Empirically Driven Use Case Metamodel Evolution*

Amador Durán[1], Beatriz Bernárdez[1], Marcela Genero[2], and Mario Piattini[2]

[1] University of Seville
{amador,beat}@lsi.us.es
[2] University of Castilla–La Mancha
{marcela.genero,mario.piattini}@uclm.es

Abstract. Metamodel evolution is rarely driven by empirical evidences of metamodel drawbacks. In this paper, the evolution of the use case metamodel used by the publicly available requirements management tool REM is presented. This evolution has been driven by the analysis of empirical data obtained during the assessment of several metrics–based verification heuristics for use cases developed by some of the authors and previously presented in other international fora. The empirical analysis has made evident that some common defects found in use cases developed by software engineering students were caused not only by their lack of experience but also by the expressive limitations imposed by the underlying use case metamodel used in REM. Once these limitations were clearly identified, a number of evolutionary changes were proposed to the REM use case metamodel in order to increase use case quality, i.e. to avoid those situations in which the metamodel were the cause of defects in use case specifications.

Keywords: metamodel evolution, use cases, empirical software engineering

1 Introduction

Metamodel evolution is usually based on a previous theoretical analysis. The usual *evolution vectors* are elimination of internal contradictions, simplification of unnecessary complexities, or enhancement of expressiveness in order to model unforeseen or new concepts [1, 2]. In this paper, the evolution of the use case metamodel implemented in the REM requirements management tool [3] is described. This evolution has been driven not by a theoretical analysis but by the analysis of empirical data obtained during the assessment of several metrics–based verification heuristics for use cases developed by some of the authors (for a description of the verification heuristics and their implementation in REM using XSLT, see [4]; for their empirical assessment and review, see [5]). This empirical analysis has revealed that some common defects in use cases developed by software engineering students had their roots in the underlying REM metamodel, therefore making its evolution necessary in order to increase requirements quality.

The rest of the paper is organized as follows. In the next section, the initial REM use case metamodel is described. The metrics–based verification heuristics that originated the metamodel evolution are briefly described in section 3. In section 4, the results of the

* This work is partially funded by the following projects: AgilWeb (TIC 2003–02737), Tamansi (PCB–02–001) and MESSENGER (PCC–03–003–1).

empirical analysis in which the problems in the metamodel were detected are presented. The proposed changes to the metamodel and their analysis are described in section 5. In section 6, some related work is commented and, finally, some conclusions and future work are presented in section 7.

2 Initial Use Case Metamodel

The initial use case metamodel, i.e. the REM metamodel [4], is shown in Fig. 1. This metamodel was designed after a thorough analysis of several proposals for natural language use case templates like [6, 7, 8, 9]. One of the goals in mind when this initial metamodel was developed was to keep use case structure as simple as possible, but including most usual elements proposed by other authors like conditional steps or exceptions.

Apart from inherited requirements attributes, a use case in REM is basically composed of a *triggering event*, a *precondition*, a *postcondition*, and a *ordinary sequence* of steps describing interactions leading to a successful end. Steps are composed of one *action* and may have a condition (see Fig. 1). Three classes of actions are considered: *system actions* performed by the system, *actor actions* performed by one actor, and *use case actions* in which another use case is performed, i.e. UML *inclusions* or *extensions*, depending on whether the step is conditional or not [10]. Steps may also have attached *exceptions*, which are composed of an exception condition (modeled by the *description* attribute), an action (of the same class than step actions) describing the exception treatment, and a *termination* attribute indicating whether the use case is resumed or canceled after the performance of the indicated action.

Another metamodel goal was to make XML encoding simple so the application of XSLT stylesheets were as efficient as possible. In REM, XML data corresponding to requirements is transformed into HTML by applying a configurable XSLT stylesheet, thus providing a WYSIWYG environment for requirements management (see Fig. 2).

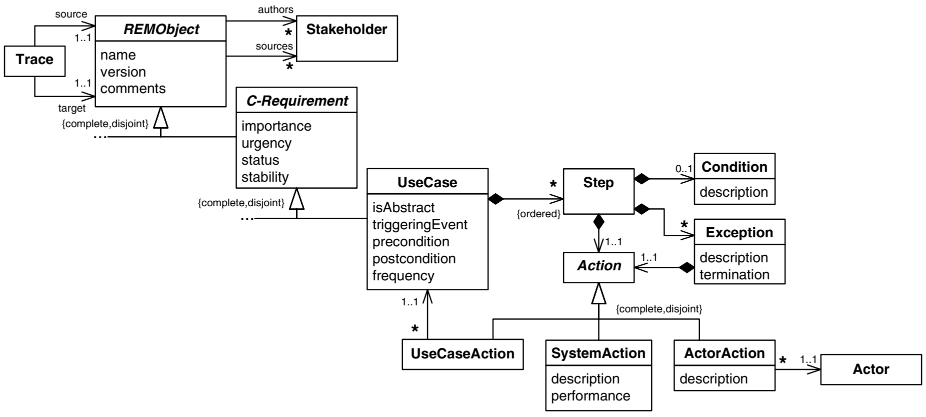
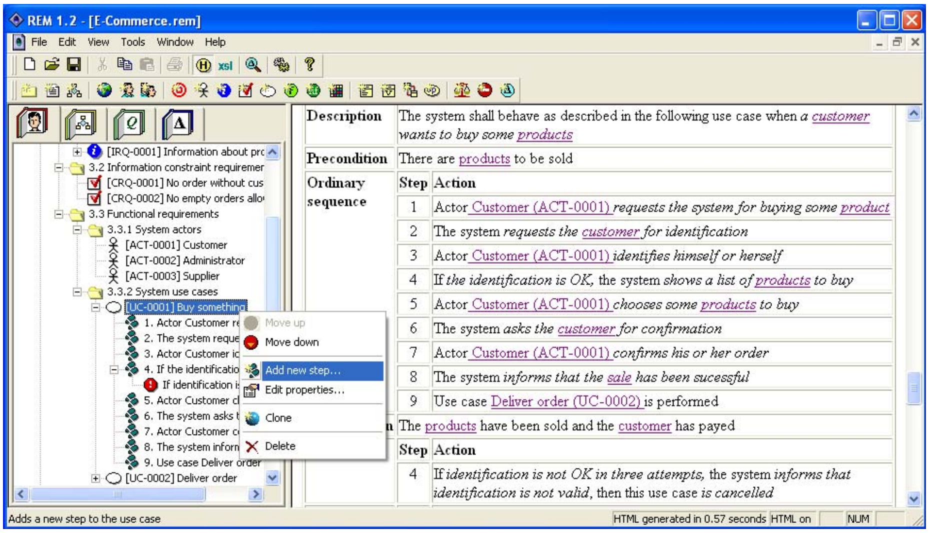


Fig. 1. Initial REM use case metamodel



Description	The system shall behave as described in the following use case when *a customer wants to buy some products*	
Precondition	There are products to be sold	
Ordinary sequence	**Step**	**Action**
	1	Actor Customer (ACT-0001) *requests the system for buying some product*
	2	The system *requests the customer for identification*
	3	Actor Customer (ACT-0001) *identifies himself or herself*
	4	If *the identification is OK,* the system *shows a list of products to buy*
	5	Actor Customer (ACT-0001) *chooses some products to buy*
	6	The system *asks the customer for confirmation*
	7	Actor Customer (ACT-0001) *confirms his or her order*
	8	The system *informs that the sale has been sucessful*
	9	Use case Deliver order (UC-0002) is performed
n	The products have been sold and the customer has payed	
	Step	**Action**
	4	If *identification is not OK in three attempts,* the system *informs that identification is not valid*, then this use case *is cancelled*

Fig. 2. REM user interface for use cases

In this way, all composition relationships (*black diamonds*) in Fig. 1 are easily mapped into XML hierarchies.

3 Verification Heuristics

As described in [4], the experience of some of the authors in the verification of use cases developed by students using REM led to the definition of several metrics–based, defect

```
context UseCase def:
  -- helper definition
  let NOS_TYPE( t:OclType ) : Sequence( Action ) =
        step->select( action.oclIsTypeOf( t ) )

  -- metrics definition
  let NOS  = step->size()                          -- number of steps
  let NOAS = NOS_TYPE( ActorAction    )->size() -- number of actor steps
  let NOSS = NOS_TYPE( SystemAction   )->size() -- number of system steps
  let NOUS = NOS_TYPE( UseCaseAction  )->size() -- number of use case steps
  let NOCS = (step.condition)->size()            -- number of conditional steps
  let NOE  = (step.exception)->size()            -- number of exceptions
  let NOAS_RATE = NOAS / NOS                     -- actor steps rate
  let NOSS_RATE = NOSS / NOS                     -- system steps rate
  let NOUS_RATE = NOUS / NOS                     -- use case steps rate
  let NOCS_RATE = NOCS / NOS                     -- conditional steps rate
  let NOE_RATE  = NOE  / NOS                     -- exceptions rate
  let CC        = NOCS + NOE + 1                 -- cyclomatic complexity

  -- some metrics relationships
  inv: NOAS + NOSS + NOUS = NOS
  inv: NOAS_RATE + NOSS_RATE + NOUS_RATE = 1
```

Fig. 3. Use case metrics definition in OCL

detection heuristics. These heuristics are based on a simple idea: there are some use case metrics for which a range of *usual* values can be defined; if for a given use case, its metric value is out of its corresponding usual range, then the use case is considered as potentially defective and it should therefore be checked.

In Fig. 3, the OCL definition of the metrics used in the verification heuristics, based on the metamodel in Fig. 1, is shown. As described in [4], these metrics can be easily computed in XSLT and the heuristics can be applied automatically in REM using the XML representation of use cases.

The metrics–based heuristics that led to metamodel evolution and their rationales are briefly described below. The usual ranges were chosen after a statistical analysis of 414 use cases developed by students using REM. For a comprehensive discussion and other metrics–based verification heuristics see [4].

Heuristic (A): NOS should be in $[3, 9]$.
Rationale: A use case with just a few steps is likely to be incomplete. Too many steps usually indicate too low level of detail and make the use case too complex to be understood and defect–prone.

Heuristic (B): NOAS_RATE should be in $[30\%, 70\%]$.
Heuristic (C): NOSS_RATE should be in $[40\%, 80\%]$.
Rationale: A use case describes system–actor interactions, so the rate of actor and system steps should be around 50%.

Heuristic (D): NOUS_RATE should be in $[0\%, 25\%]$.
Rationale: An abusive use of use case relationships makes use cases difficult to understand — customers and users are not familiar with procedure call semantics. Use them to avoid repetition of common steps only.

Heuristic (E): CC should be in $[1, 4]$.
Rationale: A high value of the cyclomatic complexity implies many conditional steps and exceptions, probably making the use case too complex to be understood and defect–prone.

4 Empirical Analysis

The empirical assessment and analysis of the verification heuristics introduced in previous section was carried out by manually verifying 127 use cases in 8 requirements documents developed by students of Software Engineering at the University of Seville. The whole process and detailed results are described in [5]. In this section, only most relevant results are presented, especially those related to the REM use case metamodel evolution.

4.1 Analysis of Heuristic A

This heuristic was widely validated by empirical data: 85% of the use cases out of the usual range of the NOS metric were identified as defective. A subsequent analysis of the detected defects revealed that whereas those use cases with too low NOS were usually either incomplete or trivial or described no interaction at all, use cases with too high NOS were usually at a too low level of detail. On the other hand, for the most part of the 15% of the use cases that were wrongly identified as potentially defective, NOS was high because of the *writing style* or because of the presence of *actor–to–actor* interactions.

Writing Style The writing style has been identified as a very important factor for the accuracy of heuristic A. Whereas some students used only one step for specifying a sequence of consecutive actions carried out either by the system or by a given actor, others used one step for each action, thus increasing NOS (see [7] for a comparison of both styles). This heuristic was designed with the former writing style in mind, but as commented in [5], a further analysis for identifying another usual range for the latter writing style is currently being carried out.

Actor–to–Actor Interactions The inclusion of actor–to–actor interactions cannot be in anyway considered as a defect in use cases, but it dramatically affects the accuracy of heuristic A by increasing NOS without making use cases defective (heuristics B and C are also affected, see below). This is one of the metamodel evolution factors that will be taken into consideration in section 5.

4.2 Analysis of Heuristics B and C

Heuristics B and C were also confirmed by empirical data with 80% and 70% respectively of defective use cases out of usual ranges. Both heuristics are tightly coupled because a high value of one of them implies a low value of the other. Use cases with high NOAS_RATE (and therefore low NOSS_RATE) are usually use cases in which system behavior has been omitted or in which a lot of actor–to–actor interactions have been considered (usually *business use cases*), as commented above. On the other hand, use cases with high NOSS_RATE (and therefore low NOAS_RATE) are usually defective use cases describing *batch* processes or internal system actions only.

4.3 Analysis of Heuristic D

Heuristic D, confirmed with 75% of use cases out of usual range being defective, usually detects use cases with a high number of extensions due to a *menu–like* structure, i.e. use cases without a clear goal in which, depending on an actor choice, a number of different use cases are performed. Nevertheless, most of the 25% of use cases out of usual range but presenting no defects were use cases in which the impossibility of the metamodel of representing *conditional blocks* of steps, i.e. a group of steps with the same condition, forced students to create extending use cases in order to avoid the repetition of the same condition along several consecutive steps. The same happened when the treatment of an exceptional situation required more than one single action to be performed (the metamodel in Fig. 1 only allows one action to be associated to an exception). In this case, students were also forced to create an extending use case that was performed when the exception occurred. An example of this situation using a different use case metamodel (Leite's metamodel for scenarios [11]) can be also seen in [12].

4.4 Analysis of Heuristic E

This heuristic was confirmed by empirical data with 87% of use cases out of usual range being defective. The usual cause of defect was the abusive use of conditional steps of the form "*if ¡condition¿, the system goes to step X*", making use cases almost impossible to understand. As commented above for heuristic D, the lack of conditional blocks was the usual case for abnormally high values of CC in non–defective use cases when students decided not to create an extending use case but repeating the same condition along several steps, thus artificially increasing CC value.

5 Metamodel Evolution

Taking into consideration the analysis of empirical data presented in previous section, three main evolution vectors were identified: one for evolving the metamodel in order to be able to represent conditional branches, another for allowing complex treatments of exceptions, and another for introducing new specializations of actions.

5.1 Conditional Branches

The analysis of empirical data has made evident the inadequacy of use case metamodels in which alternative branches consisting of more than one step can be represented only by means of extension relationships or by repeating the same condition along several steps. The analysis has detected that students have overcame this problem either creating an excessive number of *abstract use cases* [13] which extend one use case only (i. e. *singleton abstract use cases*), or repeating the same condition along several consecutive steps. Both situations are clearly undesirable and must therefore be avoided by enhancing the underlying use case metamodel.

In order to allow conditional branches of more than one step, two tentative evolved metamodels were initially proposed (see Fig. 4). The main difference between them is that whereas the former allows the nesting of conditional branches, the latter does not. Keeping the initial goal of having a simple use case structure, and taking into consideration that non–software professionals find nested conditional structures difficult to understand, we decided to allow only one level of conditional branches in the ordinary sequence (a second level is introduced by exceptions, see section 5.2).

A question raised during the discussion of the metamodel evolution was whether conditional branches should have an *else* branch or not. Once again, the goal of keeping use case structure simple made us discard the *if–then–else* structure, which is not often familiar for customers and users. If an *else* branch is necessary, another conditional branch with the explicit negation of the *if* condition can be added to the ordinary sequence of the use case.

Another issue considered during metamodel evolution was the termination of conditional branches. Considering the excellent study of alternatives in use cases presented in [2], we found necessary to specify if a conditional branch: *(a)* resumes the ordinary sequence after performing its last step; *(b)* leads to a goal success termination of the use case; or *(c)* leads to a goal failure termination. This information is modeled as the enumerated attribute termination in class ConditionalBranch. See Fig. 5 for the explicit visual representation adopted for the three different situations in REM.

Thus, the second metamodel in Fig. 4 was finally adopted for the next version of REM. Notice that in both models composition associations can be easily mapped into XML hierarchies, thus keeping the original goal of having a direct mapping into XML.

5.2 Exception Treatment

The analysis of heuristic D pinpointed the need of allowing exceptions to have more than one associated action so *singleton abstract use cases* could be avoided. In order to do so, the composition relationship between exceptions and actions has increased its maximum cardinality from 1 to many in the evolved metamodel (see Fig. 4).

Although conditional branches and exceptions have a lot in common in the evolved metamodel, we decided to keep them as different classes because no empirical evidence pinpointed this as a problem. Notice that keeping exceptions as part of steps, instead of considering as conditional branches, introduces a second nesting level, i.e. a step in a conditional branch can have an attached exception. This situation, which was already present in previous metamodel, presented no empirical evidence of being a source of problems, so we considered there was no need for any change.

See Fig. 6 for the new visual representation of exceptions, in which actions are represented by the same icon than steps for the sake of usability, and exception termination (*resuming* or *canceling* the use case) has been made visually explicit coherently with visual representation of conditional branches shown in Fig. 5.

5.3 Specializations of Actions

The metamodel evolution for the inclusion of new specializations of actions (see Fig. 7) might seem not as obviously necessary as the changes described in previous sec-

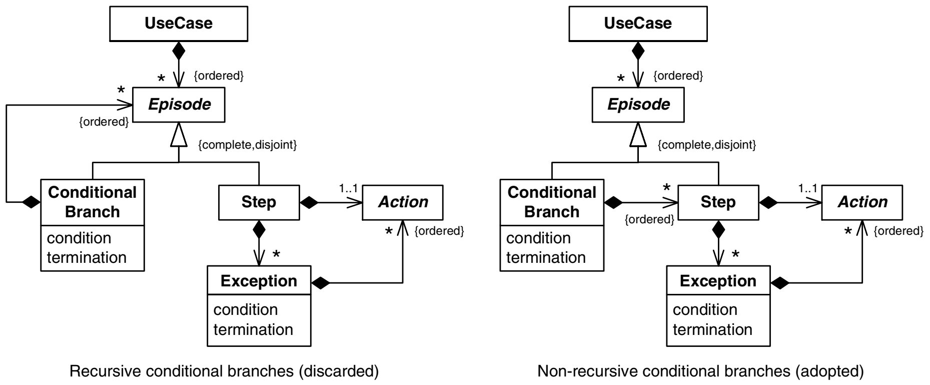


Fig. 4. Tentative evolved metamodels for conditional branches

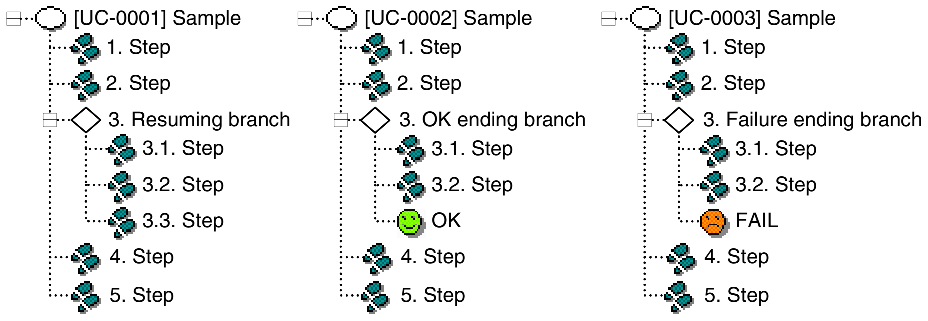


Fig. 5. Visual representation of conditional branches (evolved metamodel)

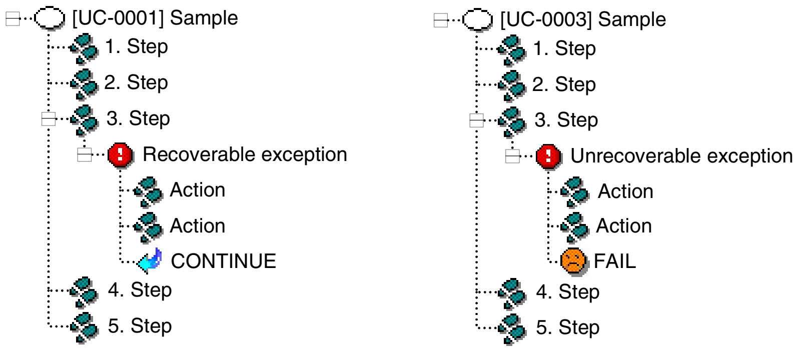


Fig. 6. Visual representation of exceptions (evolved metamodel)

tions. Nevertheless, this evolutive change makes possible the definition of new use case metrics (see Fig. 8) that allow the redefinition of the verification heuristic B in order to increase its accuracy. These new metrics also make possible the definition of new heuristics for defect detection considering different types of actor actions (see section

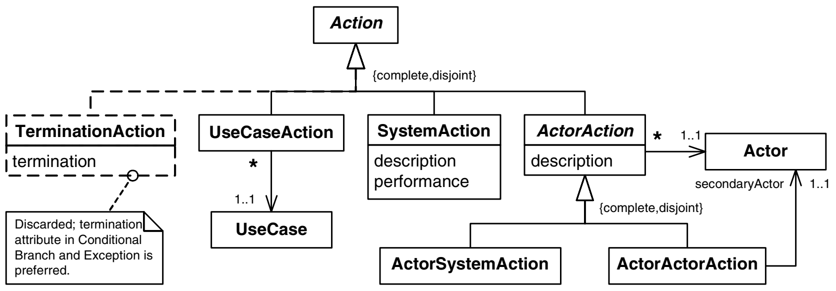


Fig. 7. Evolved metamodel for new specializations of actions

```
context UseCase def:
  -- helper definitions
  let allSteps : Sequence( Step ) =
    episode->iterate( e : Episode; acc : Sequence( Step ) = Sequence{} |
      if   e.oclIsTypeOf( Step ) then acc->including( e.oclAsType( Step ) )
      else acc->union( e.oclAsType( ConditionalBranch ).step )
      endif
    )

  let allBranches : Sequence( ConditionalBranch ) =
    episode->select( e | e.oclIsTypeOf( ConditionalBranch ) )

  let NOS_TYPE( t:OclType ) : Sequence( Action ) =
    allSteps->select( action.oclIsKindOf( t ) )

  -- metrics definition
  let NOS   = allSteps->size()                      -- no. of steps
  let NOAAS = NOS_TYPE( ActorActorAction )->size()  -- no. of actor-actor steps
  let NOASS = NOS_TYPE( ActorSystemAction )->size() -- no. of actor-system steps
  let NOAS  = NOS_TYPE( ActorAction )->size()       -- no. of actor steps
  let NOSS  = NOS_TYPE( SystemAction  )->size()     -- no. of system steps
  let NOUS  = NOS_TYPE( UseCaseAction )->size()     -- no. of use case steps
  let NOCB  = allBranches->size()                   -- no. of cond. branches
  let NOE   = (allStep.exception)->size()           -- no. of exceptions
  let NOAS_RATE  = NOAS / NOS                       -- actor steps rate
  let NOAAS_RATE = NOAAS / NOS                      -- actor-actor steps rate
  let NOASS_RATE = NOASS / NOS                      -- actor-system steps rate
  let NOSS_RATE  = NOSS / NOS                       -- system steps rate
  let NOUS_RATE  = NOUS / NOS                       -- use case steps rate
  let NOCS_RATE  = NOCS / NOS                       -- conditional steps rate
  let NOE_RATE   = NOE  / NOS                       -- exceptions rate
  let CC         = NOCB + NOE + 1                   -- cyclomatic complexity

  -- some metrics relationships
  inv: NOAS = NOAAS + NOASS
  inv: NOAS + NOSS + NOUS = NOS
  inv: NOAS_RATE = NOAAS_RATE + NOASS_RATE
  inv: NOAS_RATE + NOSS_RATE + NOUS_RATE = 1
```

Fig. 8. New use case metrics definition in OCL

7). Moreover, the explicit identification of the secondary actor in actor–to–actor actions adds important information for use cases; in the previous metamodel, the secondary actor in an actor–to–actor actions was *hidden* in the text of the action description.

As shown in Fig. 7 in dashed line, a new kind of action, namely TerminationAction, was temporarily considered for expressing the termination of conditional branches and exceptions. This change was eventually discarded in favour of the termination attribute because this new kind of action made possible the termination of the use case at any point, something that would require adding complex constraints to the metamodel in order to be avoided.

5.4 Evolutive Changes Summary

The evolutive changes in the finally adopted use case metamodel are the following:

1. A new abstract class, Episode, has been introduced for representing both steps and conditional branches in the ordinary sequence of use cases.

2. A new class, ConditionalBranch, has been introduced for representing conditional branches inside ordinary sequence of use cases. Conditional branches are composed of a sequence of steps. Notice that this is a non–recursive composition, i.e. conditional branches cannot contain other conditional branches. Conditional branches also have an enumerated attribute, termination, that can take one of the following values: *resumes, OK_ending, failure_ending*.
3. The cardinality of the composition between Exception and Action classes has changed from 1–to–1 to 1–to–many in order to allow the specification of more than one Action as the exception treatment, thus avoiding the need of creating singleton abstract use cases for such situations.
4. The attribute description of class Exception has been renamed as condition in order to make its semantics more evident.
5. The ActorAction class has been specialized into two new subclasses, ActorSystemAction and ActorActorAction. The latter has an association with the Actor class representing the secondary actor.

6 Related Work

As far as we know, this is the only work on empirically driven metamodel evolution, i.e. metamodel evolution motivated after the analysis of empirical data about the quality of the models that are *instances of* the metamodel. Other works on metamodel evolution and use case metamodels are commented below.

Henderson–Seller's works [1, 14] are an excellent example of a proposal for metamodel evolution driven by a thorough theoretically analysis, although not directly related to use cases. He focuses on metamodel deficiencies in the UML.

In [2], several changes to the UML 1.3 use case metamodel are proposed in order to support several types of alternative courses, namely *alternative stories*, *use case exceptions*, and *alternative parts*. This work has had a strong influence on the evolution of the REM use case metamodel, especially on the evolution of conditional branches, as commented in section 5.1.

In [15], an evolution of the UML 1.3 use case metamodel is proposed in order to support a viewpoint–oriented approach to requirements engineering. In [16], a refactoring–oriented use case metamodel is described, including a rich set of use case relationships.

7 Conclusions and Future Work

In this paper, we have presented an evolution of the REM use case metamodel. Unlike other metamodel evolutions which are driven by theoretical analysis, the evolution presented in this paper has been driven by the analysis of empirical data. We consider that, specially in an informal realm like use cases, empirical evidence should be the main metamodel evolution force, even above theoretical analysis. In our case, the REM use case metamodel has experienced a significant evolution that would have not probably taken place without an empirical approach.

Following our empirical philosophy, the immediate future work is to validate the metamodel evolution, i.e. to check if the evolved metamodel increases the quality of use cases. By the time the next version of the REM tool will be available, we will be able to drive the corresponding empirical studies with our students.

We are also investigating new defect detection heuristics, but for the moment, the lack of use cases developed using the new metamodel allows only speculative approaches.

References

[1] Henderson-Sellers, B.: Some problems with the UML V1.3 metamodel. In: Proc. of 34th Annual Hawaii International Conference on System Sciences (HICSS), IEEE CS Press (2001)

[2] Metz, P., O'Brien, J., Weber, W.: Specifying Use Case Interaction: Types of Alternative Courses. Journal of Object Technology **2** (2003) 111–131

[3] Durán, A.: REM web site. http://rem.lsi.us.es/REM (2004)

[4] Durán, A., Ruiz-Cortés, A., Corchuelo, R., Toro, M.: Supporting Requirements Verification using XSLT. In: Proceedings of the IEEE Joint International Requirements Engineering Conference (RE), Essen, Germany, IEEE CS Press (2002) 141–152

[5] Bernárdez, B., Durán, A., Genero, M.: An Empirical Evaluation and Review of a Metrics–Based Approach for Use Case Verification. Journal of Research and Practice in Information Technology (2004) To be published in a special collection on Requirements Engineering.

[6] Coleman, D.: A Use Case Template: Draft for Discussion. Fusion Newsletter (1998)

[7] Cockburn, A.: Writing Effective Use Cases. Addison–Wesley (2001)

[8] Schneider, G., Winters, J.P.: Applying Use Cases: a Practical Guide. Addison–Wesley (1998)

[9] Durán, A., Bernárdez, B., Ruiz, A., Toro, M.: A Requirements Elicitation Approach Based in Templates and Patterns. In: WER'99 Proceedings, Buenos Aires (1999)

[10] OMG: Unified Modeling Language Specification, v1.5. The Object Management Group, Inc. (2003)

[11] Leite, J.C.S.P., Hadad, H., Doorn, J., Kaplan, G.: A Scenario Construction Process. Requirements Engineering Journal **5** (2000)

[12] Ridao, M., Doorn, J.: Anomaly Modeling with Scenarios (in Spanish). In: Proceedings of the Workshop on Requirements Engineering (WER), Valencia, Spain (2002)

[13] Jacobson, I., Griss, M., Jonsson, P.: Software Reuse: Architecture, Process and Organization for Business Success. Addison–Wesley (1997)

[14] Barbier, F., Henderson-Sellers, B., Le Parc–Lacayrelle, A., Bruel, J.M.: Formalization of the Whole–Part Relationship in the Unified Modeling Language. IEEE Transactions on Software Engineering **29** (2003)

[15] Nakatani, T., Urai, T., Ohmura, S., Tamai, T.: A Requirements Description Metamodel for Use Cases. In: Proc. of 8th Asia–Pacific Software Engineering Conference (APSEC), IEEE CS Press (2003)

[16] Rui, K., Butler, G.: Refactoring Use Case Models: The Metamodel. In: Proc. of 25th Computer Science Conference (ACSC). (2003)