2005.

[IEE90] IEEE, *IEEE Standard Glossary of Software Engineering Terminology*, report IEEE Std 610.121990.

[IEE98] IEEE, *IEEE Standard for Software Maintenance*, report IEEE Std 1219-1998.

[JAV04] T. Javed, Manzil-e-Maqsood, Qaiser S. Durrani, “A Study to Investigate the Impact of Requirements Instability on Software Defects,” *ACM Software Engineering Notes*, 29, 3 (May 2004), pp. 1-7.

[KAZ96] Rick Kazman, Gregory D. Abowd, Leonard J. Bass, Paul C. Clements, “Scenario-Based Analysis of Software Architecture,” *IEEE Software* 13(6), 1996, pp. 47-55.

[KIM99] Kyung-Hee Kim, Jai-Nyun Park, Yong-Ik Yoon, “A Graph of Change Impact Analysis for distributed object-oriented software,” *8th IEEE International Conference on Fuzzy Systems (FUZZ-IEEE),* 1999. Volume 2, pp. 1137 - 1141.

[KNE03] Antje von Knethen, Mathias Grund, “QuaTrace: A Tool Environment for (Semi-) Automatic Impact Analysis Based on Traces,” *Int’l Conference on Software Maintenance* 2003, pp. 246-255.

[KOR03] Bogdan Korel, Inderdeep Singh, Luay Tahat, Boris Vaysburg, “Slicing of State-Based Models,” *Int’l Conference on Software Maintenance,* 2003, pp.34-43.

[KOR04] Bogdan Korel, Luay H. Taha, “Understanding Modifications in State-Based Models,” *12th International Workshop on Program Comprehension (IWPC),* 2004, Italy, pp. 246-250.

[KOR90] B. Korel, J. Laski, “Dynamic slicing of computer programs, ” *Journal of Systems and Software*, 13(3), 1990, pp.187-195.

[KUN94] David Chenho Kung, Jerry Gao, Pei Hsia, F. Wen, Yasufumi Toyoshima, Cris Chen, "Change Impact Identification in Object Oriented Software Maintenance," *Int'l Conference on Software Maintenance,*1994, pp. 202-211.

[LAM98] W. Lam, Martin Loomes, "Requirements Evolution in the Midst of Environmental Change: A Managed Approach," *Conference on Software Maintenance and Reengineering,* 1998, pp. 121-127.

[LAM98b] W. Lam, V. Shankararaman, S. Jones, J. Hewitt, C. Britton, "Change Analysis and Management: a Process Model and Its Application Within a Commercial Setting," *Symposium on Application-Specific Systems and Software Technology (ASSET-98),* March 1998, pp. 34 - 39.

[LAM99] W. Lam, Martin Loomes, V. Shankararaman, "Managing Requirements Change Using Metrics and Action Planning," *Conference on Software Maintenance and Reengineering,* 1999, pp. 122-129.

[LAW03] James Law, Gregg Rothermel, Whole Program Path-Based Dynamic Impact Analysis," *International Conference on Software Engineering,* 2003, pp. 308-318.

[LEE00] M. Lee, A. J. Offutt, and R. T. Alexander, "Algorithmic Analysis of the Impacts of Changes to Object-Oriented Software," *Computer Performance Evaluation: Modelling Techniques and Tools, 11th International Conference, TOOLS*, 2000.

[LEH80] M. M. Lehman, "Programs, lifecycles and the Laws of Software Evolution," *Proc. IEEE*, vol. 68, no. 9, September 1980.

[LHA05] Abdelwahab Hamou-Lhadj, Edna Braun, Daniel Amyot, Timothy Lethbridge, "Recovering Behavioral Design Models from Execution Traces," *Conference on Software Maintenance and Reengineering,* 2005, pp.112-121.

[MAL98] Y. Malaiya, J. Denton, "Requirements Volatility and Defect Density," In *Proc. of the 10th International Symposium on Software Reliability Engineering*, pp. 285. 1998.

[MIG01] Andrew Miga, Daniel Amyot, Francis Bordeleau, Donald Cameron, C. Murray Woodside, "Deriving Message Sequence Charts from UCMs Scenario Specifications," *SDL Forum,* 2001, pp.268-287.

[NAK00] Masahide Nakamura, Tohru Kikuno, Jameleddine Hassine, Luigi Logrippo, "Feature Interaction Filtering with UCMs at Requirements Stage," *Feature Interactions in Telecommunications and Software Systems,* 2000, pp.163-178.

[NAN02] Vivek Nanda, Nazim H. Madhavji, "The Impact of Environmental Evolution on Requirements Changes," *Int'l Conference on Software Maintenance,* 2002, pp. 452-461.

[NEL96] M. L. Nelson, "A Survey of Reverse Engineering and Program Comprehension," *ODU CS 551-Software Engineering Survey*, April 19 1996.

[ORS03] A. Orso, T. Apiwattanapong, and M. J. Harrold, "Leveraging Field Data for Impact Analysis and Regression Testing," *In proc. of the 9th ESEC and 10th ACM SIGSOFT Int'l Symp. on Foundations of Software Engineering*, 2003, pp 128-137.

[PAU01] Raymond Paul, "End-to-End Integration Testing," *2nd Asia-Pacific Conference on Quality Software (APAQS),* 2001, pp. 211-222.

[PAU01b] Ray J. Paul, Lian Yu, Wei-Tek Tsai, Xiaoying Bai, "Scenario-Based Functional Regression Testing," *25th International Computer Software and Applications Conference (COMPSAC)*, 2001, pp. 496-501.

[PET02] D. Petriu, M. Woodside, "Software Performance Models from System Scenarios in Use Case Maps", *Computer Performance Evaluation / TOOLS*, 2002, pp.141-158.

[PIG05] Thomas M. Pigoski, "SWEBOK Knowledge Area Description for Software Evolution and Maintenance (version 0.5)," http://www.swebok.org/stoneman/version_0.5/KA_Description_Software_Evolution_Maintenance(Version_0_5).pdf

[QUE94] J-P. Queille, J-F. Voidrot , M. Munro, N. Wilde, "The Impact Analysis Task in Software Maintenance: A Model and a Case Study," *Int'l Conference on Software Maintenance*, 1994.

[RADZI] Eimutis S. Radzius, "A Methodology for the Planning of a Scenario Based Test Program," http://www.usecasemaps.org/pub/test_planning_SBT.pdf

[REN04] Xiaoxia Ren, Fenil Shah, Frank Tip, Barbara Ryder, and Ophelia Chesley, "Chianti: A tool for change impact analysis of Java programs," *Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, 2004, pp. 432-448.

[RIL03] Juergen Rilling, Tuomas Klemola, "Identifying Comprehension Bottlenecks Using Program Slicing and Cognitive Complexity Metric," *International Workshop on Program Comprehension,* 2003, pp. 115-124.

[RUG95] S. Rugaber, "Program Comprehension," *Encyclopedia of Computer Science and Technology* 35(20), 1995, pp.341-368.

[RUS02] J. T. Russell, "Program Slicing for Codesign," *In Proceedings of the Tenth International Symposium on Hardware/Software Codesign (CODES '02)*, 2002, pp. 91-96.

[SET04] Raffaella Settimi, Jane Cleland-Huang, Oussama Ben Khadra, Jigar Mody, Wiktor Lukasik, Chris DePalma, "Supporting Software Evolution through Dynamically Retrieving Traces to UML Artifacts," *7th International Workshop on Principles of Software Evolution (IWPSE)*, 2004, pp. 49-54.

[SNE00] H. M. Sneed, "Source Animation as a Means of Program Comprehension," *International Workshop on Program Comprehension,* 2000, p.179.

[STR96] M.R. Strens, R.C. Sugden, "Change Analysis: A Step towards Meeting the Challenge of Changing Requirements," *IEEE Symposium and Workshop on Engineering of Computer Based Systems (ECBS'96)*, 1996, p. 278.

[TIP96] Frank Tip, Jong-Deok Choi, John Field, G. Ramalingam, "Slicing Class Hierarchies in C++," *Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA),* 1996, pp. 179-197.

[TON03] Paolo Tonella, "Using a Concept Lattice of Decomposition Slices for Program Understanding and Impact Analysis," *IEEE Trans. Software Engineering,* 2003, 29(6), pp. 495-509.

[TSA01] Wei-Tek Tsai, Xiaoying Bai, Ray J. Paul, Weiguang Shao, Vishal Agarwal, "End-To-End Integration Testing Design," *25th International Computer Software and Applications Conference (COMPSAC),* 2001, pp. 166-171.

[TSA03] W. T. Tsai, L. Yu, X. X. Liu, A. Said, Y. Xiao, "Scenario-Based Test Case Generation for State-Based Embedded Systems," http://asusrl.eas.asu.edu/Publications/IPCCC2003.pdf

[UCM] http://www.usecasemaps.org/index.shtml

[WAN96] Yamin Wang, Wei-Tek Tsai, Xiaoping Chen, Sanjai Rayadurgam, "The Role of Program Slicing in Ripple Effect Analysis," *8th International Conference on Software Engineering and Knowledge Engineering (SEKE)*, 1996, pp.369-376.

[WEI81] Mark Weiser, "Program Slicing," *Int'l Conference on Software Engineering,* 1981, pp. 439-449.

[WEI98] K. Weidenhaupt, K. Pohl, M. Jarke, P. Haumer, "Scenarios in System Development: A Report on Current Practice," *IEEE Software*, 1998.

[WU01] Ye Wu, Dai Pan, Mei-Hwa Chen, "Techniques for Testing Component-Based Software," *7th International Conference on Engineering of Complex Computer Systems (ICECCS)*, 2001, pp. 222-232.

[XIA03] Daniel Amyot, Xiangyang He, Yong He, Dae Yong Cho, "Generating Scenarios from UCM Specifications," *International Conference on Quality Software (QSIC)*, 2003, pp.108-115.

[YAC04] S. Yacoub, B. Cukic, H. H. Ammar, "A Scenario Based Reliability Analysis Approach for Component Based Software," *IEEE Transactions on Reliability*, Vol. 53, No. 4, pp. 465-480, December 2004.

[ZHA01] Youtao Zhang, Rajiv Gupta, "Timestamped Whole Program Path Representation and its Applications," *Conference on Programming Language Design and Implementation (PLDI)*, 2001, pp. 180-190.

[ZHA02] Jianjun Zhao, Hongji Yang, Liming Xiang, Baowen Xu, "Change Impact Analysis to Support Architectural Evolution," *Journal of Software Maintenance* 14(5), 2002, pp. 317-333.