Change Impact Analysis for Requirement Evolution using Use Case Maps

Jameleddine Hassine, Juergen Rilling, Jacqueline Hewitt
Department of Computer Science, Concordia University
Montreal, QC, Canada, H3G 1M8
{j_hassin, rilling, j_hewitt}@cs.concordia.ca

Rachida Dssouli
Concordia Institute for Information Systems Engineering, Concordia University
Montreal, QC, Canada, H3G 1M8
dssouli@ciise.concordia.ca

Abstract

Changing customer needs and computer technology are the driving factors influencing software evolution. There is a need to assess the impact of these changes on existing software systems. Requirement specification is gaining increasingly attention as a critical phase of software systems development process. In particular for larger systems, it quickly becomes difficult to comprehend what impact a requirement change might have on the overall system or parts of the system. Thus, the development of techniques and tools to support the evolution of requirement specifications becomes an important issue. In this paper we present a novel approach to change impact analysis at the requirement level. We apply both slicing and dependency analysis at the Use Case Map specification level to identify the potential impact of requirement changes on the overall system. We illustrate our approach and its applicability with a case study conducted on a simple telephony system.

1 Introduction

In developing software systems, it is rare that the initial design is the final design or final implementation. Evolution is critical in the life cycle of all software systems particularly those serving highly volatile business domains such as banking, e-commerce and telecommunications. In fact, software is used to implement solutions that are expected to change periodically to adapt to environment changes [5, 7]. 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 made. Changing customer needs leads to adaptive changes as part of software evolution. The efficient management and execution of these changes are critical to software quality and managed evolution of software systems [7].

Software change encompasses various activities that can be grouped under three main categories: comprehension of the software with respect to the change, actually implementing the change, and testing the newly modified system for side effects. Without complete comprehension of the change impact, it is likely that there will be unwanted side effects. Although change impacts are inevitable since different parts of a software system rarely work completely independently from each other, understanding the software helps to discover which parts of the system are potentially affected by a change and where these ripples may occur [5].

Change impact analysis can be defined as "identifying the potential consequences of a change, or estimating what needs to be modified to accomplish a change" [5]. Currently, the majority of change impact analysis methodologies focus on analyzing the impact of changes at the source code level [11, 17, 28]. These approaches results in an accurate analysis of a change and its impact on other system parts, since the source code contains the final implementation of the design. However, source code based analysis approaches are: (1) extremely time consuming requiring a good understanding of the source code and the places in the source code where the change will have to take place, (2) the amount of low level information provided can be overwhelming, especially if what is desired is an overall assessment of the affected components related to the potential impact of requirements changes.

Therefore, it is essential to apply change impact analysis at higher levels of abstraction to predict and assess the impact of a requirement or specification changes. In order to move the analysis to a higher level of abstraction, notations that are used to represent the requirement, specification or design of the software at that level must be used as the underlying model for the analysis.

Use Case Maps (UCMs) [14] have been introduced to capture and integrate functional requirements in terms of causal scenarios representing behavioral aspects at a higher level of abstraction, providing stakeholders with guidance and reasoning about the system-wide functionalities and behavior. Use Case Maps are part of a new proposal to ITU-T for a User Requirements Notation (URN) [14, 15]. UCMs have been applied in a number of areas: Design and validation of telecommunication and distributed systems [1], detection and avoidance of undesirable feature interactions [22] and evaluation of architectural alternatives [21]. UCM is not intended to replace UML, but rather complement it and bridge the gap between requirements (use cases) and design (system components and behaviour). UCM allow developers to model dynamic systems where scenarios and structures may change at run-time. In this paper we address how change impact analysis can be applied at the UCM requirement level, to support an early analysis and localization of the system parts that are affected by a requirement change.

The remainder of the paper is organized as follows. Section 2 introduces the UCM notation and an illustrative example, Section 3 introduces different requirement dependencies that can be identified at the UCM level. Section 4 introduces our UCM forward slicing algorithm, followed by a discussion on ripple effect analysis 5 and a case study to illustrate the applicability of our approach (section 6).

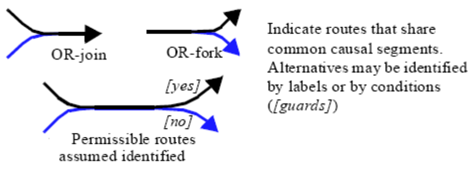
2 Use Case Maps (UCM)

The Use Case Maps notation is a high level scenario based modeling technique used to specify functional requirements and high-level designs for various reactive and distributed systems. UCMs expressed by a simple visual notation allow for an abstract description of scenarios in terms of causal relationships between responsibilities (e.g. operation, action, task, function, etc.) along paths allocated to a set of components. Components are generic and can represent software entities (objects, processes, databases, servers, etc.) as well as non-software entities (e.g. actors or hardware). These relationships are said to be causal because they involve concurrency, partial ordering of activities, and they link causes (e.g., preconditions and triggering events) to effects (e.g. post-conditions and resulting events).With the UCM notation, scenarios 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). One of the strengths of UCMs is their ability to integrate a number of scenarios together (in a map-like diagram), and to reason about the architecture and its behavior over a set of scenarios. UCM specifications may be refined into more detailed models such as MSCs [21].

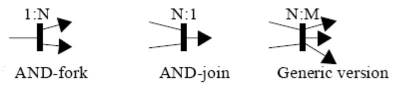
2.1 UCM Functional Notation

A basic UCM path contains at least the following constructs: start points, responsibilities and end points. **Start points.** The execution of a scenario path begins at a start point. A start point is represented as a filled circle representing preconditions and/or triggering events. **Responsibilities.** Responsibilities are abstract activities that can be refined in terms of functions, tasks, procedures, events. Responsibilities are represented as crosses. **End points.** The execution of a path terminates at an end point. End points are represented as bars indicating post conditions and/or resulting effects.

Scenarios integration. UCMs provide help in structuring and integrating scenarios sequencies, using alternatives (with OR-forks/joins as illustrated in Figure 1(a) or concurrently (with AND-forks/joins as illustrated in Figure 1(b).


OR-join
OR-fork
Indicate routes that share common causal segments. Alternatives may be identified by labels or by conditions (*[guards]*)
[yes]
[no]
Permissible routes assumed identified

(a) Shared routes and OR-Fork/Joins



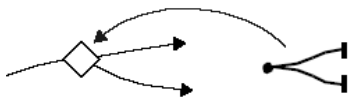
(b) Concurrent routes with AND-Fork/Joins

Figure 1. Structuring Scenarios

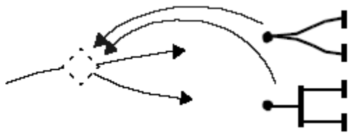
OR-Forks. Represent a path where scenarios split as two or more alternative paths. An OR-Fork has one incoming hyper-edge and two or more outgoing ones (a hyperedge is an edge that can have more than one source and target). Conditions (Boolean expression called guards) can be attached to alternative paths. **OR-Joins.** Capture the merge of two or more independent scenario paths. **AND-Forks.** Split a single control into two or more concurrent controls. **AND-Joins.** Capture the synchronization of two or more concurrent scenario paths.

When maps become too complex to be represented as one single UCM, a mechanism for defining and structuring sub-maps becomes necessary. Path details can be hidden in sub-diagrams called plug-ins, contained in stubs (diamonds) on a path. There are two kinds of stubs: (1) Static stubs represented as plain diamonds, contain only one plug-in, enabling hierarchical decomposition of complex maps.

(2) Dynamic stubs, represented as dashed diamonds, may contain several plug-ins, whose selection can be determined at run-time according to a selection policy (often described with preconditions). Figure 2 illustrates the stub concept.



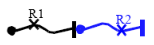
(a) Static stubs have only one plug-in



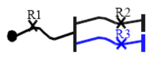
(b) Dynamic stubs may have multiple plug-ins

Figure 2. Stubs and plug-ins

Scenarios interactions. Different paths may interact with each other synchronously and asynchronously. Synchronous interactions are shown by having the end point of one path touching the start point (called a waiting place) of another path (Figure 3(a)). One path splitting into two concurrent segments represents an asynchronous interaction (Figure 3(b)).



(a) Synchronous Intersection



(b) Asynchronous interaction

Figure 3. Scenarios Interactions

2.2 UCM Example

The UCM model of figure 4 describes the connection request phase in an agent based telephony system with user-subscribed features. It 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 a number of results in the originating and terminating users.Stub *Sorig* contains the Originating plug-in whereas stub *Sterm* contains the Terminating plugin. These sub-UCMs have their own stubs, whose plug-ins are user-subscribed features.

In stub *Sscreen* we have:

-**OCS (Originating Call Screening)**: blocks calls to

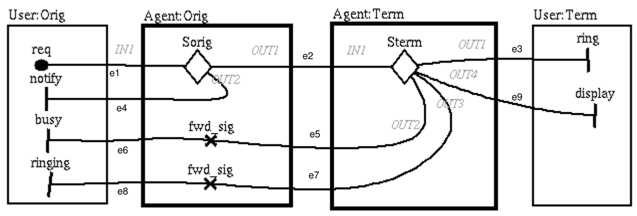


Figure 4. Simple telephone system

people on the OCS filtering list. It checks whether the call should be denied or allowed (chk). When denied, an appropriate event occurs at the originator side (notify).

-**Default**: used when not subscribed to any other originating feature.

And in stub *Sdisplay* we have:

-**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.

The set of global variables for the UCM map are: *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). These plug-ins, generated with the UCM Navigator, are presented in Figure 5. 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 can be found in [13]. In what follows we extend the interpretation of Use Case Maps provided in [13], creating the foundation for our UCM impact analysis approach.

Definition 1 (Use Case Maps) *We assume that a UCM Requirement specification* RS *is denoted by (D, C, H, λ, Bc) where:*

-D is the UCM domain, composed of sets of typed elements. D= SP $\cup$ EP $\cup$ R $\cup$ AF $\cup$ AJ $\cup$ OF $\cup$ OJ $\cup$ Tm $\cup$ 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, Tm is the set of Timers and ST is the set of Stubs.

-C is the set of components in RS (C=$\emptyset$ for unbound UCM)

-H is the set of of hyperedges connecting UCM constructs to each other

-λ is a transition relation defined as: λ=D$\times$H$\times$D

-Bc is a component binding relation defined as Bc =D$\times$C. Bc specifies which element of D is associated with which component of C.

UCM components are computation units in the system. A

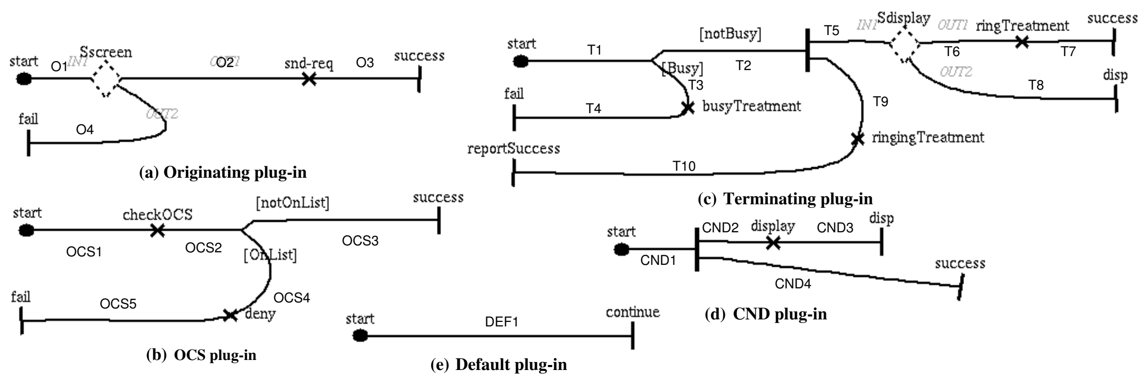


Figure 5. Plug-ins for Basic Call and Two features

causal flow between two constructs (especially responsibilities) in two different components implies the need for message exchange. To ensure the causality between the responsibilities and events performed by two different components, message-like interactions are essentially required. To understand how a component interacts with other components, we must examine each interface of the component of interest.

Definition 2 (Component Interface) *Let RS = (D, C, H, λ, Bc) be an UCM specification and c$\in$C is a component. A component Interface Ic is defined as a subset of H (Ic$\subset$ H) that defines the interaction between c and other components in RS.*

A hyper-edge, element of Ic, is usually shared between c and the component that c interacts with. For example, in figure 4, the interface of the component *Agent:term* is I(Agent:term) = {e2,e3,e5,e7,e9}

3 UCM Requirement Dependencies

Impact analysis techniques can be partitioned into two categories: traceability analysis and dependence analysis [5]. Dependence-analysis-based impact analysis found in [9, 12, 19, 24, 25] attempts to assess the resulting changes on semantic dependencies among program entities. This is done by identifying the syntactic dependencies that may signal the presence of such semantic dependencies [6]. The techniques used to identify these syntactic dependencies include static [30] and/or dynamic [16] slicing techniques. Other techniques usingtransitive closure on call graphs [5] attempt to approximate slicing-based techniques, while avoiding the cost associated with dependency analysis. Approximate dependence-based impact analysis techniques include expert judgment and code inspection. These approaches my often be incorrect [20], and performing impact analysis by inspecting source code can be expensive [23], due to a lack of automation.

In this research we focus on combining a UCM slicing algorithm (section 4) with the dependency analysis techniques introduced in this section to address some of the shortcoming of the existing approaches [5, 12, 19, 30]. The dependency analysis algorithm uses as input an UCM specification and the necessary slicing criterion (based on the change request) that will provide the set of impacted UCM elements. In what follows we provide a detailed discussion on UCM based dependency analysis at the scenario and component level.

3.1 UCM Scenario Dependencies

Scenarios in UCM inherently contain dependency information as part of their modelling. Scenario dependencies can be applied in assessing the ripple effects of a change at the scenario level of understanding. At the UCM level of abstraction, we distinguish three types of dependencies: functional, containment and temporal.

3.1.1 Functional dependency

As discussed earlier in section 2, UCMs integrate many individual scenarios. We can define system level scenarios as being end to end scenarios, where each scenario starts at a start point and ends at an end point. Scenario definitions are used to describe particular scenarios, representing them as partial orders of UCM elements (i.e. sequence and concurrency are preserved, but alternatives are resolved). System level scenarios make use of a path data model composed of global (Boolean) variables used on guarding conditions. A scenario definition contains an identifier, a name, initial values for the global variables, a list of start points, and (optionally) post-conditions expressed using the global variables.

From the telephone system introduced in section 2.2, one can identify seven system level scenario definitions that are summarized in figure 6. All scenarios start at the start point req. Variables Busy and OnOCSList are used to guard the two OR-forks found in the plug-ins (see Fig.5), whereas subOCS and subCND are used to define the selection policies found in the dynamic stubs (i.e. they indicate whether a user is currently subscribed to a feature, see the conditions in section 2.2). No postconditions are necessary here. These scenarios cover all the paths found in this UCM model and they are organized in functional groups. Functional dependencies capture the coexistence of two or more scenarios inside a same conceptual (or logical) cluster. For instance, we have grouped scenarios according the features they are describing.

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

Figure 6. System scenarios definitions

3.1.2 Containment dependency

A containment dependency exists between a scenario S2 and a scenario S1, if S2 is used in the description of S1. Stub plug-ins are contained in system level scenarios since they describe disjoint pieces of the system scenarios. For instance, CND plugin is part of the terminating plugin which is part of all system level scenarios except the *OCSdenied*. Default plugin is part of both "Originating plugin" and "terminating plugin". Figure 7 illustrates the containment dependency graph.

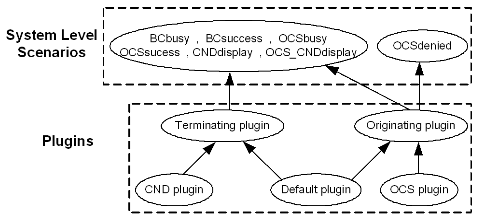


Figure 7. Containment dependency

3.1.3 Temporal dependency

Temporal dependency capture different types of temporal relationships that may exist between scenarios (e.g., one scenario excludes, waits for, aborts, rendezvous or joins another, concurrent, mutually exclusive, etc.). For the sake of generality temporal dependencies may be defined between system level scenarios, plugins or even sequential pieces of behavior. We denote the precedence relation by "$\ll$", the concurrency relation by "$|||$", the alternative relation by "[]", abort relation by "[$>$" etc. Figure 8 illustrates some examples of temporal dependencies between scenarios of the simple telephone system.

Precedence relation: <<
Originating plugin << Terminating plugin
OCSdenied < Terminating plugin
BCbusy < CND plugin
Concurrency relation: |||
CND plugin ||| (RingingTreatment ; reportSuccess)
(Display ; disp) ||| success
Alternative relation: []
OCSBusy [] OCSsuccess

Figure 8. Temporal dependencies

3.2 Component Dependencies

Components described in UCM, provide similar information to that of the scenarios but at a lower level of system abstraction. Component relationships depend on scenarios to provide the semantic information about their dependencies; meaning that components are dependent if they share the same scenario execution path. For a given component, we can identify forward and backward dependence. Components are forward dependent on a specific component (C) if in the containing scenario sequence their responsibilities are executed after that of the C. That is, there is component interface Ic emanating from C to the dependent component. Similarly, C is backward dependent on other components if there exists a component interfaces leading from them to C. This means that C waits on them before executing its responsibilities. We determine dependency information for a specific component by analyzing the scenarios that the component is contained within. For each scenario we identify components that execute before and after the specified component and create the dependency model.

In the following section, we introduce a UCM forward slicing algorithm that is an extension of our previous work [13] that facilitates impact analysis at the UCM level.

4 Use Case Maps Forward Slicing

In our previous work [13], we presented an algorithm for slicing UCM specifications based on a backward traversal algorithm. Intuitively, a UCM slice may be viewed as a subset of the behavior of a global UCM specification. While a traditional slice intends to isolate the behavior of a specified set of program variables, a UCM slice intends to isolate a set of scenarios with respect to a slicing criterion.

4.1 UCM Slicing Criteria

The selection of a slicing criterion depends on the particular analysis and maintenance task. The focus is frequently on the analysis of a requirement with respect to particular behavior or a particular component.

Definition 3 (UCM Slicing Criterion) *Let RS be a requirement Specification. A Slicing criterion (SC) for RS may be:*

-A UCM construct (responsibility, OR-fork. . . etc.)

Or

-UCM component

The task to be performed and the degree of existing system understanding will determine the slicing criteria. The following are some examples of potential slicing criteria supported by the algorithm:

-A user may want to assess the impact of modifying, adding or removing a specific functionality. In such a case, the user specifies a responsibility, a start point or an end point as the slicing criterion.

-A user may want to assess the impact of changing the semantic of a particular behavior (i.e, particular UCM construct) by changing a temporal relation. In such situation, the user specifies the construct subject to change as slicing criterion. For instance, substituting an AND-fork by an OR-Fork removes one concurrency relation and adds a new alternative relation. Therefore, the slicing criteria would be AND-fork.

-A user may want to focus the analysis on one specific component. The slicing criterion would be a UCM component.

Depending on the task at hand the maintainer specifies the slicing criterion. For instane, if the task at hand consists on deleting a scenario, the maintainer may choose the slicing criterion to be either the start point of the underlined scenario (if any) or the responsibility or the end point that triggers the execution of the scenario (i.e., where the preconditions of the scenario are satisfied). Similarly, adding a new scenario consists on localizing where the integration is supposed to take place and have this location (that corresponds to a startpoint, responsibility or an end point) as slicing criterion.

In order to define a UCM slice, we introduce the concept of: reduced domain, reduced stub, reduced component, and reduced transition relation.

Definition 4 (Reduced UCM elements) *Let RS = (D, C, H, λ, Bc) be an UCM Requirement Specification.*

-A reduced domain is a set D' that is derived from D by removing zero, or more elements (i.e. $D' \subseteq D$).

-Since a plug-in is also a stand alone UCM, a reduced plug-in can be defined in the same way as a reduced UCM(see definition 5).

-A reduced stub is a stub that contains reduced plug-ins and may have fewer plug-ins than the original stub.

-A reduced component c' is a component that has less functionalities than the original component.

-A reduced transition relation λ' is a relation derived from λ by removing zero or more tuples.

-A reduced component binding relation Bc' is a relation derived from Bc by removing zero or more couples.

Definition 5 (Reduced UCM) *Let RS = (D, C, H, λ, Bc) and RS' = (D', C', H', λ', Bc') be two UCMs. RS' is a reduced UCM of RS if:*

-D' is a reduced set of D

-C' = $c'1$, $c'2$,. . . , $c'n$ is a subset of C such that for k = 1, 2,. . . ,n. $c'k$ is a reduced component of ck

-λ' is a reduced transition relation of λ

-Bc' is a reduced component binding relation of Bc

Given an existing UCM, our goal is to compute a UCM forward slice which corresponds to a subset of the original UCM that preserves the semantics of the UCM with respect to chosen slicing criterion *SC*.

4.2 UCM forward slicing algorithm

In what follows, we present our two phase UCM slicing algorithm, which is based on a forward traversal of the UCM specification. The UCM slicer takes as input a complete UCM requirement specification (i.e., an XML file generated by UCMNav) and a slicing criterion (SC), and generates a reduced UCM specification and a sequence of impacted components. The first step consists on localizing SC. If SC is not found, a notification is sent to the user and the algorithm is stopped. Otherwise, two cases are considered:

-If SC is a component, the slicer looks into the Component binding relation (Bc) and the transition relation λ (both defined in definition 1) and compute its interface. The hyperedge that corresponds to the first transfer of control to SC (i.e., triggering the first action of SC), referred to as lower-I(SC), will be used as a starting point for the computation of the forwarded slice is step2.

-If SC is a UCM construct, then the search is reduced to a traversal of the transition relation λ for the root map and all its submaps. The transition that contains SC is then recorded and the component to which SC belongs is added to the sequence of Components: Seq-Comp.

```
Input: UCM + Slicing criterion SC
Output: Reduced UCM, Seq-Comp (Sequence of components)
Step1: /* Searching SC */
If (SC = component) Then
  If (SC found) then Compute I(SC); Go to Step2;
                else notify the user; Exit;
  endif
else /* SC not a component */
  If (SC found) then  record the tuple from λ that contains SC
                      add the enclosing component to Seq-Comp
                      Go to step 2
                Else  notify the user; exit
  Endif
endif
Step2: /* UCM forward traversal */
If (SC = component) then start := lower-lc(SC)
                         else start := SC
endif
Compute the transition closure for λ starting from "start"
Add new visited components to Seq-Comp
Add (GenericStartPoint, GenericHyperedge, SC) to the transition
relation of the resulting slice
```

Figure 9. UCM forward Algorithm

Once SC is localized, step 2 consists of a forward traversal of the specification starting from SC. A transition closure is applied to the transition relation λ staring from the tuple containing SC. We obtain as result a reduced transition relation containing only UCM transitions occurring after the execution of SC. At each step of the computation of the transition closure, we check if new components are visited. This is achieved by looking for the current construct in the component binding relation. New visited components are added to the sequence of components. In order to obtain a valid reduced UCM, we add a start point to the slice obtained from step 2. Figure 9 describes the high level schema of the UCM slicing algorithm.

5 Ripple Effect Analysis

Impact analysis techniques based on source code analysis have the clear advantage of being very accurate in the analysis as they identify impacts in the final product; however, they have the disadvantage of being very time consuming, limited in scope, and they require implementation of the change before the impact can be determined [8].

Change impact analysis also often referred to as ripple effect analysis is generally performed after the change has been implemented [5]. However, during change impact analysis, it is useful to see the potential effect that performing a change might have on the rest of the system. Ripple effect analysis is an iterative process which continues until no further ripples can be determined. We apply the UCM slicing algorithm to the UCM specifications to determine the ripple effect of a change.

For component ripple effect determination the output is the set of components that are related to the change component through its scenario paths. Specifically, for a given component we use its interface to determine the slicing criteria. The execution of the slicing algorithm adds to the impact set any new components that are encountered along the execution path. This impact set contains all the components that relate to the change component through any of the scenario paths that it is contained within.

Ripple effect may also be determined for any UCM construct. In what follows, we apply the forward slicing algorithm to determine the ripple effect of changing a UCM construct in the Simple Telephone System presented in Section 2.2.

6 Applying UCM Change Impact Analysis

In what follows we apply our UCM CIA prototype tool on the simple telephony system introduced in Section 2.2 to illustrate the applicability of the approach and a discussion on related work.

6.1 CIA tool

An architectural view of our CIA tool is shown in figure 10. UCMNAV is used to generate the XML file that is used as input for the CIA tool. An XML parser, integrated within the tool, transforms the XML file into a hyper graph format representation (see Definition 1). The slicing criterion is provided in the form of a UCM construct or a component to be changed. The ripple effect analyzer combines the output from the Slicer and the Scenario Dependency Manager to provide the user with the UCM components and constructs that are affected by the change.

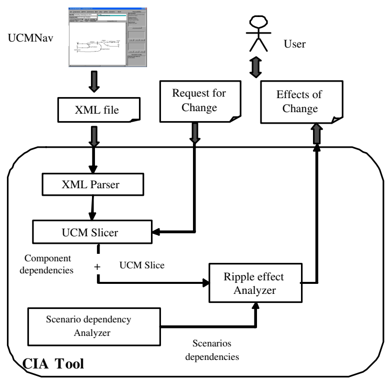
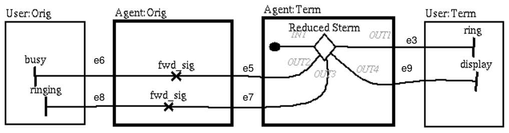


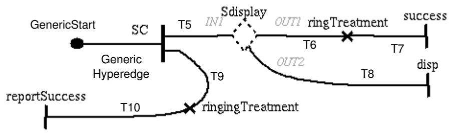
Figure 10. CIA Architecture

UCM Slicer. Figure 11 describes the resulting UCM slice obtained from the original UCM of the telephone system (figure 4 and figure5) that was computed with respect to the slicing criterion “AND-Fork” of the terminating plug-in. The Sequence of components produced by the slicer is SeqComp = [Agent:term, User:Term, Agent:Orig, User:Orig].

It can be observed that the resulting forward slice no longer includes stub Sorig within Agent:Orig, also the corresponding start and end points in the User:Orig component are excluded from the slice. As with traditional program slicing approaches [16, 30], the slice size is directly affected by several factors. As shown in [4] the slicing criteria, its position within a scenario/component and the overall cohesiveness of the system play an important role for the slice reduction. Furthermore, compared to the more traditional program slicing techniques, where the slicing criterion is restricted to a single variable, our UCM slicing approach supports multiple types of slicing criteria (see section 4) at different level of granularity, that are also influencing the slice size.



(a) Reduced Root Map



(b) Reduced terminating plug-in

Figure 11. Telephony system slice for SC

6.1.1 Dependency Analysis

Scenario dependency Manager. The scenario dependency analysis is performed statically on the complete UCM specification. A detailed discussion of the dependency analyzed by the scenario dependency manager can be found in section 3. Within the Scenario Dependency Manager, scenarios descriptions and their dependencies are stored and maintained. The resulting Up-to-date scenario dependencies are described as functional scenario table (figure 6), scenario containment graph (figure 7) and a table of temporal relations (figure 8). These dependencies are passed to the “Ripple Effect Analyzer” for analysis.

Ripple Effect Analyzer. It takes as input (1) A UCM slice and a component dependency list from the UCM slicer and (2) Scenario dependencies information from the scenario dependency manager. The ripple effect analyzer, identifies the scenarios that are contained in the slice, and then checks whether the requested change will affect the scenario dependencies. The affected scenarios and components are then communicated to the user.

Suppose that the system’s maintainer wants to assess the possible impact of changing the semantics of the Terminating plug-in in the example of Simple Telephone System, for instance changing the AND-Fork by an OR-Fork. The UCM Slicer computes the UCM forward slice (figure 11) along with the set *Seq-comp* and passes it to the “Ripple Effect Analyzer”. Substituting the AND-Fork by an OR-Fork will alter the stored temporal dependencies. For example, scenarios “CND plugin” and the scenario “RingingTreatment; reportSuccess” behave now as alternatives not concurrently as initially described in figure 8. Hence, both parts composing the temporal dependency (i.e., both scenarios) may be affected and then should be investigated with respect to functional and containment dependencies.

On the one hand, based on the identified containment graph, the CND plugin is enclosed in all system level scenarios (except OCSdenied) and in the terminating plugin as well. All these identified scenarios may be affected by the planned change and should be communicated to the user. On the other hand, the introduction of the new OR-Fork may also alter the table of functional dependencies. The Ripple Effect Analyzer has to check whether the existing system level scenarios still hold in the new context and communicates any potentially effected parts of the UCM to the user. In our example, the maintenance task has also an effect of increasing the number of functional dependencies within the system since by applying our scenarios definitions alternatives are resolved whereas concurrencies are preserved. Therefore new system level scenarios have to be defined. Once the user commits the changes these new scenarios are integrated in the Scenario Dependency Manager.

6.2 Discussion and Related Work

Change impact analysis that uses the design of the software as its basis for assessing relationships has several advantages and disadvantages on its own. These approaches assume that the model is consistent with the final implementation, although there are ways to verify this.

Our presented UCM impact analysis approach provides an important step in supporting software evolution at the specification level. As illustrated by our telephone example the approach changes are assessed at the early stages of the development process. The resulting output of the Ripple Effect Analyzer (scenarios and components subject to change) represents an excellent insight for the system maintainer to accomplish the task at hand. Only the selected scenarios are investigated in the detailed design and most likely in the implementation.

We believe that the chosen dependency classification is more suitable at the UCM level compared with the one presented in [2]. Indeed, the authors in [2] define among others: Input dependency relating scenarios based on their input, Output dependency relating scenarios that give the same output or leave the system in the same state after their execution, Input/Output dependency relating scenarios in which the output of one is the input to another. Input/output information is irrelevant at UCM stage and such information is left for later design stages where the scenarios are refined, for instance, in terms of message exchanges.

Related work

Lehman provides an in-depth analysis of different aspects of software evolution in [18]. He addresses the different types of systems and how they evolve; the evolution of the system in its context; the evolution of the development process. Requirements evolution is highly focused on tracing changed requirements to design, but there is little mention of how to assess the impact of changes at the requirement or design level. Requirements change analysis is discussed in [27] with a focus on assessing the information and techniques useful in assessing the risk of a changed requirement. Both sensitivity analysis and impact analysis are needed in a pro-active approach to change analysis. Settimi et al. present in [26] their work on software evolution, with a similar aim then ours - to provide a higher level of understanding of the change impact. However, they focus on Information Retrieval (IR) methods to facilitate traceability analysis to UML models. Similarly in [29] a fine-grained trace model for requirements impact analysis in embedded systems is presented.

Bai et al. [2] propose a scenario-based functional regression testing. In their approach have also integrated into their approach scenario based ripple effect analysis, traceability information, and slicing to determine affected components. Their focus however is on identifying components that have to be retested, limiting the analysis to a subset of the slicing criterion supported by our approach.. Furthermore their approach requires the availability of source and being able to create traceability links between scenarios and source code.

Ecklund et al [10] propose the notion of change cases, an adapted version of use cases, to be developed and maintained at the time of design to identify and incorporate expected future changes into the design to enhance the long-term robustness of the design. This idea provides an idealistic view, since it assumes that there are no time constraints on the development, and that it is possible to provide a conclusive prediction of future requirement changes. Furthermore it also requires that the change cases are maintained during the software and design evolution.

Briand et al [8] propose a change impact analysis method that is based on UML models and can be applied before implementing the changes. They have defined impact analysis rules to determine the directly and indirectly affected model components that depend on the type of change for which the impact analysis has to be performed. They also defined one rule for each change type. This approach focuses on defining rules using OCL that can be used on static UML models to formally determine the impact of a change. As well, very detailed UML models are used in this approach, requiring that detailed design descriptions are completed. Furthermore the approach focuses on the functional requirements rather on design changes.

7 Conclusions

Progress in analyzing and assessing the impact of requirement changes is crucial to the evolution of software systems. It is essential to extend and adapt functional and nonfunctional requirement changes, without destroying the integrity of the underlying system architecture and functionality. Furthermore, the analysis should be conducted as early as possible, before the change is implemented or a detailed understanding of the underlying source code is needed. In our approach, impact analysis is performed at the UCM specification level to allow for an early analysis of potential change impacts, at the requirement level, on the remainder of the system. We present a novel integrated approach that applies both scenario and component based dependency analysis techniques and our UCM forward slicing approach to identify change impacts at the requirement level. We provide an overview of our UCM-CIA tool and demonstrate the applicability of our approach on a case study based a simplified telephone system.

As part of our future work we will investigate the use of dynamic information to further reduce the size of the UCM slice. Furthermore we will be investigating predictive impact analysis measurements at the UCM level.

References

[1] Amyot D., Buhr R.J.A., Gray T. and Logrippo L., Use Case Maps for the Capture and Validation of Distributed Systems Requirements. RE99, Limerick, Ireland, June 1999,44-53.

[2] Bai X., Tsai W., PauL R., Feng K., and Yu L., Scenario-Based Modeling and Its Applications, Proceedings of the Seventh IEEE International Workshop on Object-Oriented Real-Time Dependable Systems, 2002, pp.253-260.

[3] Bai X., Tsai W., Paul R., and Yu L., Scenario-Based Functional Regression Testing, COMPSAC 2001, pp. 496-501.

[4] Binkley D., Harman M., A Large-Scale Empirical Study of Forward and Backward Static Slice Size and Context Sensitivity. ICSM 2003: 44-53

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

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

[7] Bohner S. A., Software Change Impacts: An Evolving Perspective, Proc. of IEEE Intl Conf. on Software Maintenance, 2002, pp. 263-271.

[8] Briand L.C., Labiche Y., and OSullivan L.O., Impact Analysis and change Management of UML Models, Proc. of the International Conference on Software Maintenance, 2003, pp. 256-265.

[9] Chang J. and Richardson D.G, Static and dynamic specification slicing. In Proceedings of the fourth Irvine Software Symposium, April 1994.

[10] Ecklund E. F., Delcambre Jr. L. M. L. and Freiling M. J., Change Cases: Use Cases that Identify Future Requirements, ACM SIGPLAN Notices, October 1996, pp. 342-358.

[11] Goradia T., Dynamic Impact Analysis: A Cost-effective Technique to Enforce Error-propagation, Proc. of the 1993 ACM SIGSOFT International Symposium on Software Testing and Analysis, 1993, pp. 171- 181.

[12] Gu J., Purdom P.W., Franco J. and Wah B.W., Algorithms for satisfiability (SAT) problem: A survey. DIMACS Volume Series on Discrete Mathematics and Theoretical Computer Science: The Satisfiability (SAT) Problem, American Mathematical Society, 1996.

[13] Hassine J., Dssouli R., Rilling J., Applying Reduction Techniques to Software Functional Requirement Specifications. SAM 2004: 138-153.

[14] ITU-T, URN Focus Group (2002), Draft Rec. Z.152 - UCM: Use Case Map Notation (UCM). Geneva. http://www.usecasemaps.org

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

[16] Korel, B. and Laski, J., Dynamic program slicing, In. Process. Letters, 29(3), pp.155-163, Oct. 1988.

[17] Law J. and Rothernel G., Whole Program Path-Based Dynamic Impact Analysis, Proceedings of the 25th International Conference on Software Engineering, 2003, pp. 308-318.

[18] Lehman M. M., Ramil J. F., Evolution in Software and Related Areas, Proceedings of the 4th International Workshop on Principles of Software Evolution, 2001, pp.1-16.

[19] Li L. and Offutt A. J., Algorithmic analysis of the impact of changes to object-oriented software. In Proceedings of the International Conference on Software Maintenance, , Monterey, CA, USA, Nov. 1996. IEEE, pp 171-184.

[20] Lindvall M.and Sandahl K., How well do experienced software developers predict software change? Journal of Systems and Software, 43(1):Oct. 1998, pp. 19-27.

[21] Miga A., Amyot D., Bordeleau F., Cameron C. and Woodside M., Deriving Message Sequence Charts from Use Case Maps Scenario Specifications. SDL'01, Copenhagen, 2001. LNCS 2078, 268-287.

[22] Nakamura N., Kikuno T., Hassine J., and Logrippo L., Feature Interaction Filtering with Use Case Maps at Requirements Stage. (FIW00), Scotland, May 2000.

[23] Pfleeger S. L., Software Engineering: Theory and Practice. Prentice Hall, Englewood Cliffs, NJ, 1998.

[24] Podgurski A. and Clarke L., A formal model of program dependences and its implications for software testing, debugging, and maintenance. IEEE Trans. Softw. Eng., 16(9): Sept. 1990, pp. 965-979.

[25] Ryder B. G. and Tip F., Change impact analysis for objectoriented programs. In ACMSIGPLAN- SIGSOFT workshop on Program analysis for software tools and engineering,. ACM Press, 2001. pp. 46-53.

[26] Settimi R., Clelena-Huang J., and Ben Khadra O., Supporting Software Evoution throught Dynamically Retrieving Traces to UML Artifacts, IWPSE 2004, pp. 49- 54.

[27] Strens M. R.and Sugden R. C., Change Analysis: A Step towards Meeting the Challenge of Changing Requirements, IEEE Symposium and Workshop on Engineering of Computer Based Systems (ECBS'96), IEEE, 1996. pp.278-283.

[28] Tonella P., Using a Concept Lattice of Decomposition Slices for Program Understanding and Impact Analysis, IEEE Transactions on Software Engineering, 2003, pp. 495-509.

[29] Von Knethen A., A Trace Model for System Requirements Changes on Embedded Systems, IWPSE 2001, pp. 17-26.

[30] Weiser M., Program slicing. IEEE Transactions on software Engineering, SE- 10(4):352-357, July 1984.