Modification Analysis Support at the Requirements Level

Maryam Shiri
Department of Computer Science and Software Engineering, Concordia University, Montreal, Canada
1-(514)- 848-2424 Ext. 7148
ma_shiri@cse.concordia.ca

Jameleddine Hassine
Department of Computer Science and Software Engineering, Concordia University, Montreal, Canada
1-(514)- 848-2424 Ext. 7148
j_hassin@cse.concordia.ca

Juergen Rilling
Department of Computer Science and Software Engineering, Concordia University, Montreal, Canada
1-(514)- 848-2424 Ext. 7148
j_rilling@cse.concordia.ca

ABSTRACT
Modification analysis is part of most maintenance processes and includes among other activities, early prediction of potential change impacts, feasibility studies, cost estimation, etc. Existing impact analysis and regression testing techniques being source code based require at least some understanding of the system implementation. In this research we present a novel approach that combines UCM with FCA to assist decision makers in supporting modification analysis at the requirements level. Our approach provides support for determining the potential modification and re-testing effort associated with a change without the need to analyze or comprehend source code. We demonstrate the applicability of our approach on a telephony system case study.

Categories and Subject Descriptors
D.2.1 [**Software Engineering**]: Requirement and Specification – l*anguages, methodologies*

General Terms
Management

Keywords
Change Impact Analysis, Regression Testing, Formal Concept Analysis, Use Case Maps

1. INTRODUCTION
While developing software systems, it is rare that the initial implementation of a system will correspond to the final version of a system at its retirement. Changing customer needs lead to ongoing adaptive requirement changes and new versions of the same system [13]. In fact, software implements solutions that are expected to change periodically, to allow systems to adapt to environment changes [3] [4]. The efficient management and execution of these changes is critical to software quality and managed evolution of such systems [4]. Existing maintenance processes (e.g., [10]) have been established to guide both managers and maintainers during typical maintenance and evolution tasks. Common to most of these process models is that they require some type of modification analysis prior the commitment to a change. Modification analysis is typically performed by managers or non-programmers and includes among other activities, feasibility studies, cost estimations, potential re-testing effort involved and resource planning. Common to these activities is the need to be able to identify the potential impact of a change request to empower the decision makers to predict and plan for the effort involved to complete a modification request [3][4]. Existing work in change impact analysis and regression testing focuses mainly on identifying changes at the source code level [12] [17]. These source code based approaches typically result in an accurate analysis of the change impacts, since the source code represents the final design implementation. However, they also tend to be time consuming and require an understanding of both the system requirements and their implementations. The need for the comprehension of source code makes these approaches less applicable for management and non-technical decision makers who are typically the ones performing the modification request analysis prior to signing up on a specific maintenance request.

In our research, we focus on supporting these management and decision makers during the early modification analysis. More specifically, we propose an approach for impact analysis and on an early detection of change impact and re-testing effort at the requirements level, without the need for programming or implementation knowledge. We combine an existing requirement modeling technique that is being translated into a formal semantics [8] to make the requirements model executable. From the executable Use Case Maps model, then traces can be collected that are further analyzed and abstracted using Formal Concept Analysis (FCA) [6]. Execution and functional dependencies are computed at different abstraction levels to determine the potential impact of a modification request on an existing system.

Our research is significant for several reasons: (1) We present a novel approach to determine impact analysis and selective regression testing at the requirements level without the need for the comprehension of the source code that implements the requirements. (2) Our approach provides different abstraction levels to perform change impact analysis and estimate the testing effort associated with a change request. (3) We combined our analysis techniques with visual abstractions to further provide support for management during their decision making process.

The remainder of the article is organized as follows. Section 2 introduces UCM, FCA and some background relevant to change impact analysis and regression testing. Section 3 introduces our

IWPSE'07, September 3-4, 2007, Dubrovnik, Croatia

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 our conclusions and future work.

2. RELATED BACKGROUND

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

2.1 Formal Concept Analysis

Formal Concept Analysis (FCA)[6], is a mathematical approach that dates back to Birkoff in 1940 [2]. FCA is commonly used for representing and analyzing information, by performing logical grouping of objects with common attributes. An FCA context is a triple C=(O,A,R, where O represents a set of objects and A, a set of attributes, with R $\subseteq$ O x A being a relation among them[14].

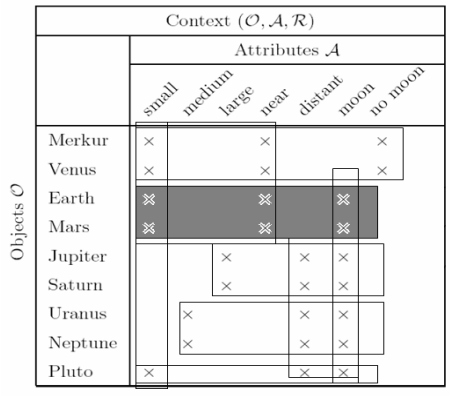

Figure 1. FCA context table example

A context is normally represented as a relation or a context table, where rows represent objects and columns their attributes. In the context table (Figure 1), cells marked with an "x" in each row indicate attributes associated with a particular object.

In a relation R, the concept (O, A) corresponds to the maximal set of objects (extent) that share a set of attributes (intent). For example, in Figure1 there exists a concept with Earth and Mars as its objects and small, near, and moon as its attributes (shown highlighted). The goal of FCA is to group concepts in such a way that no other object outside of an identified concept contains the same attributes and no other external attribute can be ascribed to other objects of that group. 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, or "concept lattice".

Formal Concept Analysis has gained popularity in recent years, due to: (1) Its programming language independence that allows users to easily define different views (using different object, attribute combinations); (2) Availability of tool support to automatically generate context tables and lattices; (3) Its relatively inexpensive analysis, compared to other more traditional dynamic dependency and trace analysis techniques. It has however be noted that FCA does not consider semantic information about the trace content, which limits its applicability for certain types of analysis. Furthermore, it is not always easy to determine a meaningful context for the analysis.

2.2 Use Case Maps

Use Case Maps (UCM) [11], are a modeling technique that has been applied to capture functional requirements in terms of causal scenarios. UCMs can represent behavioral aspects at a higher level of abstraction than for example UML diagrams. UCM can visualize an entire system to provide a better understanding of the evolving behavior of complex and dynamic systems at the requirements and specification level. The Use Case Maps notation is a high level scenario based modeling technique, to specify functional requirements and high-level designs for various reactive and distributed systems. UCMs can also provide stakeholders with guidance and reasoning about system-wide functionalities and behavior. Originally introduced to model the behavior of telecommunication systems, the application of UCMs has also been extended to other application domains, such as web services, airline reservations, object-oriented frameworks and many more.

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

Scenarios in UCM are expressed above the level of messages exchanged between components, hence, they are not necessarily bound to a specific underlying structure (such UCMs are called Unbound UCMs). Path details can be hidden in sub-diagrams called plug-ins, contained in stubs (containers) on a path. A stub can be either static (represented as plain diamond) which contains only one plug-in, or dynamic (represented as dashed diamonds), which may contain several plug-ins

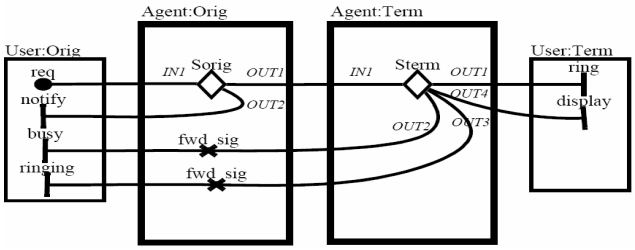

Figure 2. A simple telephony system (UCM root map)

Dynamic plug-ins are selected at run time according to a selection policy. One of the strengths of UCM is its ability to integrate a number of scenarios together (in a map-like diagram), and the ability to interpret the architecture and its behavior over a set of scenarios. UCM is not intended to replace UML, but rather complement it and bridge the gap between requirements (use cases) and design (system components and behavior). Use cases describe the system according to its external behavior (black-box view), whereas UML class diagrams are used to describe how the

system is constructed (glass-box), but do not describe the system behavior. UCMs are an attempt to close this conceptual gap that typically exists between requirement and design [1].

2.2.1 Basic UCM Notation

Each scenario execution path begins with a start point (filled circle) and contains responsibilities (represented as crosses) that are abstract activities used to represent functions, tasks, procedures, events. The execution of a path terminates at an end point (represented as vertical bars). UCMs also provide in the ability to structure and integrate scenario sequences, using alternatives (with OR-forks/joins) or concurrently (with AND-forks/joins) [9].

2.2.2 UCM Example - A Simple Telephony System

Figure 2 shows the UCM root map of a telephone example that was originally introduced in [15]. The example describes the connection request phase in an agent based telephony system with user-subscribed features. The case study contains four components (originating/terminating users and their agents) and two static stubs. Upon the request of an originating user (req), the originating agent 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 corresponding to user-subscribed features.

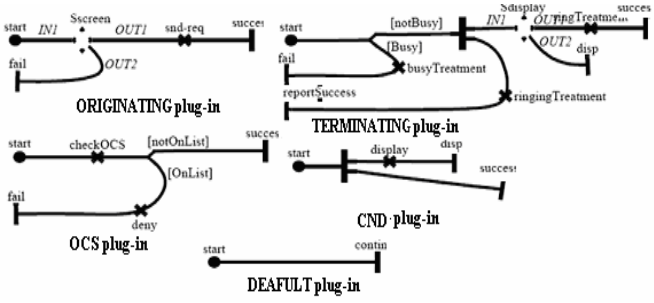

Figure 3. Plug-ins for Simple Telephony Features

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.

As part of this UCM a set of global variables are defined: Busy (the callee is busy), OnOCSList (the callee on OCS list), subCND (the callee is subscribed to CND), subOCS (the caller is subscribed to OCS). Figure 3 illustrates the corresponding UCM that was generated with UCM Navigator1. Each plug-in is bound to its parent stub, i.e. stub input/output segments (IN1, OUT1, etc.) are connected to the plug-ins' start/end points.A more detailed discussion and description can be found in [9].

2.3 Change Impact Analysis (CIA)

When a change occurs in a system, regardless whether it is at the requirement, design or code level, it will likely affect certain parts of a system. Impact analysis focuses on identifying these parts of a system that are (potentially) affected by modification request. Change impact analysis is also the basis for regression testing, allowing for the identification of these test cases that have to be re-tested after a modification is performed. Currently, most of the research on change impact analysis focuses on source code [12] and design level analysis [5] using either traceability or dependency analysis [3]. 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.

2.4 Regression Testing

After a modification is implemented, a system has to be validated to ensure that the modified parts have not introduced any new errors into previously tested code [7]. One existing approach to reusing test suites is called retest-all that requires re-running all the test cases in the existing test suite. However this approach is often too expensive and unnecessary. In most cases, except for the rare event of a major rewrite, changes are localized and do not affect the entire system. Selective regression testing allows for a reduction of the number of test case to be re-executed and therefore the cost associated with testing, by identifying the subset of test cases that are relevant and have to be re-run. Regression techniques always imply some type of impact analysis to determine the coverage needed by the selected regression tests. One can therefore informally describe regression testing 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', with T' $\supseteq$ T that provides sufficient confidence in the quality of P′ [17].

3. REQUIREMENS MODIFICATION ANALYSIS

One of the challenges for management is to determine early in the software maintenance cycle, the potential affect of a modification request on the overall system and to determine the testing effort related to a particular change.

The early detection of potential change impacts not only allows predicting the potential cost associated with a modification request but also its feasibility.

In what follows we introduce the major activities involved in our approach (Figure 4). In (1), we utilize a formalized UCM semantics to generate traces (Section 3.1). In (2) traces generated

[1] www.ucm.org

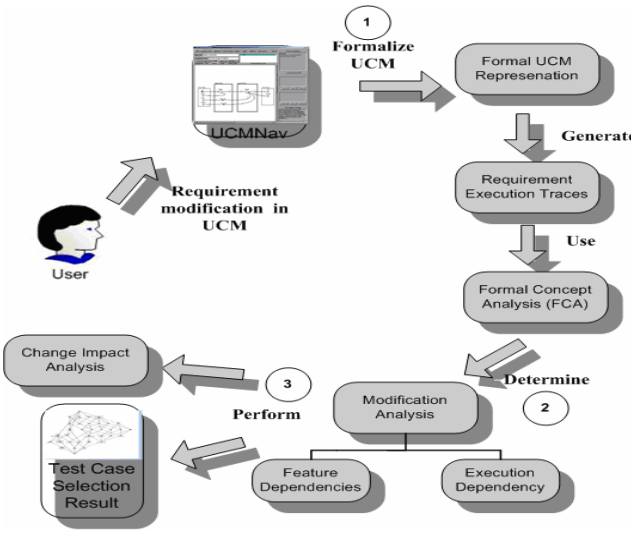

Figure 4. Process of Implementing UCM_FCA Technique

In what follows we introduce the major activities involved in our approach (Figure 4). In (1), we utilize a formalized UCM semantics to generate traces (Section 3.1). In (2) traces generated by these formalize UCMs are analyzed through dependency analysis to allow for an impact analysis of a modification request on the overall system (Section 3.2). In (3) we support the modification analysis at the requirements level, by combining UCM and FCA by identifying potential change impacts and regression test cases (section 3.3).

3.1 Formal Semantics for UCM

The lack of a formal representation in many requirements modeling languages is the main cause for their ambiguity and misinterpretation. The same holds for UCMs, where both the abstract syntax and static semantics are only informally defined as XML document type definition. Resulting in UCM diagrams that are non-executable and scenarios, global variables, and stubs have to be analyzed and traced manually Furthermore, with the absence of a formal semantic, the interpretation and verification of these diagrams are also left completely to the user. This existing ambiguity and lack of verifiability can be addressed by adding formal semantics to requirement specification languages. We introduced in our previous work [8] a formal semantics for UCM that addresses this issue. This approach can be seen as a complementary, unambiguous documentation approach that provides additional insights in the UCM language and its notation, as well as a basis for future formal verification of UCMs.

Our formal operational semantics for the UCM language is based on Multi-Agent Abstract State Machines (ASM) [8]. The definition of the ASM formal semantics consists of associating each UCM construct with an ASM rule to model its behavior. The resulting operational semantics are then embedded in an ASM-UCM simulation engine and are expressed in AsmL2, an advanced ASM-based executable specification language. The ASM-UCM simulation engine is designed for simulating and executing timed UCM specifications. It is written in AsmL, a high level executable specification language.

In order to apply ASM rules, the UCM specification (originally described in XML format) should be translated into a hyper graph format where constructs are connected using hyperedges. For this purpose, we define a UCM specification as a hyper graph: SPEC = (C, H, λ. where:

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

- H is the set of hyperedges

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

The input for a ASM-UCM is a UCM specification and selected global variables for which, UCM is then executed

3.2 Dependency analysis on UCM traces

Scenarios are behavioral definitions of use cases, which typically correspond to user requirements. From a maintainer's perspective it becomes important to identify which scenarios are affected by a modification request. Also, in our approach each of these scenarios (presented in Table2) represents a test case which executes all UCM scenario elements. It has to be noted that due to scenario 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 utilize functional and execution dependency analysis.

Functional dependency. Scenarios are directly related to use cases and can therefore be mapped to functional requirements, involving one or more system features. Therefore, we can informally define scenarios to be functional dependent when the following holds:

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

Scenarios are functional dependent if one of these conditions holds. We further refine this definition with the assumption that functional features are represented by UCM plug-ins/stub combinations. Therefore, we can now say that functional 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 Scenario Sc1 and Sc2 are functional dependent if they share the same plug-in (feature) Pa, where Pa $\supseteq$ 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 on importance due to the wide spread use of protocols and communicating entities in most systems. UCMs support the modeling of dynamic behavior through sub-maps, called plug-ins,

2 http://research.microsoft.com/foundations/asml

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 our formalized UCM approach is that these dynamic dependencies of the plugs are resolved 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 the UCM context one can apply scenario execution dependencies at two levels of abstraction: component and domain element.

- *Component execution dependency*
  From an UCM perspective a component dependency exist, when two or more scenarios share the same component C', where C' $\supseteq$ {set of UCM components}.

- *Domain element execution dependency*
  Domain elements are execution dependency 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 $\bigcup$ EP $\bigcup$ R $\bigcup$ AF $\bigcup$ AJ $\bigcup$ OF $\bigcup$ OJ $\bigcup$ 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' $\supseteq$ E.

  Execution dependencies can be seen as an extension of functional dependencies, allowing for different granularity levels for the impact analysis to be performed. The component execution dependency is in particular of interest for distributed and/or larger systems, where one wants to gain a general understanding of the components (subsystems) to be affected by a specific requirements modification. Furthermore, it can be utilized for the verification of the system behavior after performing a modification request equivalent.

  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

One of the major advantages of FCA is its flexibility in generating automatically different types of contexts, depending on the combination of objects, attributes selected by the user. The contextual representation created by FCA supports directly 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, attribute pair. The resulting FCA context table can then be visualized as a context lattice through visualization tools like GraphViz3, an open source graph visualization software. The visual representation eases the interpretation from the dependency analysis and allows the generation of different views to guide maintainers during their analysis. Table 1 provides an overview of some of these contextual views we provide.

Table 1. An overview of our system context view

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

For example, functional dependencies can be utilized to determine potential change impacts at the scenario level. Utilizing FCA with UCMs allows us to validate and restructure scenario group classifications within an UCM to identify feature interaction among plug-ins.

For selective regression test case analysis, the concept (requirement) to be modified is selected in the concept lattice. The regression test analysis will identify 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, contained in non-leaf nodes that are included in the path between the modified node and its reachable leaves are excluded, because they are already covered by the leaf notes.

4. CASE STUDY

In what follows we revisit the telephony case study (Figures 2 and 3), to illustrate the applicability of our approach. We have implemented the algorithms in a prototype system to guide managers during modification analysis at the requirements level, by performing automatic impact analysis and selective regression testing at the UCM level.

Table 2. System scenario definitions

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

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

The telephony system (Figure 2) contains 4 functional features: basic call, OCS, CND, and the combination of OCS_CND. A scenario definition consists of an identifier, a name, initial values for the global variables, a list of start points, and post-conditions (optional) based on the global variables. For the telephony system seven system level scenario definitions can be identified (Table 2).

4.1 Change Impact Analysis

One 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.

Functional Dependency

In UCM a plug-in represents a group of sub-scenarios containing a certain feature. An example for such a feature interaction is the OCS_plugin (depicted with dashed line in Table 3) that represents sub scenarios that utilize the Originating Call Screening (OCS) feature. Therefore, all scenarios containing the OCS plug-in share the OCS functionality and are functionally dependent on the OCS feature. As for identifying the potential change impact, one can determine from Figure 5 that Scenarios Sc3, Sc4. Sc5, and Sc7 share the OCS-plug-in. Therefore, any change in this feature or any of its elements will potentially affect these 4 scenarios and have to be retested.

Table 3. Feature dependency context table

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

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

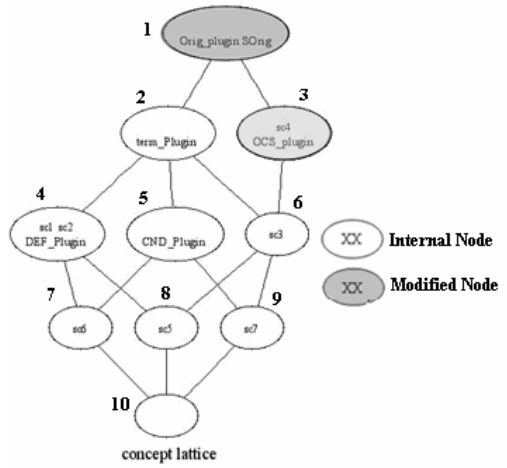

Figure 5. Feature dependency lattice

A concept lattice representation can reduce such information overload by providing a visual grouping of common attributes. Figure 5 shows a feature dependency concept lattice for context table in Table 3. Objects and attributes in the concept lattice can be distinguished by

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

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

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

Execution dependency

In the execution dependency example (Figure 6), we select a concept which contains only one attribute ring (concept#7). In this case, all scenarios that are sharing this UCM element are included, one can easily identify that any change to this concept will potentially affect sc7, sc5, sc6. In the next example we assume a modification request for scenario sc4 (concept #13). Based on the execution dependency analysis, sc4 has an execution dependency E for the following 7 attributes (all attributes passed down from the upper levels): E={Req, Start, SOrig ,Agent_term, Sscreen, fail, notify}.

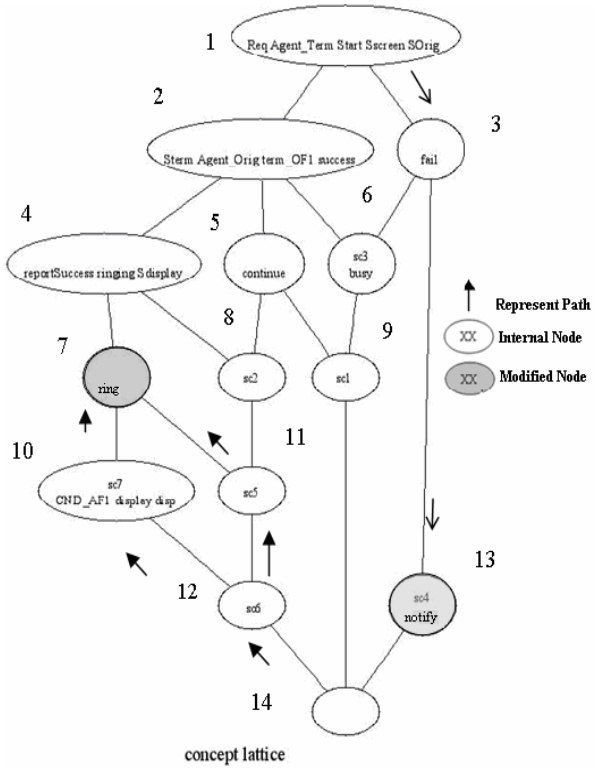

Figure 6. Domain element execution dependency lattice

These attributes are not only be potentially affected by a modification request for sc4, also any change to one of these domain elements can potentially affect scenario 4.

4.2 Selective Regression Testing

In this section we demonstrate how our approach can assist in identifying regression tests on a domain execution lattice (similar can be performed for the feature dependency lattice) to identify the scenarios that have to be re-executed after a modification request is completed. Assuming a given modification request involves concept #7 (containing only the "ring" attribute in Figure 7). This modification will affect *sc6*, *sc7*, and *sc5*. Applying our selective regression test selection approach, only test cases *sc6* has to be re-executed since it includes *sc5* and *sc7*.

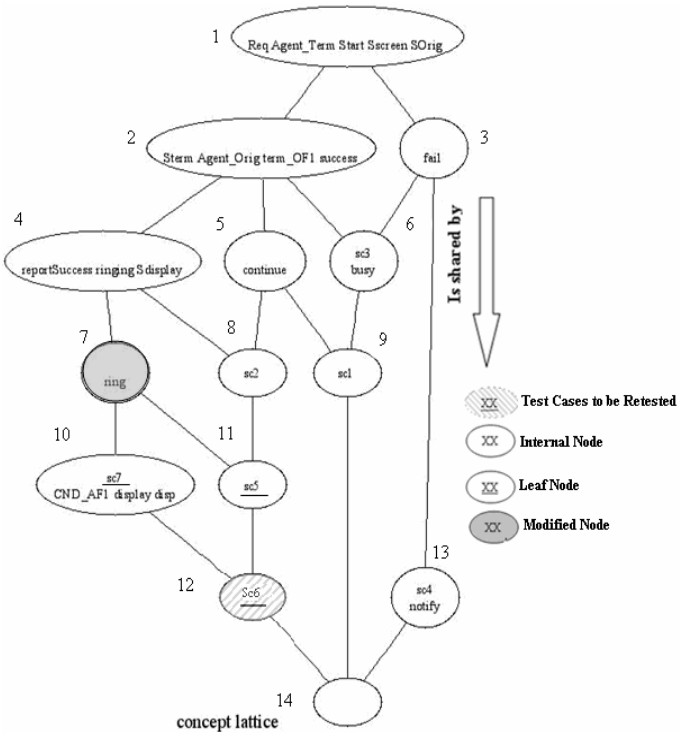

Figure 7. Domain element modification example

In a second example, we introduce a change in the most common domain elements (concept#1). Using our selective regression testing approach, one quickly can identify that only sc6 (concept #12), sc1 (concept #9) and sc4 (concept #13) have to be retested, all other scenarios and their executions are covered by these 3 scenarios.

Our initial case studies illustrate 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. It has to be noted, that similar to other dynamic analysis techniques, our execution dependency analysis depends directly on the quality of the traces used as input. More specifically, in our case the coverage will directly affect the accuracy of our algorithm. In cases with low coverage, the concept lattice created by the FCA algorithm will be small and imprecise.

5. RELATED WORK AND DISCUSSION

There exists a significant body of research on impact analysis with the majority this research focusing on identifying changes and their impact at the source code level [12]. Performing change impact analysis at the source code level however requires that a maintainer has an understanding of both the requirements and their mapping to the source code, to be able to identify and localize the potential change. Our approach differs from these source code based impact analysis approaches by identifying the potential impact of a change at the requirement level. It does not require programming expertise or knowledge about the source code. Our impact analysis approach also differs from the lightweight UCM approached we introduced in [9] that was based on a static analysis approach,. Due to its static nature, this approach resulted in conservative, imprecise handling of dynamic plug-ins. Furthermore, the dependency analysis in [9] was not fully automated and limited in the type of analysis supported. Our approach is automated and based on dependency analysis technique, taking advantage of FCA and its analysis and logical clustering flexibilities. Furthermore, because of FCA, the analysis is not limited to UCMs and its notation; in fact, any formalized notation can be integrated with our FCA tool

Similarly to research on impact analysis, existing work on regression test selection has mostly focused on source code level analysis [7],[17], and [18]. However, to the best of our knowledge, there exists no previous work on selective regression testing by means of FCA at the requirement level. In [5]a mapping between design changes in UML and code changes has been created to classify code tests. In [16] an approach for selective retest strategy is presented that relies on categorizing changes to UML design.

The most closely related work to ours is a greedy algorithm presented for FCA [18]. The algorithm uses attributes and object implications among the requirements and test cases with a focus on minimize the number of test cases at the source code level.

It has to be noted that our approach has limitations similar to other impact analysis and regression testing approaches. Like other techniques, our approach only supports modifications or deletions. For new requirements it is possible to use an iterative approach, by introducing the modification at the UCM level and comparing the system behavior after re-executing the UCM scenarios. Furthermore, the scalability of the lattice representation can become a challenge. This limitation is not specific to our approach it is common to the lattice representation used to represent FCA results. With a large number of scenarios and domain elements to be displayed, the concept lattice representation might suffer from information overload, making the lattice difficult to read and interpret. Filtering, context sensitive zoom and other techniques typical used in software visualization can be applied to improve further the visual scalability of the concept lattice and therefore its readability.

6. CONCLUSION AND FUTURE WORK

Modification analysis is an early activity in most software maintenance process cycles. Our approach supports the modification reanalysis at the UCM level, by identifying potential change impacts and testing efforts at the requirement specification level. Our approach introduces the following three major contributions: (1) Eliciting dynamic information from Use Case

Maps, by formally mapping UCM constructs as part of an AsmL formalism to allow for the collection of execution traces from UCMs. (2) Applying FCA on these collected dynamic traces to identify functional and execution dependencies. (3) A tool implementation that supports a fully automatic approach of both, impact analysis and selective regression testing at the UCM requirements level without the need for source code.

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

7. ACKNOWLEDGEMENTS

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

8. References

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

[2] Birkhoff, G. *Lattice Theory.* Providence, Rhode Island. Amer. Math. Soc. Col. Pub.,XXV, 3rd. ed., 1967.

[3] Bohner, S. A. and Arnold, R. S. An Introduction to Software Change Impact Analysis. *Software Change Impact Analysis*, IEEE Computer Society Press, pp. 1–26, 1996.

[4] Breitman, K.and Leite, J. Scenario Evolution: A Closer View on Relationships. *Proc.of the 4th Intl Conf.on Requirements Eng. (ICRE 2000)*, pp. 95-105

[5] Briand, L., Labiche, Y. and Soccar,G. Automating impact analysis and regression test selection based on UML designs. *Int. Software Maintenance Conf.*,pp. 252-261, 2002.

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

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

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

[9] Hassine, J., Rilling, J., Hewitt, J. and Dssouli, R. 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] International Standard - ISO/IEC 14764 IEEE Std 14764-*2006 Software Engineering, Software Life Cycle Processes, Maintenance*, ISBN: 0-7381-4961-6, 2006

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

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

[13] Lehman, M.M. and Belady, L. *Program Evolution – Processes of Software Change*, Academic Press, London, 1985.

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

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

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

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

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