# A Requirement Level Modification Analysis Support Framework

Maryam Shiri, Jameleddine Hassine, Juergen Rilling
*Concordia University, Montreal, Canada*
*{ma_shiri,j_hassin, rilling}@cse.concordia.ca*

## Abstract

*Modification analysis is an essential phase of most software maintenance processes, requiring decision makers to perform and predict potential change impacts, feasibility and costs associated with a potential modification request. The majority of existing techniques and tools supporting modification analysis focusing on source code level analysis and require an understanding of the system and its implementation. In this research, we present a novel approach to support the identification of potential modification and retesting efforts associated with a modification request, without the need for analyzing or understanding the system source code. We combine Use Case Maps with Formal Concept Analysis to provide a unique modification analysis framework that can assist decision makers during modification analysis at the requirements level. We demonstrate the applicability of our approach on a telephony system case study.*

***Keywords:*** *Change impact analysis, regression testing, feature interaction, Formal Concept Analysis, Use Case Maps*

## 1. Introduction

While developing software systems, it is rare that an initial system design will correspond completely with the final design or implementation of a system [16]. Ever changing customer needs lead to requirements modifications [4]. The efficient management and execution of these changes are critical to software quality and for managing the evolution of software systems [4]. Maintenance processes (e.g., [11]) have been established to guide both managers and maintainers during typical maintenance and evolution tasks. Common to these process models is that they require some type of modification analysis (MA) prior to committing to a change. Often during MA, managers who are not familiar with the detailed implementation of a system have to determine the feasibility, impact, and cost associated with a change request.

Existing work in change impact analysis and regression testing has focused mainly on identifying changes at the source code level [14] [10]. These source-code-based approaches typically result in an accurate analysis of the change impacts, since the source code represents the final requirements implementation. However, these approaches also tend to be time consuming and require an understanding of both the system requirements and their implementations which makes them less applicable for management and non-technical decision makers.

In this research, we propose a framework for supporting modification analysis for an early detection of change impact and re-testing effort at the requirements level, without the need for programming or implementation knowledge. We illustrate our approach using an existing requirement modeling technique that is being translated into a formal semantics [8] to make the requirements model executable. Traces are created from this model by executing UCM [22] scenarios that are collected and further analyzed using Formal Concept Analysis (FCA) [7]. We introduce three typical modification analysis tasks, which we will revisit throughout the article to illustrate the applicability of our approach. The three analysis task examples are:

1. *Determine impact of a modification request on traces.*
2. *Estimate the test cases that have to be retested as part of a modification request.*
3. *Identify Use case scenarios, which contain feature interactions.*

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

## 2. Related Background

### 2.1 Formal Concept Analysis

Formal Concept Analysis (FCA) [7] is a mathematical approach that dates back to Birkoff in 1940 [3]. FCA is commonly used for representing and analyzing information, by performing logical grouping of objects with common attributes. An FCA *context* is a triple C= (O,A,R) where *O* represents a set of objects and *A,* a set of attributes, with $R \subseteq O \times A$ being a relation among them[17]. We invite the readers to consult [17] for detailed examples. Within the software engineering community, FCA has various applications in program comprehension and maintenance applications and among these applications are reengineering, software component retrieval, identifying objects from legacy code, model restructuring [21], and minimizing test suites [20].

Formal Concept Analysis has gained popularity in recent years, due to the following reasons: (1) programming language independence, (2) the ability to easily define different views (using different object, attribute combinations), (3) tool availability to generate context tables and lattices, and (4) the analysis itself is quite inexpensive, especially compared with other dynamic dependency and trace analysis techniques. It should, however, be noted that FCA does not consider semantic information about the trace content, which limits its applicability for more specific fine-grained type of analysis. Furthermore, identifying a meaningful context for the analysis is not always easy.

### 2.2 Use Case Maps

Use Case Maps (UCM) [6], a modeling technique that is part of a new User Requirements Notation (URN) proposal to the ITU-T [12], has been applied to capture functional requirements in terms of casual scenarios. Since UCMs can represent behavioral aspects at a higher level of abstraction than for example UML diagrams, they are not necessarily bound to a specific underlying structure. They are not intended to replace UML, but to complement it. UCM can visualize an entire system to provide a better understanding of the evolving behavior of complex and dynamic systems such as reactive and distributed systems at the requirement and specification level. It can also provide stakeholders with guidance and reasoning about system-wide functionalities and behavior. Originally introduced to model the behavior of telecommunication systems, applications of UCMs have been extended to other application domains, e.g. web based application, For a more detailed coverage of the UCM notation, we refer the reader to [1][9].

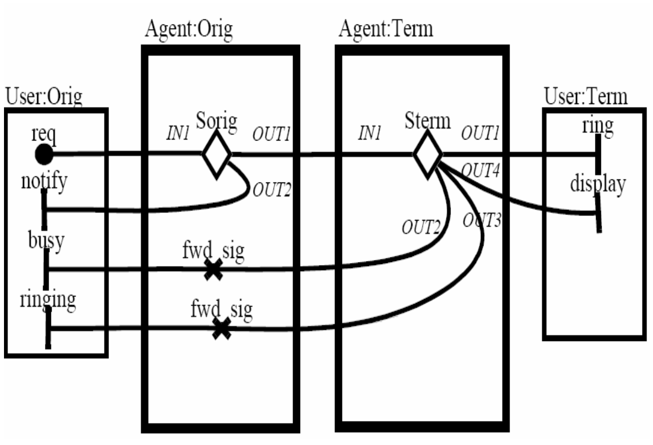

**Fig.1 A simple telephony system (UCM root map)**

#### 2.2.1 UCM Example - A Telephony System

Figure 1 shows the UCM root map of a telephony example originally introduced in [1] and further discussed in [9]. The example describes the connection request phase in an agent based telephony system with user-subscribed features (*Originating Call Screening* and *Call Number Delivery)* containing four components (originating/terminating users and their agents) and two static stubs. Upon the request of an originating user (*req*), the originating agent will select the appropriate user feature (in stub *Sorig*) that could result in some feedback (*notify*). This may also cause the terminating agent to select another feature (in stub *Sterm*), which in turn can cause different results in the originating and terminating users. Stub *Sorig* contains the *Originating* plug-in whereas stub *Sterm* contains the *Terminating* plug-in. These sub-UCMs have their own stubs and plug-ins (Figure 2) corresponding to user-subscribed features and as part of this UCM a set of global variables is defined.

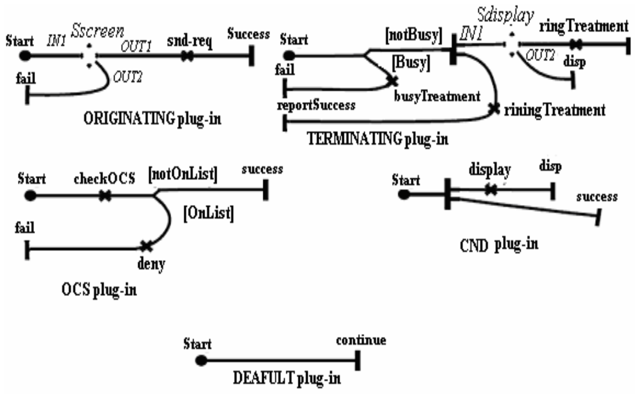

**Fig2. Plug-ins for Simple Telephony Features**

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

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

The plug-ins in *Sdisplay* are:

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

## 2.3 Modification Analysis Techniques

Within the scope of our research, we focus on three analysis techniques to be performed as part of the modification request analysis.

***Change Impact Analysis.*** Impact analysis focuses on identifying these parts of a system that are (potentially) affected by a modification request. Currently, most of the research on change impact analysis focuses on source code [14] and design level analysis [5] using either traceability or dependency analysis [4]. Common to these source code base approaches is that they are computationally expensive and often limited in their applicability by being restricted to a specific programming language.

***Regression Testing.*** Regression testing is based on impact analysis and is used to validate that after a modification is implemented no new errors were introduced in a previously tested code [18] [20]. A r*etest-all* approach that requires re-running all the test cases in the existing test suite is typically too expensive and unnecessary. In most cases, except for the rare event of a major rewrite, changes only affect parts (requirements) of a system. Selective regression testing allows for a reduction of the number of test cases to be re-executed and therefore reducing the cost associated with testing. Regression testing can be informally described as: Given a program P, its modified version P′ and a set of test cases T used previously to test P. Regression testing identifies the subset T′, where T′ $\supseteq$T and provides sufficient confidence in the correctness of P' [18].

***Feature Interaction (FI).*** Often, maintainers have to deal with situations where one feature modifies or subverts the desired operation of another feature or when a system functions incorrectly due to the presence of certain features [13]. Identifying those scenarios and features that are either affected or prone to be affected by a modification request is a challenge especially for large, feature rich systems. Existing work on feature interaction has focused on both the requirements level [15] and source code level [23]. More detailed information on feature interaction analysis techniques can be found in [13], [15], and [23].

## 3. Requirements Modification Analysis

The challenges for management include: 1) determining the potential affect of a modification request on the overall system early in the software maintenance cycle; 2) identifying which scenarios contain which features, and 3) analyzing, whether a feature interacts with other features and determining the testing effort related to a particular change. We now introduce the major activities involved in our approach (Fig 3).

In step 1, we utilize a formalized UCM semantics to generate traces (Section 3.1). In step 2, traces generated by these formalized UCMs are analyzed through dependency analysis to support impact analysis at the modification request level (Section 3.2). In step 3, we combine UCM and FCA to support modification analysis through impact, feature interaction and regression test selection analysis at the requirements level (Section 3.3).

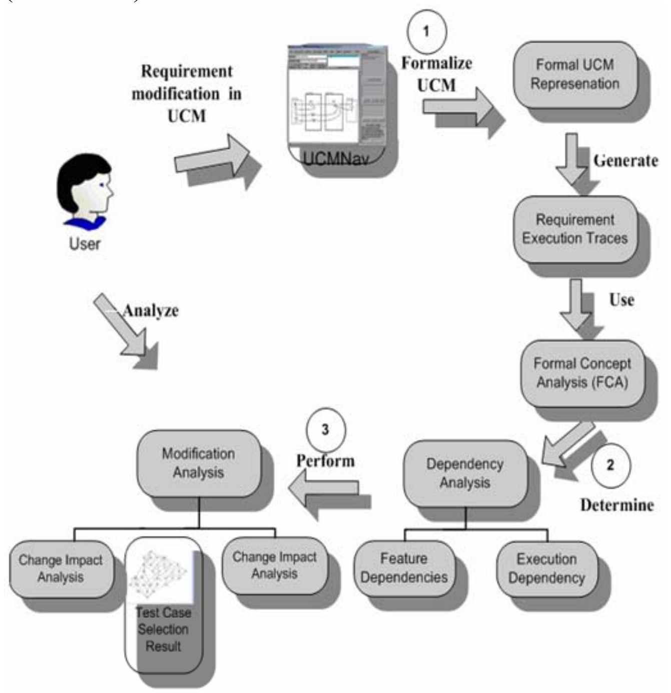

**Fig.3 Process of Implementing UCM_FCA Technique**

## 3.1. Generating UCM System Level Scenarios

In UCM, both its abstract syntax and static semantics are informally defined as XML document type definitions. Given the absence of a formal semantic, the interpretation of UCM specifications are also left completely to the user Furthermore, this informal representation results in a notation that does not support symbolic execution of these scenarios. In [8], we introduced an operational semantics for UCM based on Multi-Agent Abstract State Machines. The definition of the ASM formal semantics consists of associating each UCM construct with an ASM rule to model its behavior. The resulting ASM semantics are embedded in an ASM-UCM simulation engine designed for simulating and executing UCM specifications. It is written in AsmL [2], a high level executable specification language developed by the Foundations of Software Engineering (FSE) group at Microsoft Research. For a detailed description of the semantic we refer the reader to [8].

## 3.2. Dependency Analysis based on UCM Traces

Scenarios or traces (we use these terms interchangeable) are behavioral definitions of use cases, which typically correspond to user requirements (one or more features). In our approach, each of these scenarios represents a test case, which executes all UCM scenario elements in a particular scenario. Due to feature interactions, a requirement modification will often affect more than one scenario. Having the traces from the executable UCM allows us to apply dynamic dependency analysis to determine the impact of a modification on the overall system. In our research, we define and use feature and execution dependency analysis to answer questions about what and how objects are related with another.

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

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

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

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

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

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

- *Component execution dependency*

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

- *Domain element execution dependency*

Domain elements are execution dependent if their scenarios share common Use Case Map domain elements. We base our definition of domain elements on a subset of the elements introduced in [9]. We can now state that E is a set of UCM domain elements where E = {SP $\cup$ EP $\cup$ R $\cup$ AF $\cup$ AJ $\cup$ OF $\cup$ OJ $\cup$ ST}, where SP is the set of Start Points, EP is the set of End Points, R is the set of Responsibilities, AF is the set of AND-Fork, AJ is the set of AND-Join, OF is the set of OR-Fork, OJ is the set of OR-Join, and ST is the set of Stubs. Thus, we can specify now that two scenarios *sc1* and *sc2* are domain element execution dependent if both scenarios share any executed element E', where E' $\subset$E.

Execution dependency, therefore, can be seen as an extension of feature dependencies, allowing for different impact analysis granularity levels. The component execution dependency is of particular interest for distributed and/or larger systems in order to comprehend the components (subsystems) to be affected by a specific requirements modification.

The domain element execution dependency on the other hand, focuses on a more fine grained analysis of the UCM domain elements and how these might be potentially affected by a modification request. Given these dependencies one can now apply FCA to automatically identify these dependencies from collected UCM traces.

### 3.3 Combining UCM with FCA

The contextual representation created by FCA directly supports the dependency analysis introduced in the previous sections. The formalization of UCM allows us to execute and generate traces to be used in FCA. FCA depends on the quality and coverage achieved by the traces used for the analysis. For that reason, we assume UCM scenario coverage. That is, every scenario at the UCM level was executed at least once. Depending on the dependency type, one can create now the corresponding context by selecting the appropriate object and attribute pair. The resulting FCA context table can then be visualized as a context lattice through visualization tools like GraphViz [2].

Domain element dependencies can be used to determine potential change impacts at the scenario level. Combining FCA with UCMs allows for the logical clustering of scenario groups within an UCM to identify feature interaction among plug-ins.

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

For the analysis of feature interactions, we adopt a three step methodology that analyzes and classifies features based on their interactions. In the first step of our methodology, execution traces are generated and collected for all scenarios defined in the UCM.

In the second step, a FCA context table is generated, where global variables of features correspond to attributes and scenarios become the objects. In the resulting FCA context table, those scenarios with no or only one feature can be eliminated, since they are *FI never occurs* [15]. This elimination reduces the number of scenarios that have to be further analyzed for feature interaction. In the third step, the remaining scenarios are classified as either *FI occur* or *FI prone* [15], by passing down all attributes from the upper levels in the concept lattice that are associated with a particular scenario. Once all scenarios are classified as either FI never occurs, FI prone or FI occur the modification request analysis can be performed.

## 4. Case Study

In this section, we revisit the telephony case study (Figures 1 and 2) to illustrate the applicability of our approach. We have implemented the framework in a prototype system to guide managers during modification analysis at the requirements level by performing automatic impact analysis (feature and execution dependency level) and selective regression testing at the UCM level. Due to space limitations, we only illustrate the execution dependency and selective regression testing analysis and omit the feature interaction analysis.

The telephony system (Figure 2) contains four functional features: basic call, OCS, CND, and the combination of OCS_CND. For the telephony system seven system level scenario definitions can be identified (Table 1).

**Table1: System scenario definitions**

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

### 4.1. Change Impact Analysis

Two of the main challenges in analyzing functional and execution dependencies in UCM are the use of *dynamic stubs* and the need to identify inter-scenario dependencies that might exist in an UCM. The use of

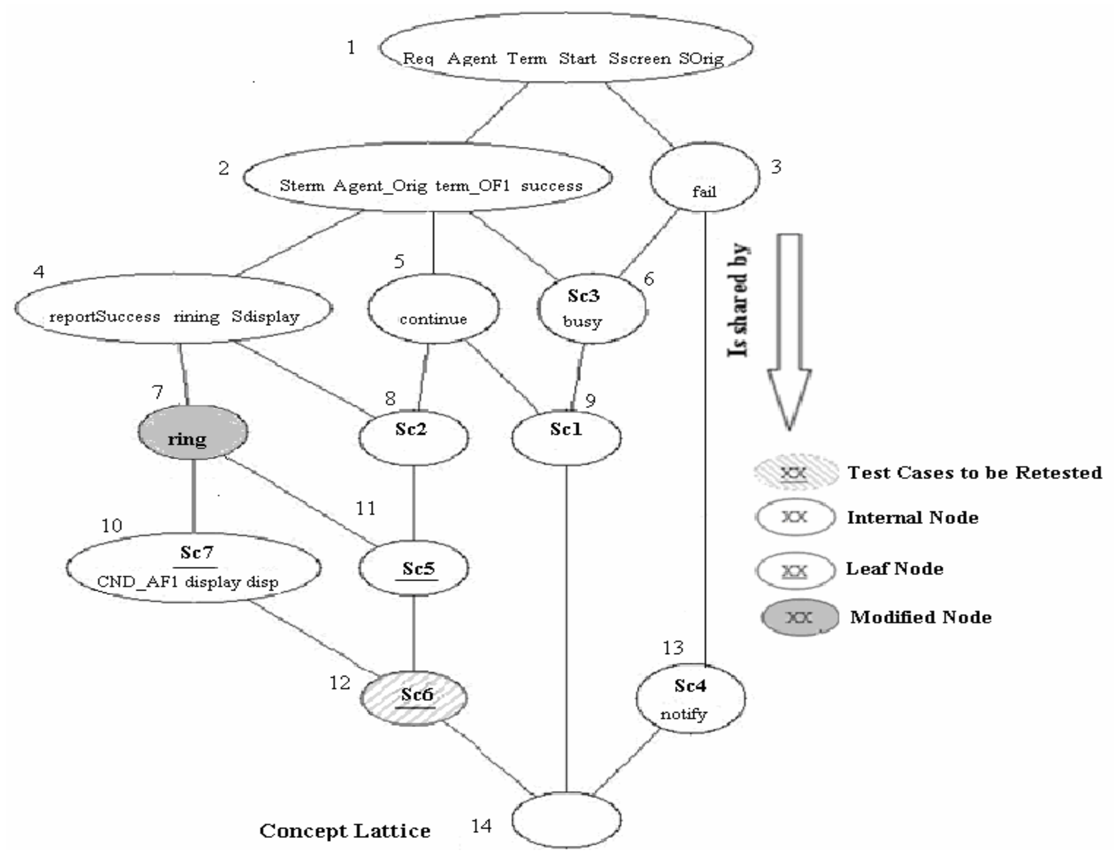

**Fig 4: Domain element execution dependency lattice**

FCA lattices allows for the generation of different views to guide maintainers during their analysis and to eases the interpretation of the analysis results. Table 2 provides an overview of some of these contextual views provided in our system.

**Table2: Overview of Execution dependency views**

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

One of the major advantages of FCA is its ability to visualize relationships between sets of objects and their common attributes (concepts) as a hierarchical graph (concept lattice). Objects and attributes in the concept lattice can be distinguished by

- The object (scenario) – is always on the upper line (i.e., Sc4 (concept #13 in Figure 4)).
- The attribute (or list of attributes) is found in the lower part of each concept node (i.e., notify (concept #13)).

In the execution dependency example (Figure 4), we select a concept, which contains only one attribute *ring* (concept#7) and pass all of its objects up to this node. In this case, all scenarios that are sharing this UCM element are included and one can easily identify that any change to this concept will potentially affect scenarios sc7, sc5, *sc6.*

## 4.2. Selective Regression Testing

In what follows, we present how our framework can assist in identifying regression tests (scenarios that have to be retested after a modification request is completed), based on a UCM domain element execution lattice.

The following assumes a modification request that involves concept #7 (containing the "*ring*" attribute in Figure 4). This modification will potentially affect scenarios *sc7*, *sc5* and *sc6* (mentioned in Section 4.1). Using our selective regression test selection approach, one can identify that only test case *sc6* is a leaf node that has to be re-executed since it automatically includes the execution of *sc5* and *sc7*. Our initial case study illustrates that our test case selection technique can successfully be applied at different analysis level to reduce the number of test cases at the requirements level. Similar to other dynamic analysis techniques, our execution dependency analysis depends directly on the quality of the traces used as input. In cases with low coverage, the concept lattice created by the FCA algorithm might be too small and imprecise to be useful for further analysis.

## 5. Related Work and Discussion

Most research on impact analysis and regression testing focuses mainly at source code level [14], [18]. Performing change impact analysis at source code requires a prior understanding of both the requirements and their mapping to the source code. Our approach supports modification analysis at the requirement level using a common representation - FCA lattices. Furthermore, we also provide for a unified approach to support different analysis techniques without requiring a previous understanding of the source code. Our impact analysis approach goes beyond our previous static approach [10] by automating the analysis part and using dynamic (trace information) to provide support for dynamic plug-ins. Additionally, we take advantage of FCA and its analysis and logical clustering flexibilities to support additional modification analysis (e.g. feature interaction). Furthermore, because of the use of FCA, the analysis is not limited to UCMs and its notation; in fact, any formalized and executable requirement notation can be integrated within our modification analysis framework. The most closely related work to ours is a greedy algorithm presented for FCA [20] with a focus on minimizing the number of test cases at the source code level. However, this approach is source code based and does not provide support for change impact and feature analysis.

It should be noted that our approach has limitations similar to other impact analysis and regression testing approaches and can only support modifications or deletion requests.

## 6. Conclusions and Future Work

Modification analysis is an early activity in most software maintenance process cycles and is often performed by decision makers not familiar with system implementation details. Our approach supports the modification analysis at the UCM requirements level by identifying potential change impacts and re-testing effort associated with a modification. We presented the following three major contributions: (1) of the ability to collect dynamic information from UCMs by formally mapping UCM constructs as part of an AsmL formalism; (2) applying FCA on these collected traces to identify feature and execution dependencies; (3) a tool implementation to support a fully automatic approach for both impact analysis and selective regression testing at the UCM requirements level.

A more detailed case study to validate the applicability of our approach is part of our ongoing work in this area.

**Acknowledgements**

We would like to acknowledge the contribution of Jacqueline Hewitt and Pabhanin Leelahapant.

## References

[1] D. Amyot, X. He, Y. He and D. Y. Cho, “Generating Scenarios from Use Case Map Specifications”, 3rd Int. Conf. on Quality Software (QSIC'03), Dallas, November 2003.

[2] AT&T Labs-Research, "Graphviz," http://www.graphviz.org, September 2006

[3] G. Birkhoff, "Lattice theory," Providence, Rhode Island: Amer.Math.Soc, 1967.

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

[5] L. Briand, Y. Labiche and G. Soccar, "Automating impact analysis and regression test selection based on UML designs," ICSM 2002., pp. 252-261, 2002.

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

[7] B. Ganter and R. Wille, *Formal Concept Analysis: Mathematical Foundations*. Springer-Verlag NY, 1997.

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

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

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

[11] International Standard - ISO/IEC 14764 IEEE Std 14764-2006 Software Engineering, Software Life Cycle Processes, Maintenance, ISBN: 0-7381-4961-6, 2006.

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

[13] Kimbler K., Velthuijsen H., “Feature Interaction Benchmark”, Discussion paper for the panel on Benchmarking at FIW’95 (Feature Interaction Workshop), 1995.

[14] J. Law and G. Rothermel, "Whole program path-based dynamic impact analysis," *Proceedings of the Interna-*

*tional Conference on Software Engineering,* pp. 308–318, 2003.

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

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

[17] C. Lindig, "Concept-based component retrieval," *IJCAI95 Formal Approaches to the Reuse of Plans,Proofs, and Programs,* 1995.

[18] G. Rothermel and M. Harrold, "Analyzing regression test selection techniques",IEEE TSE, Vol. 22, pp.529-551,1996.

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

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

[21] T. Tilley, R. Cole,P. Becker, and P. Eklund, *A Survey of Formal Concept Analysis Support for Software Engineering Activities*. International Conf. on Formal Concept Analysis, 2003

[22] UCM, « Use Case Maps Web Page and UCM Users Group»,http://jucmnav.softwareengineering.ca/twiki/bin/view/UCM/WebHome, 2006.

[23] G. Utas. A pattern language of feature interactions. In [77], pages 98–114, September 1998.