Towards Mutation Analysis for Use Cases

Huihui Zhang
Beihang University
100191 Beijing, China
zhhui@buaa.edu.cn

Tao Yue
Simula Research Laboratory
University of Oslo
1325 Lysaker, Norway
tao@simula.no

Shaukat Ali
Simula Research Laboratory
1325 Lysaker, Norway
shaukat@simula.no

Chao Liu
Beihang University
100191 Beijing, China
chaoliu@buaa.edu.cn

ABSTRACT

Requirements inspection is a well-known method for detecting defects. Various defect detection techniques for requirements inspection have been widely applied in practice such as checklist and defect-based techniques. Use case modeling is a widely accepted requirements specification method in practice; therefore, inspecting defects in use case models in a cost-effective manner is an important challenge. However, there does not exist a systematic mutation analysis approach for evaluating inspection techniques for use case models. As the first step towards a fully-fledged mutation analysis for use case models, in this paper we present the methodology we followed to systematically derive mutation operators for use case models. More specifically, we first propose a defect taxonomy defining 94 defect types, based on the IEEE Std. 830-1998 standard. Second, we systematically applied the basic guide words of the standardized Hazard and Operability Study (HAZOP) methodology to define 189 mutation operators, together with the defect taxonomy. Last, we define a set of guidelines for devising defect seeding strategies. The proposed methodology was evaluated by a real world case study and six case studies taken from the literature. Results show that all the RUCM mutation operators are feasible to apply and the defect taxonomy is the most comprehensive one to compare with the literature.

CCS Concepts

• Software and its engineering → Software creation and management → Requirements analysis.

Keywords

Requirements Inspection; Mutation Analysis; Mutation Operator; Hazard and Operability Study (HAZOP).

1. INTRODUCTION

Requirements Engineering (RE) is the first and the most critical stage of a system development lifecycle as requirements guide downstream activities of a system development, from analysis, design, implementation all the way to testing and deployment. Use case modeling has been widely accepted in practice as a way of specifying requirements [1]. Therefore, ensuring the quality of use case models in terms of well-known quality metrics (e.g., completeness, correctness, and ambiguity [2]) is therefore a practical challenge to be addressed.

Requirements inspection, a particular application of software inspection [3] for requirements documents, has been empirically assessed and widely accepted as an effective technique of early detection and elimination of defects in requirements [4]. Software inspection was first introduced in [5]. Since then, various requirements inspection methods have been proposed. Checklist-based reading (CBR) techniques (e.g., [4]) relies on a list of questions guiding a requirements inspector throughout a defined inspection process. Defect-based reading (DBR) (e.g., [6]) approaches are capable of providing specific instructions to inspectors on how to detect defects as compared with traditional checklist techniques. Perspective-based reading methods (PBR) (e.g., [7]) perform inspection procedures from perspectives of different stakeholders such as developers and testers. Usage-based reading (e.g., [8]) is based on prioritized use cases and aims to identify defects from the perspective of end users. Controversial evidence regarding the effectiveness of these requirements inspection methods have been reported in the literature (e.g., [9]).

Being aware of such inconsistent evidence-based conclusions for evaluating different requirements inspection methods, we do believe that there is a need to introduce mutation analysis from the testing field [10] to RE for facilitating the objective evaluation of the cost-effectiveness of various requirements inspection methods. Though in the literature, experiments (e.g., [9]) have been reported for evaluating requirements inspection methods by seeding defects, it does not exist any systematic mutation analysis in the RE literature. Mutation analysis (also called mutation testing) is a technique for evaluating the effectiveness of test suites [10]. The key element of mutation analysis is mutation operators [11].

Instead of being generic, we, in this paper, concentrate on one particular use case modeling methodology, named as Restricted Use Case Modeling (RUCM) [12]. RUCM was initially proposed to reduce inherent ambiguity of natural language based specification methodologies by relying on a structured use case template and a set of restrictions on the use of natural language and keywords. RUCM and its extensions have been used to address various industrial challenges such as specifying/modeling requirements, facilitating the generation of UML analysis models, specifying test case specifications and generating executable test cases [13, 14]. RUCM has been also evaluated (via controlled experiments) to be easy to use [12].

In this paper, we present our initial work towards a full-fledged mutation analysis mechanism for evaluating RUCM-based requirements inspection techniques. To the best of our knowledge, this is the first initiation towards introducing mutation analysis to use case models. More specifically, we first propose a comprehensive defect taxonomy by following the IEEE Recommended Practice for Software Requirements Specifications (i.e., IEEE Std. 830-1998) [2]. The defect taxonomy defines in total 94 types of defects, which are classified into nine categories (e.g., *Incompleteness*) and applied on all the RUCM elements (e.g., flows of events). Such a defect taxonomy is necessary because well-defined defects are the prerequisite for developing and evaluating an inspection approach for use case models [15, 16]. Moreover, it is a foundation to systematically propose mutation operators. Second, based on the defect taxonomy, by following the well-known and standardized guide words of the Hazard and Operability Study (HAZOP) [17], we systematically define 189 RUCM mutation operators, which cover all the RUCM model elements and all the defects defined in the RUCM defect taxonomy. Last, we recommend a set of guidelines for researchers and practitioners to devise defects seeding strategies.

The proposed methodology was evaluated by one real world case study and six other case studies from the literature. In total 5588 mutants were derived by using all the mutation operators at least once and covering all the proposed 94 defect types at least once, for all the seven case studies. Results show that all the RUCM mutation operators are feasible to apply. In addition, the 5588 mutants were used to evaluate three coverage criteria for automatically generating use case scenarios from use case models. Results show that the derived mutants with the RUCM mutation operators are effective in terms of differentiating the three coverage criteria.

The rest of this paper is organized as follows. Section 2 introduces the background, followed by the related work in Section 3. RUCM mutation analysis is presented in Section 4. In Section 5, we report the evaluation. We conclude the paper in Section 6.

2. BACKGROUND

In [12], RUCM was proposed to model use cases. RUCM has three key constructs: UML use case diagrams, RUCM use case template, 26 restriction rules for writing textual use case specifications. The RUCM template is composed of a set of fields, including use case name, brief description, pre-condition, dependencies with other use cases, actors, basic flow and alternative flows. An alternative flow always depends on a condition occurring in a specific step in a reference flow, which is either the basic flow or an alternative flow. Alternative flows are classified into three types: A specific alternative flow refers to a specific step in the reference flow; A bounded alternative flow refers to more than one step (consecutive or not) in a reference flow; A global alternative flow refers to any step in a reference flow. Figure 1 presents the use case specification *Withdraw Fund* borrowed from [18] and re-specified using the RUCM methodology and editor.

The 26 restriction rules of RUCM has two categories: 16 restriction rules on the use of natural language and 10 restriction rules in terms of enforcing the use of keywords for specifying control structures. These restriction rules reduce opportunities for ambiguities in use case specifications and help to facilitate the automated generation of other artifacts such as UML models [19] and test cases [20, 21]. Especially, RUCM defines a set of keywords to specify conditional logic sentences (IF-THEN-ELSE-ELSEIF-ENDIF), concurrency sentences (MEANWHILE), condition checking sentences (VALIDATES THAT), and iteration sentences (DO-UNTIL). As shown in Figure 1, keyword VALIDATES THAT is used in steps 5-8, which correspond to alternative flows *alt 1* and *alt 2*. The connections between the basic flow and the alternative flows are through keyword RFS. For example, RFS 8 in alternative flow *alt 1* refers to step 8 of the basic flow. Notice that the keywords are highlighted with different colors in the RUCM editor.

The Use Case Metamodel (UCMeta) was proposed to formalize RUCM models for supporting automated analysis and generation of other artifacts such as UML analysis models [12].

3. RELATED WORK

3.1 Mutation Analysis/Mutation Testing

Mutation analysis, also named as mutation testing and program mutation was proposed to assess the effectiveness of test cases in terms of the capability of discovering defects in software programs [22]. Mutation analysis has been widely applied to source code written in various programming languages (named as Program Mutation [10]) such as C [23], Java [24], Ada [25]. Mutation analysis can also be applied on program specifications, which are often named as *Specification Mutation* ([10]). For example, the authors of [26] applied mutation analysis to validate finite state machines. The recent survey was conducted by Jia and Harman [10], showing evidence that researchers focused more on Program Mutation than Specification Mutation. We are not aware of any work published in *Specification Mutation* for use case models.

The traditional process of mutation analysis for testing software programs is executed as below [10]: given a program p, a set of faulty programs p' (called mutants) is generated based on predefined mutation operators. The second step is to execute a test suite T with generated mutants p'. Note that the execution of T to the original program p should be successful and correct before the conduction of any mutation analysis. If results of running p' are different from results of running p, corresponding mutants are called 'killed'. The mutation score ("a ratio of the number of killed mutants over the total number of nonequivalent mutants [10] ") is then calculated and used to assess the effectiveness of test suite T.

Use Case Name	Withdraw Fund
Brief Description	ATM customer withdraws a specific amount of funds from a valid bank account.
Precondition	The system is idle. The system is displaying a Welcome message.
Primary Actor	ATM Customer
Secondary Actors	Card Reader
Dependency	INCLUDE USE CASE Validate PIN
Generalization	None

Basic Flow (Untitled)	Steps	
	1	INCLUDE USE CASE Validate PIN.
	2	ATM customer selects Withdrawal through the system.
	3	ATM customer enters the withdrawal amount through the system.
	4	ATM customer selects the account number through the system.
	5	The system VALIDATES THAT the account number is valid.
	6	The system VALIDATES THAT ATM customer has enough funds in the account.
	7	The system VALIDATES THAT the withdrawal amount does not exceed the daily limit of the account.
	8	The system VALIDATES THAT the ATM has enough funds.
	9	The system dispenses the cash amount.
	10	The system prints a receipt.
	11	The system ejects the ATM card.
	12	The system displays Welcome message.
	Postcondition	ATM customer funds have been withdrawn.

Specific Alternative Flow "alt1"	RFS 8	
	1	The system displays an apology message MEANWHILE the system ejects the ATM card.
	2	The system shuts down.
	3	ABORT.
	Postcondition	ATM customer funds have not been withdrawn. The system is shut down.

Bounded Alternative Flow "alt2"	RFS 5-7	
	1	The system displays an apology message MEANWHILE the system ejects the ATM card.
	2	ABORT.
	Postcondition	ATM customer funds have not been withdrawn. The system is idle. The system is displaying a Welcome message.

Figure 1. The Use Case *Withdraw Fund* specified in RUCM

Inspired by the traditional mutation analysis for testing, we apply mutation analysis to RUCM models, aiming to evaluate RUCM-based requirements inspection methods. A mutant in our context means a defected use case model. RUCM-based requirements inspection methods play the role as test suite in mutation testing.

3.2 Defect Taxonomies of Use Case Models

Anda and Sjøberg [16] proposed a checklist-based reading technique to detect defects in use case models. They presented a taxonomy of defects for use case models with a defect classification scheme: *Omission*, *Incorrect Fact*, *Inconsistency*, *Ambiguity*, and *Extraneous Information*. The comparison of our their defect taxonomy with the RUCM defect taxonomy is discussed in Section 5.3.1. A defect taxonomy (used as a checklist) focusing on semantic defects of use case specifications was proposed in [15]. Later, Phalp, et al. [27] refined this taxonomy by analyzing theories of text comprehension. Note that, as reported in [15], this taxonomy was designed for inspecting one use case specification at a time and use case diagrams are not covered as part of the inspection.

In [28], the authors presented a defect classification based on IEEE Std. 830-1998 [2] for requirements specification, which lists a set of quality metrics for specifications: *consistency*, *completeness*, *correctness*, *unambiguity*, *verifiability*, *changeability*, *traceability* and *prioritization*. To cover all concerns from different stakeholders, they extended the standard scheme with comprehensibility (easy to read), feasibility (from the perspective of system designers) and adequate level of detail (avoiding over-/less-information). Note that this classification is generic and most of the defects in the classification are applicable to other requirements specification methods.

In summary, Denger, et al. [28] defined the defect taxonomy from a high level of abstraction by following IEEE Std. 830-1998 [2]. Therefore it is considered as a general scheme for defining the RUCM defect taxonomy (Section 4.2). The defect taxonomy proposed by Anda and Sjøberg [16] can be considered as an application of the general taxonomy proposed by Denger, et al. [28], which is specific to use case models. Cox, et al. [15] and Phalp, et al. [27] concentrated on semantic defects of use case specifications. Both Denger, et al. [28] and Anda and Sjøberg [16] proposed the classifications based on the impact of different errors on the quality of use case models, while Cox, et al. [15] and Phalp, et al. [27] defined a set of quality indicators for evaluating use case specifications from the perspective of error sources.

The RUCM defect taxonomy we propose in this paper is more comprehensive and systematic as it is defined by following IEEE Std. 830-1998 [2] and it covers all possible defects that could occur on all RUCM model elements.

4. RUCM MUTATION ANALYSIS

The overview of the RUCM mutation analysis methodology is presented as a conceptual model in Figure 2. The methodology is defined to evaluate RUCM-based inspection methods. As for mutation testing, the effectiveness of such an inspection method is measured and evaluated with mutation scores. Taking a RUCM model as the input, a RUCM mutation operator mutates the RUCM model and generates a corresponding RUCM mutant, which contains a defect. The defect seeding strategy is used to guide a mutation seeding process to generate mutants.

In the RUCM mutation analysis methodology, defects are classified by the RUCM defect taxonomy, which is systematically derived by following the IEEE Std. 830-1998 standard [2]. Each RUCM mutation operator is derived by following HAZOP [29], which is a systematic process for analyzing any deviation of a system design, e.g., reflected in requirements and design models from the design intent. More specifically, based on the Extended Backus-Naur Form [30] of UCMeta, we systematically map each RUCM element to each basic HAZOP guide word defined in IEC 61882: 2001 [17]. We therefore define RUCM mutation operators based on the RUCM defect taxonomy. Each RUCM mutation operator can be derived if and only if the consequence of the corresponding deviation of the RUCM model can be interpreted as a particular defect defined in the RUCM defect taxonomy.

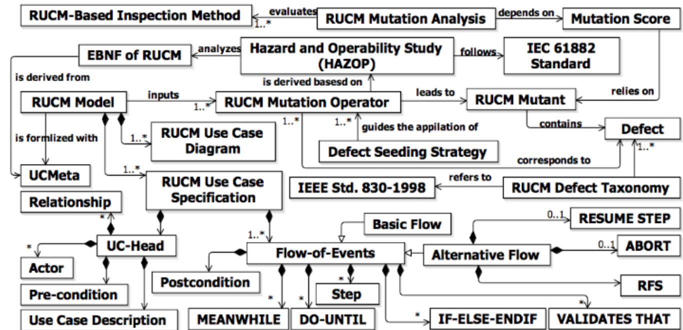

Figure 2. Overview of RUCM Mutation Analysis

In the rest of the section, we introduce the EBNF presentation (Section 4.1) of a subset of UCMeta, followed by the RUCM defect taxonomy (Section 4.2). We provide the RUCM mutation operators in Section 4.3 and the defect seeding strategy in Section 4.4.

4.1 Syntax of RUCM Models in EBNF

Mutation analysis relies on syntactic modifications of original programs [31]. Similarly, we formally define the syntax of RUCM use case models with EBNF [30] (Figure 3). Note that we apply EBNF in a more relaxed manner to represent the syntax of RUCM models. For example, the terminal terms are either natural language expressions such as 'UC-Name' or UCMeta model elements (e.g., RUCM Sentence). The EBNF representation (Figure 3) conforms to UCMeta, which is a metamodel for formalizing RUCM models for the purpose of automatically generating UML analysis models and tests (Section 2). However, it largely simplified UCMeta for the purpose of describing only RUCM model elements, on which the RUCM mutation operators can be applied. In the rest of the paper, one can refer to the EBNF representation for any definition of RUCM model elements.

4.2 RUCM Defect Taxonomy

4.2.1 Definitions

We follow IEEE Std. 830-1998 [2] to derive the RUCM defect taxonomy. The standard provides a classification of defects that might appear in software requirements specifications (SRSs). In the rest of this section, we introduce each of the categories defined in the classification. In addition, we introduce two other categories of defects borrowed from [16, 28].

Incorrectness (C1). In IEEE Std. 830-1998 [2], a SRS is correct, "if and only if every requirement stated therein is one that the software shall meet". SRSs that do not describe actual needs of a system are considered incorrect. Such defects reflect misunderstanding of user's intention and can be identified by a user and/or by comparing RUCM use case specifications with other documents (e.g., contract documents). ***Incompleteness (C2)***. As defined in IEEE Std. 830-1998 [2], a SRS is complete if and only if it includes all following elements: 1) all significant requirements

```
Use Case Model = Use Case Diagram, Use Case Specification;
Use Case Diagram = UCD-UseCase, UCD-Actor, UCD-Relationship;
UCD-UseCase = {UCD::Use Case}-;
UCD-Actor = {UCD::Actor}-;
UCD-Relationship = {Association}-, {UCD-Generalization | UCD-Include | UCD-Extend}
Use Case Specification = UC-Head, Basic Flow, {Alternative Flow};
UC-Head = Use Case, Pre-condition, Actor, {Dependency}, {Generalization};
Use Case = UC-Name, UC-BriefDescription;
UC-Name = use case name;
UC-BriefDescription = brief description of the use case;
Actor = {Prim-Actor}-, {Sec-Actor};
Prim-Actor = primary actor name;
Sec-Actor = secondary actor name;
Dependency = Include | Extend;
Generalization = {UC-Name};
Pre-condition = precondition description of the use case;
Include = "INCLUDE USE CASE", {UC-Name}-;
Extend = "EXTENDED BY USE CASE", {UC-Name}-;
Basic Flow = {Flow-name}, {Action Step}-, {Condition Step |
  MEANWHILE | VALIDATES THAT}, Post-condition;
Alternative Flow = Flow-name, RFS, {Action Step}-, {Condition
  Step | MEANWHILE}, (RESUME STEP | ABORT), Post-condition;
Flow-name = the name of a RUCM flow;
Action Step = RUCM action sentence;
MEANWHILE = Action Step, "MEANWHILE", Action Step;
VALIDATES THAT = "VALIDATES THAT", Action Step;
Condition Step = IF-ELSE-ENDIF | DO-UNTIL;
RFS = "RFS", flow-name, Step-index;
Step-index = int, {"-", int};
IF-ELSE-ENDIF = IF-THEN-ENDIF | IF-THEN-ELSE-ENDIF | IF-THEN-
  ELSEIF-THEN-ENDIF | ELSE-ENDIF | ELSEIF-THEN-ENDIF;
IF-THEN-ENDIF = "IF", condition, "THEN", {Action step}-,
  "ENDIF";
IF-THEN-ELSE-ENDIF = "IF", condition, "THEN", {Action step}-,
  "ELSE", {Action step}-, "ENDIF";
IF-THEN-ELSEIF-THEN-ENDIF = "IF", condition, "THEN", {Action
  step}-, "ELSEIF", condition, "THEN", {Action step}-, "ENDIF";
ELSE-ENDIF = "ELSE", {Action step}-, "ENDIF";
ELSEIF-THEN-ENDIF = "ELSEIF", Condition, "THEN", {Action
  step}-, "ENDIF";
DO-UNTIL = "DO", {Action step}-, "UNTIL", condition;
RESUME STEP = "RESUME STEP", Step-index;
Condition = RUCM condition sentence;
ABORT = "ABORT";
Post-condition = post-condition description of the use case;
RUCM action sentence = a RUCM Sentence describing system action;
RUCM condition sentence = a RUCM Sentence describing conditions;
RUCM Sentence = a descriptive sentence following RUCM writing rules (e.g., simple present tense, forbidden usage of adverbs, adjectives, pronouns, synonyms and negatives);
int = digital-'0', {digital};
digital = '0'|'1'|'2'|'3'|'4'|'5'|'6'|'7'|'8'|'9';
```

Figure 3. Syntax of RUCM Models in EBNF

(e.g., functionality, performance, design constraints, attributes or external interfaces), 2) definition of the response of the system to all input data in all situations, and 3) full labels and references to the figures, tables and diagrams used in the SRS. In context of RUCM models, any omission of RUCM model elements (e.g., *Action Step*) leads to the incompleteness of a RUCM model. For example, missing a use case directly results in the incomplete specification of system functionalities. The RUCM methodology itself (to certain extent) reduces the possibility of specifying incomplete SRSs by e.g., enforcing the specification of various types of alternative flows.

Inconsistency (C3). In IEEE Std. 830-1998 [2], consistency refers to internal consistency, implying that there is no inconsistency among specified individual requirements. For RUCM, we focus on the consistency between the use case diagram and the use case specifications of a RUCM model, and the internal consistency of a use case specification, e.g., two steps or flows of events conflict to each other. ***Ambiguity (C4)***. A SRS is unambiguous if and only if every specified requirement has only one interpretation [2]. To eliminate such defects, it requires that requirements engineers use consistent terminologies when specifying RUCM models. As discussed in Section 2, RUCM defines 26 restriction rules, which help to reduce ambiguities in use case models to certain extent. ***Incomprehensibility (C5)***. This category is borrowed from [15, 27, 28], where it states that a use case model should be understood easily. In case of RUCM, it addresses that each specified requirement should be comprehensible to different stakeholders such as requirements engineers and developers.

Intestability (C6). A requirement is verifiable, if and only if "there exists a finite process for a person or a machine to determine if the system meets the requirement" [2]. Apparently, any ambiguous requirement is unverifiable. RUCM has the inherent mechanism (i.e., *Pre-condition* and *Post-condition*s) to help the verification of each flow of events. It is important to mention that RUCM models have been used as inputs to derive tests [13, 14]. When deriving the RUCM defect taxonomy, we therefore put our focus on *Pre-condition*s and *Post-condition*s of RUCM specifications from the perspective of test engineers. ***Unmodifiability (C7)***. A SRS is modifiable if and only if any change to it can be made "easily, completely and consistently while retaining its structure and style" [2]. For example, in the context of RUCM, a piece of shared behavior should be specified as a use case and reused (via *Include* and *Extend*), rather than be specified in different places of the use case model. ***Infeasibility (C8)***. It is also used to characterize unverifiable requirements from the perspective of system developers. More specifically, this category focuses more on requirements that cannot be implemented from the viewpoint of system developers. ***Over-Specification (C9)***. This category is from [16, 28] and it indicates that the given information in the use case models is irrelevant. In our context, it states that the RUCM use case specification should not contain unnecessary information.

4.2.2 RUCM Defect Taxonomy

The proposed definitions in Section 4.2.1 can not only be used as quality indicators for evaluating RUCM models, but also as a classification of defects. In this section, we present the RUCM defect taxonomy with in total 94 defects defined (Table 1). The taxonomy is systematically defined based on the classification presented in Section 4.2.1.

We examined that all the RUCM elements defined in the EBNF have been considered from the nine aspects (i.e., the nine categories of the classification) and plausible matches (between a category and a RUCM model element) are presented in Table 1. These defects range from structural defects of use case models (e.g., relationships between use cases) to semantic defects of use case specifications (e.g., Ambiguity (Section 4.2.1)). We encode each defect as below:

```
Defect = Defect Prefix, Defect ID
Defect Prefix = Defect Category, Defect Element;
Defect Category = C1 | C2 | C3 | C4 | C5 | C6 | C7 | C8 | C9;
Defect Element = AR | UC | R | H | F | UCM | UCS;
Defect ID = '1'|'2'|'3'|'4'|'5'|'6'|'7'|'8'|'9';
AR = Actor; UC = Use Case;
R = relationship between two model elements;
H = the head of a use case specification;
F = flow-of-events of a use case specification;
UCM = use case model; UCS = use case specification;
```

As shown in Table 1, we divide the taxonomy into two parts: one for use case diagrams and the other for use case specifications. As shown in the table, most of the defects are for use case specifications, as they describe details of functionalities that use cases are intended to specify. Inconsistency (C3) is not applicable to use case diagrams as our context does not concern use case modeling at different levels of abstractions and does not address inconsistencies of various UML diagrams or models. However, it is indeed important to capture inconsistencies between a use case

diagram and its corresponding use case models (C3UCM1). Intestability (C6) and Infeasibility (C8) are also not applicable to use case diagrams as they do not contain detailed descriptions of system behaviors.

We further divide the use case specification category into two parts: one for UC-Head and the other for Flow-of-Events. UC-Head elements (Figure **3**) includes UC-Name, UC-BriefDescription, Pre-condition, Prim-Actor, Sec-Actor, Dependency (i.e., *Include* and *Exclude*, with other use cases), and Generalization (with other use cases). These elements correspond to the first section of the RUCM template (Figure 1). All the defects defined for UC-Head are for these model elements such as Missing Primary Actor (C2H1). As UC-Head does not specify behaviors and therefore *Infeasibility* (C8) is not applicable to them. Since Pre-condition is included as part of UC-Head, *Intestability* is still applicable (C6H1).

As flows of events describe behavior of a use case specification, defects for them focus on all the nine categories (C1-C9). Involved elements include Basic Flow, Alternative Flow, Action Step, Post-condition, RFS (connecting alternative flows to their reference flows), special sentences containing RUCM keywords (e.g., RESUME), the sequence of *Step*s in a Flow-of-Events, and Sentence. Note that there are seven defects that are related to either general aspects of use case specifications (i.e., C2UCS1, C3UCS1-C3UCS4) or aspects concerning multiple model elements of flows of events (i.e., C7UCS1).

Defect C3UCM1 concerns the consistency between the use case

Table 1. RUCM Defect Taxonomy (In Total 94 Types of Defects)

Category	Use Case Diagram (UCD)			Use Case Specification (UCS)	
	Actor	**Use Case**	**Relationship**	**UC-Head**	**Flow-of-Events**
C1	**C1AR1:** Incorrect Actor	**C1UC1:** Incorrect Use Case	**C1R1:** Incorrect Generalization between Actors **C1R2:** Incorrect Generalization between Use Cases **C1R3:** Incorrect Include between Use Cases **C1R4:** Incorrect Extend between Use Cases **C1R5:** Incorrect Association between Actor and Use Case	**C1H1:** Incorrect Primary Actor **C1H2:** Incorrect Secondary Actor **C1H3:** Incorrect Brief Description **C1H4:** Incorrect Pre-condition **C1H5:** Incorrect Include **C1H6:** Incorrect Extend **C1H7:** Incorrect Generalization	**C1F1:** Incorrect Basic Flow **C1F2:** Incorrect Alternative Flow **C1F3:** Incorrect numbering of steps **C1F4:** Incorrect Post-condition **C1F5:** Incorrect branching of alternative flow **C1F6:** Incorrect merging of alternative flow **C1F7:** Incorrect logical relationship
C2	**C2AR1:** Missing Actor	**C2UC1:** Missing Use Case	**C2R1:** Missing Generalization between Actors **C2R2:** Missing Generalization between Use Cases **C2R3:** Missing Include **C2R4:** Missing Extend **C2R5:** Missing Association between Actor and Use Case	**C2H1:** Missing Primary Actor **C2H2:** Missing Secondary Actor **C2H3:** Missing Brief Description **C2H4:** Missing Pre-condition **C2H5:** Missing Include **C2H6:** Missing Extend **C2H7:** Missing Generalization	**C2F1:** Missing Alternative Flow **C2F2:** Missing Step **C2F3:** Missing RFS **C2F4:** Missing post-condition **C2F5:** Missing RESUME **C2F6:** Missing ABORT **C2F7:** Missing logical relationship
				C2UCS1: Missing UCS	
C3	-			**C3H1:** Actor is inconsistent with its behavior in use cases. **C3H2:** Brief Description is inconsistent with the design intent. **C3H3:** Pre-condition is inconsistent with the design intent.	**C3F1:** Basic Flow is inconsistent with its expected goal. **C3F2:** Alternative Flow is inconsistent with its expected goal. **C3F3:** Inconsistent definition of references flows in an alternative flow. **C3F4:** Post-condition is inconsistent with its expected goal. **C3F5:** The numbing of steps is inconsistent.
				C3UCS1: UCS is inconsistent with the design intent. **C3UCS2:** Inconsistent terminologies used in the UCS. **C3UCS3:** Description in the UCS conflicts with another one. **C3UCS4:** UCS should be documented in a consistent level of abstraction.	
	C3UCM1: UCD is inconsistent with its corresponding UCSs.				
C4	**C4AR1:** The actor name does not reflect its role.	**C4UC1:** The use case name does not reflect its goal.	-	**C4H1:** Ambiguous Brief Description **C4H2:** Ambiguous Pre-condition	**C4F1:** Ambiguous sentence in Basic Flow. **C4F2:** Ambiguous sentence in Alternative Flow. **C4F3:** Ambiguous Post-condition.
				C4UCS1: Not using the simple present tense throughout the UCS causes ambiguity. **C4UCS2:** Not avoiding the usage of adverbs, adjectives, pronouns, synonyms and negatives causes ambiguity.	
C5	**C5AR1:** Incomprehensible actor name	**C5UC1:** Incomprehensible use case name	-	**C5H1:** Incomprehensible Brief Description **C5H2:** Incomprehensible Pre-condition	**C5F1:** Incomprehensible sentence of Flow-of-Events. **C5F2:** Incomprehensible Post-condition.
C6	-			**C6H1:** The Pre-condition can never be satisfied.	**C6F1:** The behavior of Flow-of-Events can never be measured. **C6F2:** The Post-condition can never be measured. **C6F3:** The Flow-of-Events should be terminated reasonably.
C7	-			**C7UCS1:** Alternative flows should be separated from Basic Flow.	
	C7UCM1: A shared system functionality should be specified as a separate use case and associated to others.				
C8	-			-	**C8F1:** The behavior described in Flow-of-Events cannot be implemented.
C9	**C9AR1:** Superfluous actor	**C9UC1:** Superfluous use case	**C9R1:** Superfluous Generalization between Actors **C9R2:** Superfluous Generalization between Use Cases **C9R3:** Superfluous Include **C9R4:** Superfluous Extend **C9R5:** Superfluous Association between Actor and Use Case	**C9H1:** Superfluous secondary actor **C9H2:** Superfluous sentences in Brief Description **C9H3:** Superfluous sentences in Pre-condition **C9H4:** Superfluous Include **C9H5:** Superfluous Extend **C9H6:** Superfluous Generalization (with other use cases)	**C9F1:** Superfluous alternative flow **C9F2:** Superfluous step **C9F3:** Superfluous sentence in Post-condition.

diagram and the use case specifications. C7UCM1 relates to the relationships between use cases. Therefore, these two defects are defined at the scope of the whole use case model.

4.3 RUCM Mutation Operators

Mutation analysis heavily depends on mutants, which are syntactic modifications of original programs [31], and mutation operators play the role similar to transformation rules for generating mutants from the original programs. We define our mutation operators for RUCM by following a different strategy as RUCM models are requirements, which inherently have significant differences in terms of generating mutants. In the rest of this section, we first present the HAZOP guide words (Section 4.1), followed by the mechanism we used for systematically deriving the RUCM mutation operators (Section 4.3.2). In Section 4.3.3, we present the derived mutation operators.

4.3.1 Hazard and Operability Study (HAZOP)

HAZOP is a technique widely used to examine a process, operation, design, and human error to identify hazards in various contexts such as developing safety critical systems by analyzing deviations from design intent [17]. The design intent is a baseline for applying HAZOP and should be correct and complete as much as possible [17]. In our context, the design intent is users' actual needs. A successful application of HAZOP mainly relies on two aspects [17]: 1) the identification of system elements (e.g., sensor) and their parameters (e.g., pressure, time), and 2) a set of predefined guide words, which should be applied to each parameter of each system element to identify unexpected and yet reasonable deviations from the design intent. As defined in the IEC 61882 standard [17] the basic guide words of HAZOP include NO, MORE, LESS, AS WELL AS, PART OF, REVERSE and OTHER THAN. We adapt these guide words for RUCM and define each of them in Table 2.

Table 2. Adapted definitions of the HAZOP guide words

Guideword	Definition
NO	An intended RUCM model element is not captured. A RUCM modeling intent is not achieved.
MORE	Quantitative increase in a quantifiable RUCM element.
LESS	Quantitative decrease in a quantifiable RUCM element.
AS WELL AS	Spurious RUCM model elements or behavior are included in a RUCM model.
PART OF	Incomplete RUCM modeling intent is achieved. Incomplete model element or behavior is captured.
REVERSE	A logical opposite of the RUCM modeling intent, behavior or model element is specified.
OTHER THAN	A RUCM model is substituted by unintended/incorrect behavior, intent or model element.

4.3.2 Mechanism for Deriving Mutation Operators

We apply the seven HAZOP guide words and identify RUCM elements, to which the guide words are applicable. In our mechanism, we map each HAZOP guide word to each RUCM model element and we derive the RUCM mutation operators by analyzing the corresponding consequences of applying them. We illustrate the mechanism we use for deriving RUCM mutation operators in Figure 4.

As shown in Figure 4, our mechanism is five-dimensional: 1) the seven HAZOP guide words, 2) the RUCM elements, 3) the nine defect categories of the RUCM defect taxonomy (e.g., *Incorrectness (C1)*), 4) the six editing operations (i.e., ADD (for addition), DEL (for deletion), SWAP (for exchanging two compatible elements within the RUCM model), REP (for replacing an element with another compatible element not existing in the RUCM model), ICR (for quantitative increase) and DEC (for quantitative decrease)), and 5) the defects in the RUCM defect taxonomy, which determine the consequences of the application of an editing operation to a particular RUCM element. It is worth mentioning that this mechanism ensures that each RUCM mutation operator leads to exactly one type of defects defined in our defect taxonomy, as each mutation operator is composed of three parts: an editing operation, a RUCM model element on which the mutation operator is applicable, and the type of a defect that the mutation operator can lead to. The HAZOP guide words help to systematically associate an editing operation to a defect and defect category for a specific RUCM model element.

For example, as shown in Figure 4, we derive the mutation operator 'DEL-AF-C2F1' with the following steps. First, we identify the RUCM element that a mutation operator should be applied on, which results in an alternative flow shown in the RUCM model axis in Figure 4. Second, we identify the editing operator that can lead to a NO derivation from the original RUCM model (shown on the HAZOP guide word axis). Third, applying the editing operation of DEL on the alternative flow leads to deletion of it from the original RUCM use case specification, which directly results in the defect of missing the alternative flow (C2F1 of the RUCM defect taxonomy and follows into the *Incompleteness* defect category (shown as the RUCM defect taxonomy category axis of Figure 4). We therefore define the mutation operator as 'DEL-AF-C2F1'.

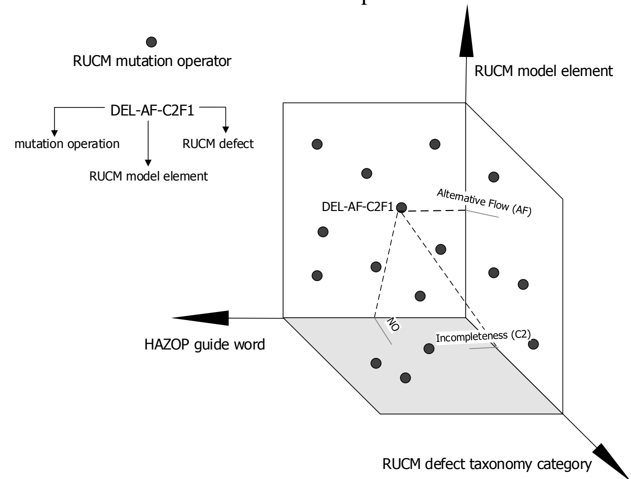

Figure 4. Mechanism for deriving RUCM mutation operators

Note that we carefully analyzed which HAZOP guide word is applicable to which RUCM elements. For example, element 'Use Case' refers to a lexical value: its name (i.e. UC-Name); therefore, guide word OTHER THAN is applicable for this element by replacing the use case name by modifying the lexical value. RUCM elements RESUME STEP and ABORT refer to action steps containing these two keywords. Since these two elements can only be applied in the alternative flows and are mutually exclusive, we thus can apply guide word OTHER THAN to substitute one with the other. In case of Flow-of-Events, both PART OF and AS WELL AS can be used to generate deviations, since 'Flow-of-Events' is the container of RUCM steps.

Table 3. Mutation operators for Use Case Diagrams (29)

	NO	AS WELL AS	REVERSE	OTHER THAN
UCD::Use Case	DEL-UC-C2UC1	ADD-UC-C9UC1	-	REP-UCN-{C1UC1, C4UC1, C5UC1}
UCD::Actor	DEL-AR-C2AR1	ADD-AR-C9AR1	-	REP-ARN-{C1AR1, C4AR1, C5AR1}
UCD::Include	DEL-INC-C2R3	ADD-INC-C9R3	SWAP-INC-C1R3	REP-INC-C1R3
UCD::Extend	DEL-EXD-C2R4	ADD-EXD-C9R4	SWAP-EXD-C1R4	REP-EXD-C1R4
UCD::Generalization	DEL-AR-C2R1 DEL-UC-C2R2	ADD-AR-C9R1 ADD-UC-C9R2	SWAP-GAR-C1R1 SWAP-GUC-C1R2	REP-GAR-C1R1 REP-GUC-C1R2
UCD::Association	DEL-ASSO-C2R5	ADD-ASSO-C9R5	-	REP-ASSO-C1R5

Table 4. Mutation operators for UC-Head (34)

	NO	AS WELL AS	PART OF	REVERSE	OTHER THAN
UCS::Use Case	-	-	-	-	-
UCS::Actor	DEL-PAR-C2H1 DEL-SAR-C2H2	ADD-SAR-C9H1	-	SWAP-AR-{C1H1, C1H2}	REP-PAR-C1H1 REP-SAR-C1H2
UCS::Brief Description	DEL-BD-C2H3	ADD-SenBD-C9H2	DEL-SenBD-C1H3 REP-SenBD-C1H3	-	REP-BD-{C1H3, C3H2, C4H1, C5H1}
UCS::Include	DEL-INC-C2H5	ADD-INC-C9H4	-	-	REP-INC-C1H5
UCS::Extend	DEL-EXD-C2H6	ADD-EXD-C9H5	-	-	REP-EXD-C1H6
UCS::Generalization	DEL-GUC-C2H7	ADD-GUC-C9H6	-	-	REP-GUC-C1H7
UCS::Precondition	DEL-PreC-C2H4	ADD-SenPreC-{C9H3, C6H1}	DEL-SenPreC-{C1H4, C6H1}	-	REP-PreC-{C1H4, C3H3, C4H3, C5H2, C6H1}

Table 5. Mutation operators for Flows-of-Events (54)

	NO	AS WELL AS	PART OF	REVERSE	OTHER THAN
Flow of Events	-	ADD-AF-C9F1	DEL-AF-C2F1	SWAP-AF-{C1F5, C3F2}	-
Basic Flow	-	ADD-SenBF-C9F2	DEL-SenBF-{C1F1, C2F2, C3F1, C3F5, C6F1, C6F3, C8F1}	SWAP-SenBF-{C1F1, C1F3, C3F1, C6F1, C6F3, C8F1}	REP-BF-C1F1
Alternative Flow	DEL-AF-C2F1	ADD-SenAF-C9F2	DEL-SenAF-{C1F2, C2F2, C3F2, C3F5, C6F1, C6F3, C8F1}	SWAP-SenAF-{C1F2, C1F3, C3F2, C6F1, C6F3, C8F1}	REP-AF-C3F2
Post-condition	DEL-PostC-C2F4	ADD-SenPostC-{C9F3, C6F2}	DEL- SenPostC-C1F4 REP- SenPostC-C6F2	-	REP-PostC-{C1F4, C3F4, C4F3, C5F2, C6F2
Action Step	DEL-AS-C2F2	-	-	-	REP-AS-{C4UCS1, C4UCS2, C4F1, C4F2, C5F1, C6F1, C6F3, C6F3}

Table 6. Mutation operators for Sentences containing keywords (72)

	NO	MORE	LESS	AS WELL AS	PART OF	REVERSE	OTHER THAN
RFS	DEL-RFS-{C2F3, C6F1}	ICR-RFSsi-{C1F5, C3F3}	DEC-RFSsi-{C1F5, C6F1}	ADD-RFSflow-{C3F2, C3F3}	DEL-RFSflow-C3F2	SWAP-RFS-C1F5	REP-RFS-C3F3
RESUME STEP	DEL-RES-C2F5	ICR-RESsi-C1F6	DEC-RESsi-C1F6	-	-	SWAP-RES-C1F6	REP-RES- C1F6 REP-toABT-C6F3
ABORT	DEL-ABT-C2F6	-	-	-	-	-	REP-toRES-C3F2
IF-ELSE-ENDIF	DEL-IFELSE-C2F7	-	-	ADD-IFELSEcs-{C1F7, C6F1, C8F1} ADD-IFELSEas-C9F2	DEL-IFELSEcs-C1F7 DEL-IFELSEas-{C1F1, C1F2 C2F2, C3F1, C3F2 C3F5, C6F1, C6F3, C8F1}	SWAP-IFELSE-{C1F1, C6F1}	REP-IFELSE -{C1F1, C6F1}
DO-UNTIL	DEL-DO-C2F7	-	-	ADD-DOcs-{C1F7, C6F1, C8F1} ADD-DOas-C9F2	DEL-DOcs-C1F7 DEL-DOas-{C1F1, C1F2 C2F2, C3F1, C3F2 C3F5, C6F1, C6F3, C8F1}	-	REP-DO-{C1F1, C6F1}
VALIDATES THAT	DEL-VLD-C1F5	-	-	-	-	REP-VLD-{C6F1, C8F1}	REP-VLD-{C6F1, C8F1}
MEANWHILE	DEL-MW-{C3F1, C3F2}	-	-	-	-	SWAP-MW-{C1F1, C1F2, C3F1, C3F2, C6F1, C8F1}	REP-MW-{C1F1, C1F2, C3F1, C3F2, C6F1, C8F1}

Each application of a guide word to a particular RUCM element is implemented as editing the original RUCM model. We define six types of editing operations: adding an element (ADD), deleting an element (DELETE), swapping two existing elements (SWAP), replacing an element with another (REP), increasing/decreasing the quantitative value of an element (ICR/DEC).

4.3.3 Mutation Operators

All the RUCM mutation operators are given in Table 3-Table 6. We first explain the encoding mechanism of naming the mutation operators:

```
RUCM mutation operator = Operation, -, Element, -, {Defect}-;
Operation = ADD | DEL | SWAP | REP | ICR | DEC;
Element = UC | AR | INC | EXD | ASSO | UCS | UCN | ARN | PAR |
  SAR | BD | GA | GUC | SenBD | PreC | SenPreC | AF | SenAF |
  BF | SenBF | PostC | SenPostC | AS | RFS | ABORT | IFELSE |
  DO | VLD | MW |RFSsi | RFSflow | toABT | RES | toRES | IFELSE
  | IFELSEcs | IFELSEas | DOcs | DOas;
Defect = Defect Prefix, Defect ID
Defect Prefix = Defect Category, Defect Element;
Defect Category = C1 | C2 | C3 | C4 | C5 | C6 | C7 | C8 | C9;
Defect Element = AR | UC | R | H | F | UCM | UCS;
Defect ID = '1'|'2'|'3'|'4'|'5'|'6'|'7'|'8'|'9';
UC = Use Case; AR = Actor; INC = Include; EXD = Extend;
ASSO = Association; PAR = primary actor; SAR = secondary actor;
UCS = use case specification; UCN = use case name;
UCM = use case model; BD = use case brief description;
ARN = Actor name; GAR = generalization between actors;
GUC = generalization between use cases;
SenBD = sentence in BD;
PreC = pre-condition; SenPreC = sentence in PreC;
BF = Basic Flow; SenBF = sentence in BF;
AF = Alternative Flow; SenAF = sentence in AF;
```

PostC = post-condition; **SenPostC** = sentence in PostC;
AS = Action step; **VLD** = VALIDATES THAT; **MW** = MEANWHILE;
RFS = sentence RFS; **RFSsi** = step index of RFS;
RFSflow = reference flow name of RFS;
ABORT = sentence ABORT; **toRES** = changing ABORT to RESUME STEP;
RES = RESUME STEP; **toABT** = changing RES to ABORT;
IFELSE = IF-SELSE-ENDIF; **IFELSEcs** = condition sentence in IF-ELSE-ENDIF; **IFELSEas** = Action Step in IF-ELSE-ENDIF;
DO = DO-UNTIL; **DOcs** = condition sentence in DO-UNITL;
DOas = Action step in DO-UNITL;

A combination of a HAZOP guide word and a RUCM element could introduce different types of defects, which therefore leads to the definition of various mutation operators. To save space, in tables Table 3-Table 6, we use the '{ }' to embrace all corresponding mutation operators. For example, REP-UCN-{C1U1, C4U1, C5U1} (Table 3) denotes three mutation operators, i.e., REP-UCN-C1U1, REP-UCN- C4U1, REP-UCN- C5U1.

Table 3 presents all the 29 mutation operators derived for use case diagrams. One example is SWAP-INC-C1R3, which indicates the application of the REVERSE guide word to the RUCM element *Include*, meaning that the swapping of the roles of two use cases connected by the *Include* relationship (from being included into including, or vice versa) results in C1R3 of the *Incorrectness* category. Note that guide words MORE and LESS are not applicable to use case diagrams, as they are for quantifiable RUCM elements. AS WELL AS is also not applicable as all the RUCM elements in use case diagrams do not contain other elements.

Table 4 shows all the 34 mutation operators derived for the UC-Head of a RUCM use case specification, i.e., participating actors, the relationships with other use cases, its brief description and the pre-condition. Same as for use case diagrams, guide words MORE and LESS are not applicable. Since the brief description and pre-condition of a RUCM use case specification can contain one or more sentences, guide word PART OF is applicable when DEL or REP are used to modify the original use case specification.

We report all the 54 mutation operators derived for RUCM flows of events in Table 5. Same as for use case diagrams and UC-Heads, guide words MORE and LESS are not applicable. Table 6 reports all the 72 mutation operators derived for sentences containing RUCM keywords (e.g., VALIDATES THAT) in use case specifications. Since keywords 'RFS' and 'RESUME STEP' are followed by step numbers of flows of events, guide words MORE and LESS are then applicable, consequences of which are implemented by applying editing operations ICR and DEC on the step numbers of the sentences containing RFS and RESUME STEP.

4.4 Guidelines for Defect Seeding Strategies

Mutant reduction is an important issue for practical application of mutation analysis [31]. As summarized in [10], four techniques are widely used for reducing the total number of mutants: *Mutant Sampling*, *Mutant Clustering*, *Selective Mutation*, and *Higher Order Mutation*. *Mutation Sampling* randomly chooses a small subset of mutants. *Mutant Clustering* applies clustering algorithms to perform selection. *Selective Mutation* aims to reduce the number of mutants by selecting a subset of mutation operators to apply based on certain selection criteria. *Higher Order Mutation* aims to find rare but valuable higher order mutants. Inspired by the literature, we define a set of guidelines for defining a cost-effective defect seeding strategy below.

Guideline 1: Defects might have different levels of *Importance* at different contexts. For example, C1F5 (incorrect branching of alternative flows from the basic flow) might be considered more important than C1R5 (incorrect association between an actor and a use case) from the perspective of developers. The property of *Importance* of a defect (and therefore a mutation operator) can be used for implementing a *Selective Mutation* strategy in the context of the RUCM mutation analysis. One can also define *Importance* for the defect categories. For example, in certain contexts, *Incompleteness* is considered more important than *Unmodifiability*.

Guideline 2: Each RUCM mutation operator is associated with exactly one type of defects (Section 4.3.3), implying that each mutant corresponds to a specific defect type. In the context of RUCM mutation operators, there is no possibility of producing *Equivalent Mutants* that are syntactically different but semantically identical to the original RUCM model from which they are created [10]. This is because of the reason that we do not execute the specifications as is the case of code.

Guideline 3: Some defects can only be seeded manually and effort required to apply the mutation operators to seed defects might be different. For example, it is more difficult to apply 'ADD-AF-C9F1' than 'DEL-SAR-C2H2', because the former needs to insert a complete alternative flow to a use case specification while the latter just deletes one of the secondary actors from a use case specification. Therefore, defining a cost-effective defect seeding strategy by using the RUCM mutation operators is critical. Cost is mainly about manual effort required to seed defects and effectiveness is about the effectiveness of killing mutants of a particular requirements inspection approach under evaluation. Cost can be reduced by applying various mutation reduction techniques such as *Selective Mutation*.

Guideline 4: One defect type might be covered by multiple mutation operators applying on different RUCM model elements. For example, C1F3 (incorrect numbering of steps) can be realized by both SWAP-SenBF-C1F3 (swapping two sentences of the basic flow) and SWAP-SenAF-C1F3 (swapping two sentences of an alternative flow). So, it is important to find a defect seeding strategy to eliminate *Redundant Mutants* that are redundant if their outcomes are the same as with other mutants, or can be derived/predicted based on the outcomes of other mutants [32]. For example, one defect seeding strategy is to include either SWAP-SenBF-C1F3 or SWAP-SenAF-C1F3, but not both. One can also define a more coarse-grained strategy by treating all the mutation operators leading to various defects (e.g., DEL-UC-C2UC1 and DEL-PostC-C2F4) of the same category as redundant defects.

Guideline 5: Our mechanism for deriving the RUCM mutation operators (Section 4.3.2) can enable easy selection of RUCM mutation operators based on the nine defect categories. All the RUCM mutation operators are naturally clustered into nine categories, each of which corresponds to a defect category of the RUCM defect taxonomy. Depending on the inspection problem at hand, we can only select the category required for inspection. In addition, we can also select the RUCM mutation operators based on RUCM model elements, on which each mutation operator is applied. For example, if a requirements inspection method only focuses on inspecting a use case specification, then mutation operators defined for use case diagrams (presented in Table 3) should be excluded.

5. EVALUATION

5.1 Case Studies

To evaluate the proposed approach, one real world case study and six other case studies from the literature are used in our evaluation. Their characteristics are summarized in Table 7. As shown in Table 7, in total, we managed to mutate 38 use cases as indicated by the number of *Basic Flow* as each use case has exactly one basic flow.

Our industrial case study is a *Navigation System* (NAS), which controls and guides an aircraft, based on control law computation that takes data sampled from sensors as input and sends commands to actuators. To evaluate our approach, we selected 11 use cases of the system as the original requirements. Notice that the case study has been used to evaluate our previous work [33]. The other six case studies are ATM (Banking System) [18], CMS (Crisis Management System) [34], CPD (Car Part Dealer System) [19], VS (Video Store system) [19], CDS (Cab Dispatching system) [19] and PAY (Payroll System) [19]. These case studies have been used to in our previous work ([12, 19]) to evaluate the RUCM methodology and the transformation from RUCM to UML analysis models.

Table 7. Characteristics of the Case Studies*

Model Element	Case Study							Total
	S1	S2	S3	S4	S5	S6	S7	
#Actor	7	2	2	4	8	3	4	30
#Include	4	3	0	2	5	0	4	18
#Extend	2	0	0	0	0	0	0	2
#Generalization	2	0	0	0	0	0	0	2
#Association	7	4	2	4	8	3	6	34
#Basic Flow	11	4	1	6	7	1	8	38
#Alternative Flow	28	7	7	12	12	3	12	81
#RFS	28	7	7	12	12	3	12	81
#RESUME STEP	20	1	7	5	7	4	8	52
#ABORT	8	7	1	8	5	0	9	38
#IF-ELSE-ENDIF	23	3	12	8	2	6	12	66
#DO-UNTIL	2	0	3	6	3	1	0	15
#VALIDATES THAT	36	10	1	5	11	3	14	80
#MEANWHILE	8	1	1	1	2	2	0	15
#Pre-condition	11	4	1	6	7	1	8	38
#Post-condition	39	11	8	18	19	4	20	119

*S1: NAS, S2: ATM, S3: CMS, S4: CPD, S56: NGP, S6: PAY, S7: VS

5.2 Research Questions

Our evaluation aims to answer the following three research questions: ***RQ1***: How complete is the proposed defect taxonomy as compared to existing ones? ***RQ2***: Are all the RUCM mutation operators feasible to apply? ***RQ3***: Can the generated mutants be useful in terms of evaluating various coverage criteria for generating use case scenarios from use case models?

5.3 Results and Discussion

5.3.1 Results for RQ1

RQ1 aims at answering if the RUCM defect taxonomy is complete in terms of covering defects commonly observed in use case models. To answer this question, we compared the RUCM defect taxonomy with the four taxonomies in the literature: [15, 16, 27, 28]. Results show that the RUCM defect taxonomy is complete as it covers all the defects classified in the four taxonomies. In addition, the RUCM defect taxonomy is applicable to entire use case models including both use case diagrams and use case specifications. Moreover, our defect taxonomy either defines additional defect types or provides more precisely definitions as compared with these four taxonomies. Though the taxonomy proposed by Anda and Sjøberg [16] is also applicable for entire use case models, the RUCM defect taxonomy defines more defect types, e.g., *Intestability* and each defect type has a precise definition tightly related to RUCM elements. Moreover the RUCM defect taxonomy was systematically defined by following IEEE Std. 830-1998 [2] (Section 4.2). Though the taxonomy in [28] was proposed by following the same standard, the RUCM taxonomy provides a more precisely definition for each defect type and it is more preferable for investigating different inspection.

5.3.2 Results for RQ2

We present all the defects/mutants seeded for the seven use case models of the seven case studies in Table 8. We derived in total 5588 mutants, including 1279 *Incorrectness (C1)* defects, 1048 *Incompleteness (C2)* defects, 633 *Inconsistency (C3)* defects, 871 *Ambiguity (C4)* defects, 499 *Incomprehensibility (C5)* defects, 328 *Intestability (C6)* defects, 101 *Unmodifiability (C7)* defects, 81 *Infeasibility (C8)* defects and 748 *Over-Specification (C9)* defects.

From Table 8, one can observe that some defect types can be simulated via multiple mutation operators; some defect types can only be realized by one particular mutation operator; and some types of defects need multiple mutation operators to implement. For example, as shwon in Table 8, the mapping betewen mutation operator `REP-ARN-C1AR1` and the defect type of `C1AR1` is one to one. In our case studies, in total, the muration operator was applied 30 times to generate 30 mutants that lead to the same type of defects, i.e., `C1AR1`. For defect type `C1F3`, it was realized via `SWAP-SenBF-C1F3` and `SWAP-SenAF-C1F3` and we applied them 142 and 264 times respectively, as shown in Table 8. For the specical defect types such as `C7UCM1`, multiple mutation operators should be used together to realize them. In our cases studies, as shown in Table 8, we applied both `ADD-SenBF-C9F2` and `DEL-SenAF-C2F2` to seed defects of the `C7UCS1` type. For the other six types of defects (i.e. `C2UCS1`, `C3UCS1`, `C3UCS2`, `C3UCS3`, `C3UCS4`, and `C3UCM1`), one can choose different mutation operators to realize. For example, we applied `DEL-PAR-C2H1` to realize `C3UCM1`. In general, we simulated 5588 defects by applying all the 189 RUCM mutation operators at least once to cover all the 94 defect types at least once. Results and our experience show that all the RUCM mutation operators are feasible to apply.

5.3.3 Results for RQ3

To answer RQ3, we used the 5588 mutants derived for the seven case studies (RQ2) to evaluate the performance of three coverage criteria for generating use case scenarios from use case models. The three RUCM coverage criteria are *All Conditio*n coverage, *All FlowOfEvents* coverage and *All Sentence* coverage [33]. To evaluate these three coverage criteria, we define *Mutation Score* (MS) as: $MS = (\#\ of\ killed\ mutants)/(\#\ of\ all\ seeded\ mutants)$.

Results of the evaluation are presented in Figure 5. The generated mutants worked well in terms of evaluating the performance (mutation score) of the three coverage criteria. For the C1 defects, the *All Condition* coverage criterion achieved the highest mutation score (0.95), followed by the *All FlowsOfEvents* coverage criterion (0.83) and the *All Sentences* coverage criterion achieved the lowest mutation score 0.67. For other defect categories (C2-C8), we can observe the similar patterns. This is reasonable because the *All Condition* coverage criterion generates more use case scenarios than the other two. These experiment results clearly show that the mutants derived with the RUCM mutation operators worked well for differentiating the performance of the different RUCM use case scenario coverage criteria in terms of mutation scores.

5.3.4 Overall Discussion

DeMillo et al. [11], who initiated the mutation technique, addressed that "formulating a complete set of mutation operators is a necessary requirement for program mutation to be deductive [11]." We believe that the generation of mutation operators by following

Table 8. Results for RQ2

Mutation Operator	Mutants	Mutation Operator	Mutants	Mutation Operator	Mutants	Mutation Operator	Mutants
REP-ARN-C1AR1	30	DEL-AR-C2AR1	30	REP-ARN-C3H1	30	ADD-SenPreC-C6H1	15
REP-UCN-C1UC1	38	DEL-UC-C2UC1	38	REP-BD-C3H2	38	DEL-SenPreC-C6H1	20
SWAP-GAR-C1R1	2	DEL-AR-C2R1	2	REP-PreC-C3H3	38	REP-PreC-C6H1	3
REP-GAR-C1R1	6	DEL-UC-C2R2	30	DEL-SenBF-C3F1	15	DEL-SenBF-C6F1	36
SWAP-GUC-C1R2	6	DEL-INC-C2R3	18	SWAP-SenBF-C3F1	4	SWAP-SenBF-C6F1	8
REP-GUC-C1R2	8	DEL-EXD-C2R4	2	REP-BF-C3F1	2	DEL-SenAF-C6F1	18
SWAP-INC-C1R3	12	DEL-ASSO-C2R5	34	DEL-IFELSEas-C3F1	2	SWAP-SenAF-C6F1	7
REP-INC-C1R3	16	DEL-PAR-C2H1	12	SWAP-IFELSE-C3F1	1	REP-AS-C6F1	1
SWAP-EXD-C1R4	6	DEL-SAR-C2H2	18	REP-IFELSE-C3F1	1	DEL-RFS-C6F1	8
REP-EXD-C1R4	10	DEL-BD-C2H3	38	DEL-DOas-C3F1	4	DEC-RFSsi-C6F1	7
REP-ASSO-C1R5	11	DEL-PreC-C2H4	38	REP-DO-C3F1	2	ADD-IFELSEcs-C6F1	1
SWAP-AR-C1H1	3	DEL-INC-C2H5	18	SWAP-MW-C3F1	3	DEL-IFELSEas-C6F1	14
REP-AR-C1H1	9	DEL-EXD-C2H6	2	REP-MW-C3F1	1	SWAP-IFELSE-C6F1	1
SWAP-AR-C1H2	6	DEL-GUC-C2H7	2	DEL-MW- C3F1	3	REP-IFELSE-C6F1	1
REP-AR-C1H2	12	DEL-AF-C2F1	119	SWAP-AF-C3F2	2	ADD-DOcs-C6F1	1
REP-SenBD-C1H3	18	DEL-AS-C2F2	85	DEL-SenAF-C3F1	22	DEL-DOas-C6F1	12
DEL-SenBD-C1H3	12	DEL-SenBF-C2F2	34	SWAP-SenAF-C3F2	4	REP-DO-C6F1	1
REP-BD-C1H3	8	DEL-SenAF-C2F2	81	REP-AF-C3F2	2	REP-VLD-C6F1	1
DEL-SenPreC-C1H4	25	DEL-IFELSEas-C2F2	30	ADD-RFSflow-C3F2	2	SWAP-MW-C6F1	1
REP-PreC-C1H4	13	DEL-DOas-C2F2	8	DEL-RFSflow-C3F2	13	REP-MW-C6F1	1
REP-INC-C1H5	18	DEL-RFS-C2F3	81	REP-toRES-C3F2	1	ADD-SenPostC-C6F2	86
REP-EXD-C1H6	4	DEL-PostC-C2F4	119	DEL-IFELSEas-C3F2	16	REP-SenPostC-C6F2	24
REP-GUC-C1H7	4	DEL-RES-C2F5	52	DEL-DOas-C3F2	12	REP-PostC-C6F2	9
DEL-SenBF-C1F1	15	DEL-ABT-C2F6	38	SWAP-MW-C3F2	3	DEL-SenBF-C6F3	21
SWAP-SenBF-C1F1	4	DEL-IFELSE-C2F7	66	REP-MW-C3F2	1	SWAP-SenBF-C6F3	4
REP-BF-C1F1	2	DEL-DO-C2F7	15	DEL-MW-C3F2	3	DEL-SenAF-C6F3	7
DEL-IFELSEas-C1F1	2	**C2UC1:** DEL-UC	38	ICR-RFSsi-C3F3	22	SWAP-SenAF-C6F3	2
SWAP-IFELSE-C1F1	1	**Total in C2**	**1048**	ADD-RFSflow-C3F3	8	REP-AS-C6F3	1
REP-IFELSE-C1F1	1	REP-UCN-C4UC1	38	REP-RFS-C3F3	2	REP-toABT-C6F3	3
DEL-DOas-C1F1	4	REP-BD-C4H1	119	REP-PostC-C3F4	119	DEL-IFELSEas-C6F3	7
REP-DO-C1F1	2	REP-PreC-C4H2	119	DEL-SenBF-C3F5	5	DEL-DOas-C6F3	7
SWAP-MW-C1F1	3	REP-AS-C4F1	76	DEL-SenAF-C3F5	5	**Total in C6**	**328**
REP-MW-C1F1	4	REP-AS-C4F2	162	DEL-IFELSEas-C3F5	5	DEL-SenBF-C8F1	25
DEL-SenAF-C1F2	89	REP-PostC-C4F3	119	DEL-DOas-C3F5	21	SWAP-SenBF-C8F1	3
SWAP-SenAF-C1F2	81	REP-AS-C4UCS1	119	***C3UCS1:***REP-UCN-C1UC1	38	DEL-SenAF-C8F1	7
DEL-IFELSEas-C1F2	66	REP-AS-C4UCS2	119	***C3UCS2:***REP-AS-C4F1	119	SWAP-SenAF-C8F1	7
SWAP-MW-C1F2	6	**Total in C4**	**871**	***C3UCS4:***REP-AS-C4F1	26	ADD-IFELSEcs-C8F1	11
REP-MW-C1F2	9	REP-ARN-C5AR1	30	***C3UCM1:***DEL-PAR-C2H1	38	DEL-IFELSEcs-C8F1	7
SWAP-SenBF-C1F3	142	REP-UCN-C5UC1	38	**Total in C3**	**633**	ADD-DOcs-C8F1	7
SWAP-SenAF-C1F3	264	REP-BD-C5H1	38	**C7UCS1:** ADD-SenBF-C9F2, DEL-SenAF-C2F2	81	DEL-DOcs-C8F1	7
DEL-SenPostC-C1F4	26	REP-PreC-C5H2	38			REP-VLD-C8F1	4
REP-PostC-C1F4	12	REP-AS-C5F1	236	**C7UCM1:** ADD-SenBF-C9F2, DEL-SenAF-C2F2, DEL-SenBF-C9F2	20	SWAP-MW-C8F1	1
SWAP-AF-C1F5	21	REP-PostC-C5F2	119			REP-MW-C8F1	2
ICR-RFSsi-C1F5	30	**Total in C5**	**499**	**Total in C7**	**101**	**Total in C8**	**81**
DEC-RFSsi-C1F5	22	DEL-IFELSEcs-C1F7	36	ADD-AR-C9AR1	38	ADD-INC-C9H4	14
SWAP-RFS-C1F5	8	ADD-DOcs-C1F7	15	ADD-UC-C9UC1	7	ADD-EXD-C9H5	7
DEL-VLD-C1F5	15	DEL-DOcs-C1F7	15	ADD-AR-C9R1	8	ADD-GUC-C9H6	12
ICR-RESsi-C1F6	24	ADD-IFELSEcs-C1F7	55	ADD-UC-C9R2	12	ADD-AF-C9F1	76
DEC-RESsi-C1F6	20	ADD-SenBD-C9H2	38	ADD-INC-C9R3	6	ADD-SenBF-C9F2	56
SWAP-RES-C1F6	3	ADD-SenPreC-C9H3	119	ADD-EXD-C9R4	6	ADD-SenAF-C9F2	101
REP-RES-C1F6	5	ADD-SenPostC-C9F3	119	ADD-ASSO-C9R5	30	ADD-IFELSEas-C9F2	66
Total in C1	**1279**	ADD-DOas-C9F2	15	ADD-SAR-C9H1	18	**Total in C9**	**748**

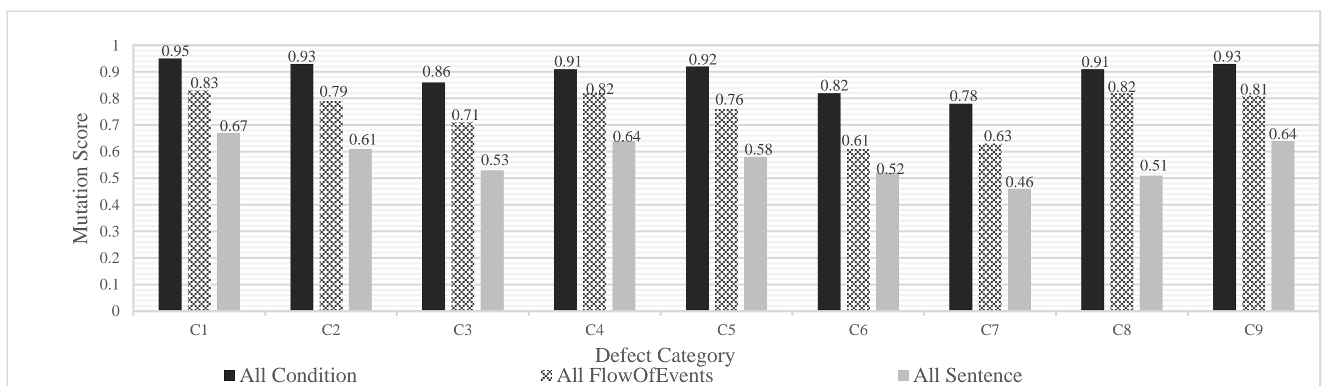

Figure 5. Mutation scores for the three coverage criteria of each defect category

a rigorous approach is the first crucial step for any mutation

analysis and it is especially true for the fault/defect injection based evaluation of use case inspection techniques.

The RUCM mutation operators are designed and expressed specifically for RUCM models; however, they are suitable for other use case models. The mutation operators for use case diagram can be applied directly to any UML use case diagram. The mutation operators for specifications are applicable for other use case templates without requiring significant modification as the RUCM use case template captures the most common constructs of use case templates in the literature. It is worth mentioning that our five dimensional mechanism for deriving the mutation operators is a rigorous method (following the HAZOP [17] and IEEE Std. 830-1998 [2] standards), and it might be useful for other researchers to propose their particular mutation operators. The results of our case studies show that the RUCM mutation operators worked well for generating effective mutants. However further empirical studies are needed in the future.

6. CONCLUSION

Requirements quality is critical for the final success of any non-trivial system development. Requirements inspection is a commonly method applied in practice to ensure requirements quality, especially when they are documented in natural language. Use case modelling is natural language based modeling approach for specifying requirements in a structured way and therefore an increasing attention is given to use case inspection methods. However, in the literature lacks a comprehensive defect taxonomy for use case models and mutation analysis has not been systematically introduced for evaluating various requirements inspection methods. In this paper, we demonstrated a novel and systematic approach to propose a comprehensive defect taxonomy based on a systematic study of the IEEE Std. 830-1998 requirements standard. By applying the hazard and operability study (HAZOP) technique, we rigorously devised a set of mutation operators and provide a set of guidelines on how to devise cost-effective defect seeding strategies with the proposed mutation operators Our approach was evaluated by a set of case studies. Results show that the proposed mutation operators are comprehensive and effective.

7. REFERENCES

[1] Neill, C. J. and Laplante, P. A. 2003. Requirements engineering: the state of the practice. *IEEE software*. 20, 6 (Nov. 2003), 40-45.

[2] IEEE 1998. IEEE Recommended Practice for Software Requirements Specifications. *IEEE Std. 830-1998*.

[3] Fagan, M. E. 1986. Advances in software inspections. *IEEE Transactons on Software Engineering*. 12, 7 (1986), 744-751.

[4] Miller, J., Wood, M., and Roper, M. 1998. Further experiences with scenarios and checklists. *Empirical Software Engineering*. 3, 1 (Marcch 1998), 37-64.

[5] Fagan, M. E. 1976. Design and code inspections to reduce errors in program development. *IBM Systems Journal*. 15, 3 (1976), 182-211.

[6] Porter, A., Votta Jr, L. G., and Basili, V. R. 1995. Comparing detection methods for software requirements inspections: A replicated experiment. *Software Engineering, IEEE Transactions on*. 21, 6 (June 1995), 563-575.

[7] Basili, V. R., Green, S., Laitenberger, O., Lanubile, F., Shull, F., Sørumgård, S., *et al.* 1996. The empirical investigation of perspective-based reading. *Empirical Software Engineering*. 1, 2 (January 1996), 133-164.

[8] Thelin, T., Runeson, P., Wohlin, C., Olsson, T., and Andersson, C. 2004. Evaluation of Usage-Based Reading—Conclusions after Three Experiments. *Empirical Software Engineering*. 9, 1-2 (March 2004), 77-110.

[9] Wohlin, C., Runeson, P., Höst, M., Ohlsson, M. C., Regnell, B., and Wesslén, A. 2012. Are the Perspectives Really Different?: Further Experimentation on Scenario-Based Reading of Requirements. In Eds. Springer, 175-200.

[10] Jia, Y. and Harman, M. 2011. An analysis and survey of the development of mutation testing. *Software Engineering, IEEE Transactions on*. 37, 5 649-678.

[11] DeMillo, R., Lipton, R., and Sayward, F. 1979. Program mutation: A new approach to program testing. *Infotech State of the Art Report, Software Testing*. 2, 107-126.

[12] Yue, T., Briand, L. C., and Labiche, Y. 2013. Facilitating the transition from use case models to analysis models: Approach and experiments. *ACM Transactions on Software Engineering and Methodology (TOSEM)*. 22, 1 (February 2013), 5.

[13] Yue, T. and Ali, S. 2012. Bridging the gap between requirements and aspect state machines to support non-functional testing: industrial case studies. In Eds. Springer, 133-145.

[14] Wang, C., Pastore, F., Goknil, A., Briand, L., and Iqbal, Z. 2015. Automatic generation of system test cases from use case specifications. In *Proceedings of the Proceedings of the 2015 International Symposium on Software Testing and Analysis* ACM, 385-396.

[15] Cox, K., Aurum, A., and Jeffery, R. 2004. An experiment in inspecting the quality of use case descriptions. *Journal of Research and Practice in Information Technology*. 36, 4 211-229.

[16] Anda, B. and Sjøberg, D. I. 2002. Towards an inspection technique for use case models. In *Proceedings of the Proceedings of the 14th international conference on Software engineering and knowledge engineering* ACM, 127-134.

[17] IEC 2001. IEC 61882: 2001: Hazard and operability studies (HAZOP studies). Application guide. In *Proceedings of the British Standards Institute*

[18] Gomaa, H. 2000. *Designing concurrent, distributed, and real-time applications with UML*. Addison-Wesley.

[19] Yue, T., Briand, L. C., and Labiche, Y. 2015. aToucan: An Automated Framework to Derive UML Analysis Models from Use Case Models. *ACM Transactions on Software Engineering and Methodology (TOSEM)*. 24, 3 13.

[20] Yue, T., Ali, S., and Zhang, M. 2015. RTCM: a natural language based, automated, and practical test case generation framework. In *Proceedings of the Proceedings of the 2015 International Symposium on Software Testing and Analysis* ACM, 397-408.

[21] Zhang, M., Yue, T., Ali, S., Zhang, H., and Wu, J. 2014. A Systematic Approach to Automatically Derive Test Cases from Use Cases Specified in Restricted Natural Languages. In Eds. Springer, 142-157.

[22] DeMillo, R. A., Lipton, R. J., and Sayward, F. G. 1978. Hints on test data selection: Help for the practicing programmer. *Computer*. 4 34-41.

[23] Shahriar, H. and Zulkernine, M. 2008. Mutation-based testing of buffer overflow vulnerabilities. In *Proceedings of the Computer Software and Applications, 2008. COMPSAC'08. 32nd Annual IEEE International* IEEE, 979-984.

[24] Chevalley, P. and Thévenod-Fosse, P. 2003. A mutation analysis tool for Java programs. *International journal on software tools for technology transfer*. 5, 1 90-103.

[25] Offutt, A. J., Voas, J., and Payne, J. 1996. *Mutation operators for Ada.* Techniqual Report.

[26] Fabbri, S. C., Delamaro, M. E., Maldonado, J. C., and Masiero, P. C. 1994. Mutation analysis testing for finite state machines. In *Proceedings of the Software Reliability Engineering, 1994. Proceedings., 5th International Symposium on* IEEE, 220-229.

[27] Phalp, K. T., Vincent, J., and Cox, K. 2007. Assessing the quality of use case descriptions. *Software Quality Journal*. 15, 1 69-97.

[28] Denger, C., Paech, B., and Freimut, B. 2005. Achieving high quality of use-case-based requirements. *Informatik-Forschung und Entwicklung*. 20, 1-2 11-23.

[29] Kletz, T. A. 1999. *HAZOP and HAZAN: identifying and assessing process industry hazards*. IChemE.

[30] ISO 1996. ISO/IEC 14977: 1996 (e), information technology syntactic metalanguage extended bnf. *International Organization for Standardization*.

[31] Offutt, A. J. and Untch, R. H. 2001. Mutation 2000: Uniting the orthogonal. In Eds. Springer, 34-44.

[32] Just, R. and Schweiggert, F. 2015. Higher accuracy and lower run time: efficient mutation analysis using non - redundant mutation operators. *Software Testing, Verification and Reliability*. 25, 5-7 490-507.

[33] Zhang, H., Yue, T., Ali, S., and Liu, C. 2015. Facilitating Requirements Inspection with Search-Based Selection of Diverse Use Case Scenarios. In *Proceedings of the 9th EAI International Conference on Bio-inspired Information and Communications Technologies (BICT)* (Columbia University, NY, USA). ACM, (In Press).

[34] Capozucca, A., Cheng, B., Georg, G., Guelfi, N., Istoan, P., Mussbacher, G., *et al.* 2011. Requirements Definition Document For A Software Product Line Of Car Crash Management Systems. *ReMoDD repository, at* http://www. *cs. colostate. edu/remodd/v1/content/bcms-requirements-definition*.