Fixing Generalization Defects in UML use Case Diagrams

Xavier Dolques, Marianne Huchard, Clémentine Nebut, Philippe Reitz

Fixing generalization defects in UML use case diagrams

Xavier Dolques, Marianne Huchard, Clémentine Nebut, and Philippe Reitz

LIRMM, CNRS and Université Montpellier II, Montpellier
{dolques,huchard,nebut,reitz}@lirmm.fr

Abstract. Use case diagrams appear in early steps of a UML-based development. They capture user requirements, structured by the concepts of use cases and actors. An admitted good modeling practice is to design simple, easy-to-read use case diagrams. That can be achieved by introducing relevant generalizations of actors and use cases. The paper proposes an approach based on Formal Concept Analysis and one of its variants, Relational Concept Analysis, to refactor a use case diagram as a whole in order to make it clearer. We developed a tool and studied the results on about twenty examples, concluding on the relevancy of the approach on our benchmark.

Keywords: Formal Concept Analysis, UML use case diagrams

1 Introduction

Within a UML-based development, the very first model to be designed is the use case diagram, that is later used all over the modeling process. For example, in the Rational Unified Process [11], the development process is driven by the use cases: the use cases are continuously used to inject further functionalities in static models, they guide the testing plan, etc. Thus even if the use case diagram is probably the furthest one from implementation, it takes a large place in the design of many artifacts. Use case diagrams model the boundary of a system, the actors that interact with the system, and the use cases (main functionalities offered to actors by the system). Use cases are linked to the actors interacting with them, and use cases may be organized using three types of relation: inclusion, extension (that can be seen as conditional inclusion), and generalization.

As mentioned in the UML user guide [2], "Organizing your use cases by extracting common behavior (through include relationships) and distinguishing variants (through extend relationships) is an important part of creating a simple, balanced, and understandable set of use cases for your system." A good practice for use case diagrams is to make them simple to read [1], so as to catch at first glance the boundaries of the system, its main actors and main functionalities. Thus a tangled use case diagram (*e.g.* with numerous and tangled relations linking use cases to actors and to other use cases) will regularly brake the development since a designer, each time he or she will consult it, will have a delay to understand (again) the use case diagram.

In this paper, we investigate an approach to make use case diagrams clearer by introducing generalized actors and use cases to factorize relations. Our approach is based on several refactoring patterns (*e.g.* when two use cases include the same third one, they can be generalized by a new use case including this third use case), that are globally combined using Formal Concept Analysis (FCA) and one of its variant Relational Concept Analysis (RCA). The approach is implemented in an Eclipse-integrated tool that takes as input a UML diagram, encodes it into FCA (or RCA) contexts, generates the corresponding concept lattices, and finally produces as output the refactored use case diagram. We studied the results of this approach on several examples, comparing the pairs of input diagram/output diagram, and FCA versus RCA.

The following of the paper is structured as follows. We first illustrate in Section 2 the use case diagram refactoring with an example that will later be used to illustrate our approach. Section 3 details the used refactoring patterns, and Section 4 explains how they are globally applied using FCA or RCA. Results of our experiments on examples are given in Section 5. Section 6 compares our approach w.r.t. related work.

2 A motivating example

This section presents a motivating example. We briefly recall the main elements involved in UML use case diagrams, and then illustrate the use case generalization defects. The application we are interested in should allow various users to buy products or apply for intership. Figure 1 shows a use case diagram for this application. Actors represent the role(s) played by external persons or systems

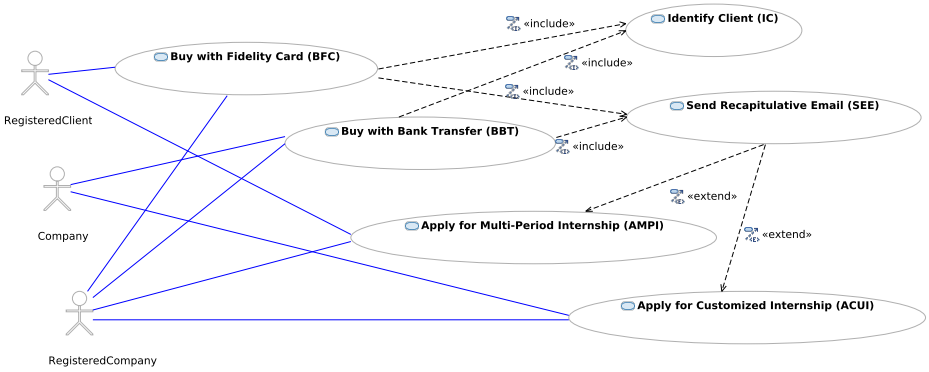


Fig. 1. A use case diagram for the internship example

that interact with the modelled system. They are depicted using a "stick man" icon (or a box stereotyped <<actor>> with the name of the actor). In the example, three roles are identified: a company, a registered client, and a registered

company. Use cases model large functionalities. They are depicted by a named ellipse. In our example, we find for example the use case `Apply for Multi-Period Internship`, modelling the functionality corresponding to the possibility to apply for internships organized over several periods (for example a day per month). Use cases are associated (graphically by a continuous line) to the actors involved in them: actors to which the functionality is dedicated or secondary actors that interact with the system during the achievement of the functionality. In our example, the use case `Apply for Multi-Period Internship` is linked to the `Registered Client` and the `Registered Company` actors, since both of those actors can apply for such internships (whereas non-registered companies cannot). A use case may include another one. Graphically, it is shown by a dashed oriented arrow stereotyped by <<include>>. Semantically, that means that the including use case explicitly incorporates the behaviour of the included use case. For example, the use case `Buy with Bank Transfer` includes the use case `Send Recapitulative Email`. A use case may be extended by another one. Graphically, it is shown by a dashed oriented arrow stereotyped by <<extend>>. Semantically, that means that the extended use case *may* include the behaviour of the extending use case. In our example, a recapitulative email may be sent while realizing an appliance for a multi-period internship. Conditions for the extension and extension points can complete the extension relationship, as discussed later in the paper. UML provides two generalization mechanisms in use case diagrams: for actors and for use cases. Figure 2 shows a refactored use case diagram for the internship example, in which those two mechanisms appear. A

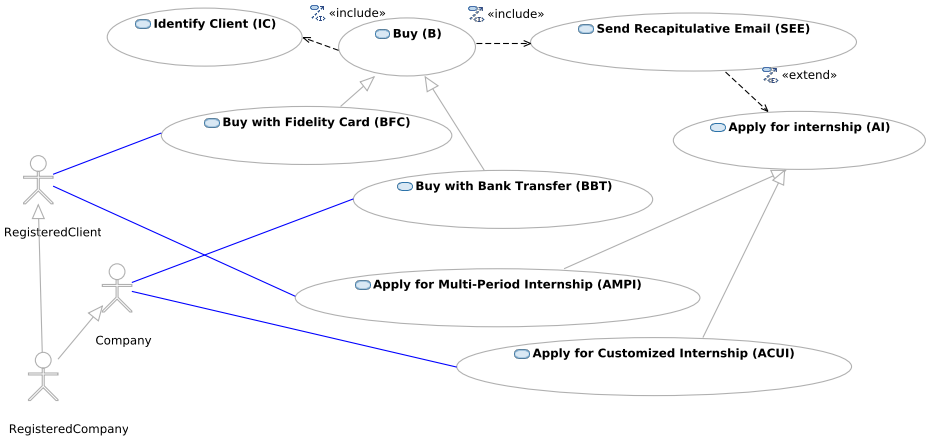


Fig. 2. A refactored use case diagram for the internship application using FCA

use case may generalize another one: the child use case inherits the behaviour of the parent use case, as well as its meaning. It is graphically represented by an edge ended by a triangle, like for class inheritance. In the refactored example of Figure 2, the use case `Buy` generalizes the use cases `Buy with Bank Transfer`

and `Buy with Fidelity Card`. An actor may generalize another one: the child actor inherits the roles of the parent actor, *i.e.* its interactions with use cases. For example in Figure 2, the actor `RegisteredCompany` specializes the actor `Company`.

The initial use case diagram of Figure 1 suffers from the large number of crossed links from actors to use cases, and to a lesser extent, from the large number of links from use cases to use cases. For example, the actor `RegisteredCompany` is linked to four use cases. Introducing a generalization from this actor to the actors `Company` and `RegisteredClient` is semantically correct and factorizes those four links: in the refactored use case diagram proposed in Figure 2, the actor `RegisteredCompany` inherits its links to the use cases from its two parent actors. Similarly, the two inclusions of the use case `Client Identification` (resp. `Send Recapitulative Email`) by the use cases `Buy with Fidelity Card` and `Buy with Bank Transfer` can be factorized introducing the sound use case `Buy`. Last, the two extensions can be factorized introducing the sound `Apply For Internship` use case.

The approach we propose aims at detecting the possible relevant generalizations in a use case diagram, and introduces them in order to obtain a simpler use case diagram. For that, we first identify (in refactoring patterns) situations for which introducing a generalization may be useful. Then to automatically apply these patterns, we use Formal Concept Analysis (FCA) or one of its variants Relational Concept Analysis (RCA). The refactoring patterns are presented in the next section while the FCA/RCA application is detailed in Section 4.

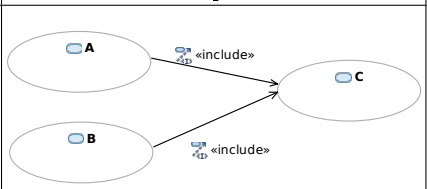
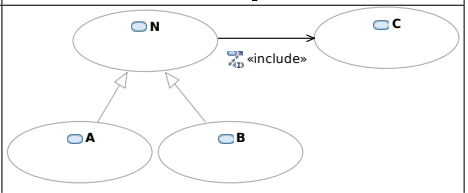
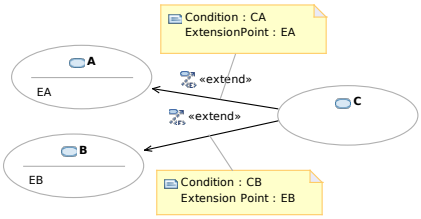
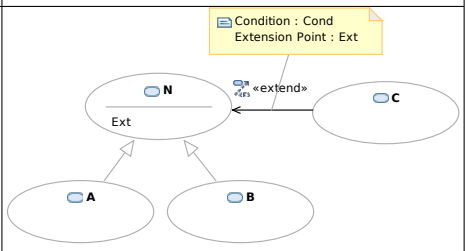
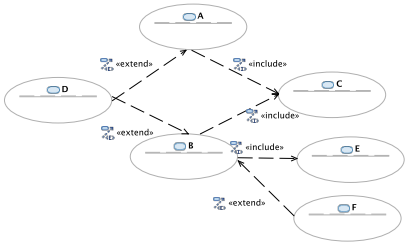
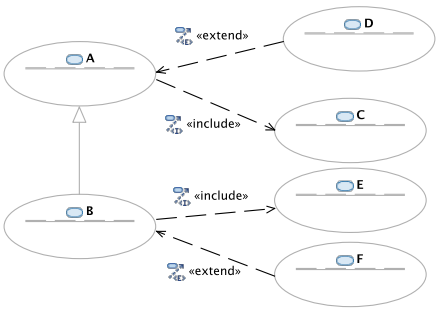
3 Local refactoring patterns

To correct generalization defaults, we propose a systematic approach that introduces more general elements (actors or use cases) based on the detection of shared relations. However, this does not guarantee the relevance of factorizing the relations using more general use cases or actors. The pertinence of refactorings is particularly challenged by the lack of information *e.g.* no role mention is given. In this section, we study several refactoring proposals. We consider here a simple set-based semantics, associating to a use case diagram the set of its actors and for each actor the feasible sequences of leaf use cases. The use cases having specializations are supposed to be abstract, *i.e.* replaced by one of their specializations during a concrete usage of the system.

Refactorings between use cases. Table 1 presents the refactoring patterns for use cases. The pattern I considers the situation of two use cases `A` and `B` including the same use case `C`. In this situation, all the sequences that can be performed by an actor and that contain `A` (resp. `B`) will also contain `C`. We propose to refactor the two inclusions as seen in the refactored pattern: a sequence that can be performed by an actor and containing `A` will still contain `C`, since `A` specializes `N` that contains `C`. In situation II, when the two conditions of extension `CA` and `CB` are identical and that the extension points (that roughly specify when the extension should occur) are also identical, then a generalization can be introduced.

It frequently occurs that extension points and conditions are not specified for the extensions, in this case we propose to apply the refactoring, whereas if the extension conditions or the extension points are different, this refactoring does not apply. The situation III introduces a specialization refactoring. The source pattern owns a use case `A` for which the sets of outgoing `include` relations and incoming `extend` relations are strictly included in those of another use case `B`. The proposed refactoring derives `B` from `A` and removes from `B` the inherited relations. Situations to factorize incoming include relations or outgoing extend relations were studied but rejected since they were not pertinent.

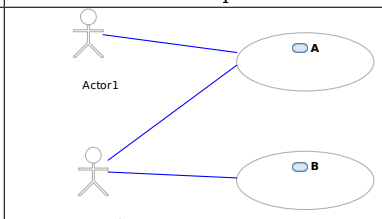
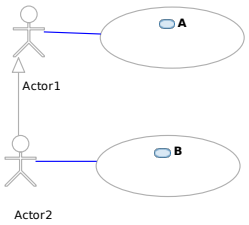
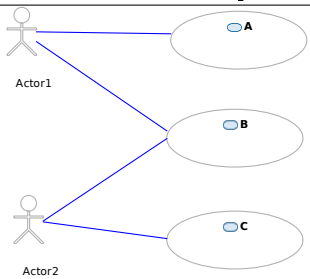
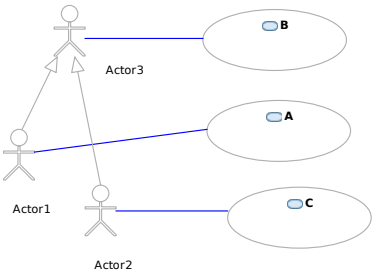
Table 1. Refactoring patterns between use cases

	source pattern	refactored pattern
I	 outgoing include	
II	 incoming extend	
III	 shared extend and include	

Refactorings between actors. Table 2 presents two refactoring patterns between use cases and actors. The first one (I) introduces a specialization relation between actors when the set of associations of one actor is strictly included in the associations of another actor, and then removes the inherited associations.

The second one (II) adds an abstract actor when the sets of associations of two actors have a non-empty intersection but each actor has associations that the other one does not have.

Table 2. Refactoring patterns between actors and use cases

	source pattern	refactored pattern
I	 Actors and use cases with specialization	
II	 Actors and use cases with factoring	

4 Global refactoring through Formal Concept Analysis

The refactoring schemas presented in the previous section do not give a unique solution to deal with a use case diagram as a whole. For that, we use Formal Concept Analysis [7] which systematically groups objects owning common characteristics in a minimal generalization structure.

Refactoring with FCA. We build two formal contexts (Table 3). In one context, we associate an actor with a use case when the actor is involved in the use case. In the second context, use cases are described by inclusion of another use case, extension by another use case and names. Figures 3 and 4 present the lattices. In the lattice of actors (Fig. 3), a generalization relationship is introduced: `C2`, which represents `RegisteredCompany`, is indeed a subconcept of `C1` (`RegisteredClient`) and `C3` (`Company`). This is an application of the first actor-use case refactoring. The lattice of use cases (Fig. 4) highlights two refactorings. The outgoing-include refactoring appears on the left: `C5` generalizes the concepts that introduce `Buy with Fidelity Card` (`C8`) and `Buy with Bank Transfer` (`C11`) because these two use cases include `Send Recapitulative Email` and

`Identify Client`. `C7` shows an opportunity to apply the incoming extend refactoring (II): it factorizes the characteristic of being extended by `Send Recapitulative Email`, which is shared by `Apply for Multi-Period Internship` (`C10`) and `Apply for Customized Internship` (`C12`). The final use case diagram (Fig. 2) is deduced from the two lattices. The (non-trivial) concepts are interpreted as actors or use cases and the lattice partial order as generalization/specialization as illustrated just before. In our example, no new actor is created, but it hap-

Table 3. Contexts for FCA refactoring (use case names have been shorten)

	IsInvolvedIn			
	BFC	AMPI	BBT	ACUI
RegisteredClient	×	×		
Company			×	×
RegisteredCompany	×	×	×	×

	Includes		IsExtendedBy	Name					
	SEE	IC	SEE	BFC	SEE	AMPI	BBT	ACUI	IC
BFC	×	×		×					
SEE					×				
AMPI			×			×			
BBT	×	×					×		
ACUI			×					×	
IC									×

pens in other diagrams of our benchmark. Two new use case factorizations are introduced: `Buy` (from `C5`) and `Apply for internship` (from `C7`) that appear in the final diagram and increase its abstraction level and reusability.

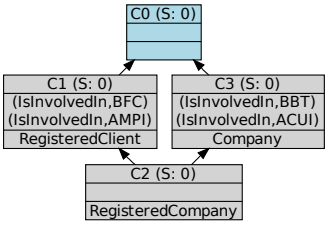


Fig. 3. The lattice of actors built using FCA

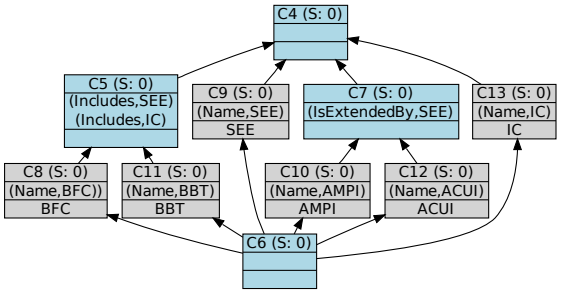


Fig. 4. The lattice of use cases built using FCA

Refining the refactoring with RCA For further refining of the result, we applied Relational Concept Analysis (RCA) [9], one of the extensions of Formal Concept Analysis that takes into account relationships between formal objects in the classification process. The input of Relational Concept Analysis is a set of tables (a relational context family or RCF) such that some tables represent object-attribute relations and some tables capture object-object relations. In the encoding we choose (Table 4), the non-relational part of the RCF is composed of two object-attribute tables: actors with no description (empty attribute set), and use cases described by their names. The relational part of the RCF is composed of three object-object tables: `Includes` relation, `IsExtendedBy` relation

and `IsInvolvedIn` relation. The choice of these relations is guided by the refactoring patterns that we have identified as relevant, and by preliminary experiments that highlighted cases where combinatorial explosion occurred.

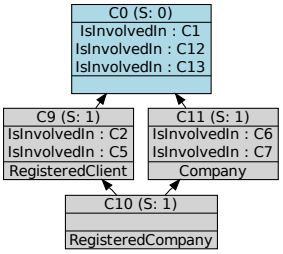


Fig. 5. The lattice of actors built using RCA

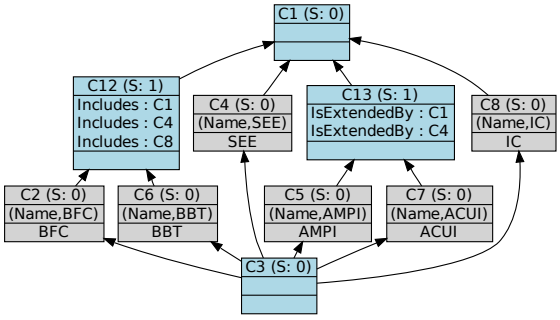


Fig. 6. The lattice of use cases built using RCA

One lattice is built for each object-attribute table, in our case a lattice for actors, and a lattice for use cases. Those two lattices integrate the attributes and the relations. RCA iterates on two successive steps: building of lattices with classical FCA framework (on each object-attribute table concatenated with the corresponding object-object tables) and integration of the concepts discovered at the current iteration in the relational part (to be used at the next iteration). The integration of the concepts discovered at the current iteration uses scaling operators, here the existential operator has been used. For the integration, an object-object relation (*e.g.* `Includes`) is transformed by replacing the objects which are in the columns by the concepts of the lattice built at the current iteration on these objects. Then a relation is established in the new, existentially scaled, table, between an object and a concept, when the object is in relation with at least one object in the extent of the concept. For example, `RegisteredClient` `IsInvolvedIn` `BFC` in the original relation. Then, when at an iteration `BFC` is in the extent of a concept, say `C12`, in the scaled `IsInvolvedIn` relation, `RegisteredClient` is considered as participating in one of the use cases grouped by `C12`. A pair (`RegisteredClient`, `C12`) is added in the scaled table `IsInvolvedIn`. This pair enriches the description of the three actors and at the next FCA step, it is factored out in `C0`, top of the actor lattice. That highlights the opportunity to define a more generalized actor, this opportunity did not appear in the one-step FCA building.

At each step, new concepts can appear, as well as new pairs relating these new concepts. The process iterates until no new concept or description emerge. Figures 5 and 6 show the final lattices. Lattices are interpreted as in the FCA analysis, adding an interpretation of the references between the two lattices. The result of RCA improves the result of FCA on this example by adding the new,

Table 4. Relational Context Family for RCA refactoring

use case	Name BFC	SEE	AMPI	BBT	ACUI	IC
BFC	×					
SEE		×				
AMPI			×			
BBT				×		
ACUI					×	
IC						×

Includes	BFC	SEE	AMPI	BBT	ACUI	IC
BFC		×				×
SEE						
AMPI						
BBT		×				×
ACUI						
IC						

IsExtendedBy	BFC	SEE	AMPI	BBT	ACUI	IC
BFC						
SEE						
AMPI		×				
BBT						
ACUI		×				
IC						

IsInvolvedIn	BFC	SEE	AMPI	BBT	ACUI	IC
RegisteredClient	×		×			
Company				×	×	
RegisteredCompany	×		×	×	×	

actor
RegisteredClient
Company
RegisteredCompany

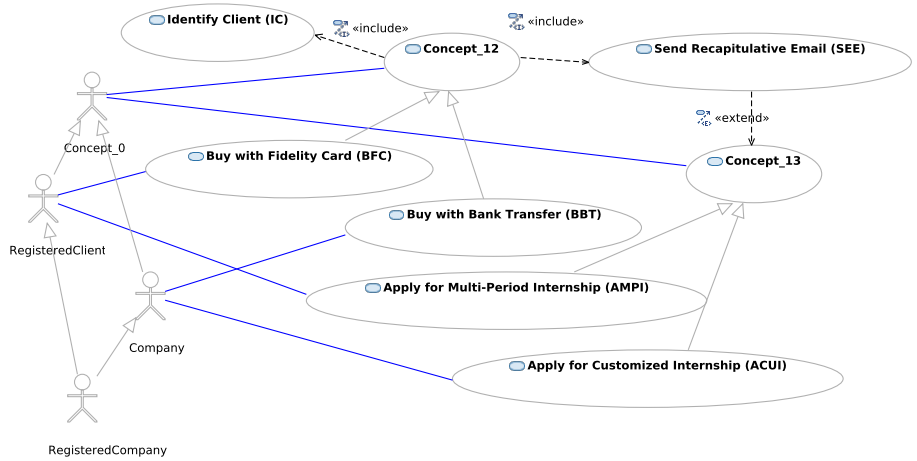


Fig. 7. The internship example refactored using RCA

5 Case Study

We present in this section the current implementation of our approach and the results obtained by running it on different kinds of data.

The Eclipse Modelling Framework (EMF) is a facility from the Eclipse IDE to implement modelling languages and generate tools to manipulate instances of those languages from programs. We use in our tool two Eclipse plugins based on

EMF: UML2[1] and eRCA[2]. UML2 is an implementation of UML and therefore allows us to load, modify and save use case diagrams. eRCA is an implementation of FCA and RCA algorithms which uses EMF to implement contexts and lattices. We have developed a tool implementing both the FCA and RCA approaches for use case refactoring. For both of them we developed the same three steps described in Section 4: first the tool loads a use case diagram using UML2 plugin and encodes it as contexts in eRCA format; second it uses the eRCA tool to generate lattices in eRCA format; finally lattices are automatically analyzed to create back a use case diagram in the UML2 plugin format and to save it.

We present here the results obtained using data gathered from different sources such as student projects or UML courses taken from the internet[3]. We encoded these diagrams using the UML2 plugin in Eclipse. Then we applied our two processes in all of them and automatically processed the results to compare them with the original diagrams. We computed the density of the diagram which is, if we consider the diagram as a graph $G = (V, E)$, the number of edges over the number of possible edges $|E|/|V|^2$. We assume that the lower density we have the clearer the diagram is. Of course, this is just indicative. We present in Figure 8 how the density can be changed by our approach on all our data. The last five diagrams considered (`test1` to `test5`) are used to test specific situations. We see that in most of the diagrams the density is lowered by using FCA and RCA, but with better results for FCA. Density goes down due to the creation of new use cases or actors and to the factorization of the relationships. An inverse situation can occur if the number of generalization links created is higher than the number of links removed by factorization. This is why FCA seems to be better here than RCA, because RCA creates more concepts. Nevertheless, the density for RCA usually remains better than the original density. We also see that density is not improved for the diagram `Invoice`. Indeed, we assume that each actor involved in a use case has the same role, so they can be factored into the same concept. But in the case of this diagram, many actors are involved in the same use cases but with different roles. This may be allowed depending on the interpretation of the UML specification, but results in the merging of many actors into one. By removing those actors and thus not adding new abstractions among actors, the density is then increased. This can be partly solved by adding an identifier for each actor when creating the contexts, but this prevents us to find new generalizations of actors since none of them would share in their intent the same identifiers.

6 Related Work

Literature dealing with designing use cases is prolific, but usually concentrates on textual use cases [13, 3], and not on UML use case diagrams. It is generally admitted that use cases should be described by quite a lot of information that

[1] http://www.eclipse.org/uml2/
[2] http://code.google.com/p/erca/
[3] all data available at http://www.lirmm.fr/%7edolques/publications/data/cla10

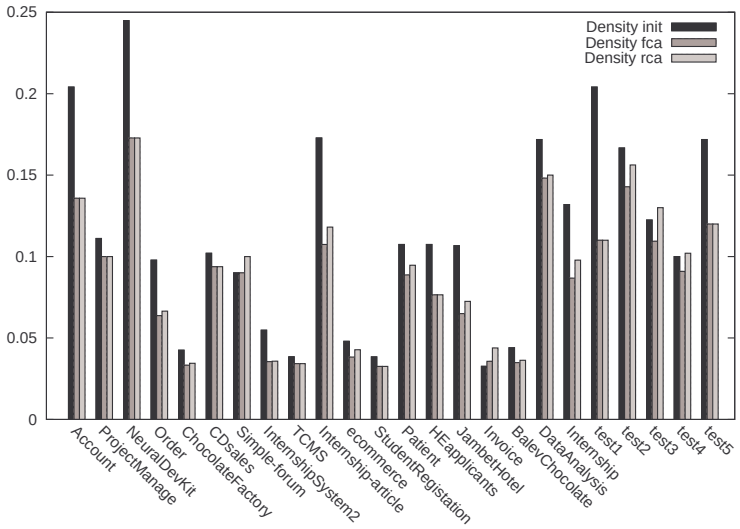


Fig. 8. Experimental results on the evolution of the density

does not appear in use case diagrams, such as the nominal and exceptional flows of events or scenarios corresponding to a use case, and many research activities studied how to structure each use case, in particular with dedicated templates [3] or natural language techniques [6]. Fewer work [14, 10] deal with the way to structure the relations linking uses cases to actors, or use cases to other use cases. In [14] a metamodel is proposed for use cases, extending and grouping existing metamodels, and detailing relations between use cases: composition, dependency, precedence, extension, generalization, etc. Our work focuses on the relations introduced in the use case part of the UML specification, however it should be interesting to investigate how use case models conform to the metamodel of [14] could be refactored. Issa [10] proposes refactoring the scenario part of a use case, for example by separating a use case into several use cases. Henderson-Sellers et al. [8] introduce metrics to compare various versions of a use case, mainly consisting of graph metrics.

FCA has already been applied to use cases [12, 5] but not with the objective to refactor use case diagrams. In [12] use cases written in a controlled natural language are classified into a lattice using the terms contained in the use cases, with a visualization objective. In [5] use cases are classified using their important terms, in order to detect main notions that can become classes. A preliminary version of this work can be found in the french-written reference [4]. The new contributions of this paper consist in the comparison of results obtained with RCA and FCA (while reference [4] focuses only on FCA), in the case study, and in a new tool integrating RCA and FCA.

7 Conclusion

UML use case diagrams can be seen as the functional cornerstone of UML modeling. In this paper, we proposed an approach to refactor such use case diagrams in order to make them clearer. The principle is to use FCA or RCA in order to detect new abstractions that, when introduced in the diagram, can reduce its visual complexity. Using the tool we developed to implement this approach, we carried on experiments using a set of about 20 use case diagrams to measure its efficiency in terms of density of relations in the diagram. It results that both FCA and RCA processes decrease the density of realistic-size use case diagrams, the density obtained with FCA being lower than with RCA. However, RCA discovers more abstract use cases or actors that can be relevant. To go further in this study, we are studying accurately other metrics, both quantitative as average degree, and qualitative, as precision.

Acknowledgments. Authors would like to thank L. M. Hakik for a preliminary study, K. Bouzroud, I. Dagha, H. El Assam, M. El Asri, and J. Ruizsimari for their help in the implementation of the current tool, and the anonymous reviewers for their suggestions on evaluation metrics.

References

1. Ambler, S.W.: The Object Primer: Agile Model-Driven Development with UML 2.0. Cambridge University Press, New York, NY, USA (2004)
2. Booch, G., Rumbaugh, J., Jacobson, I.: Unified Modeling Language User Guide, chap. 16. Addison-Wesley Prof. (2005)
3. Cockburn, A.: Writing Effective Use Cases. Addison-Wesley Professional (2000)
4. Dolques, X., Madiha Hakik, L., Huchard, M., Nebut, C., Reitz, P.: Correction des défauts de généralisation dans les diagrammes de cas d'utilisation UML. In: Proc. of LMO'10 (Langages et Modèles à Objets). pp. 51–66 (2010)
5. Düwel, S., Hesse, W.: Bridging the gap between use case analysis and class structure design by formal concept analysis. In: Proceedings of Modellierung 2000. pp. 27–40. Fölbach-Verlag (2000)
6. Fantechi, A., Gnesi, S., Lami, G., Maccari, A.: Application of linguistic techniques for use case analysis. In: IEEE International Conference on Requirements Engineering. p. 157. IEEE Computer Society, Los Alamitos, CA, USA (2002)
7. Ganter, B., Wille, R.: Formal Concept Analysis: Mathematical Foundations. Springer (1999)
8. Henderson-Sellers, B., Zowghi, D., Klemola, T., Parasuram, S.: Sizing use cases: How to create a standard metrical approach. In: OOIS. Lecture Notes in Computer Science, vol. 2425, pp. 409–421. Springer (2002)
9. Huchard, M., Hacene, M.R., Roume, C., Valtchev, P.: Relational concept discovery in structured datasets. Ann. Math. Artif. Intell. 49(1-4), 39–76 (2007)
10. Issa, A.: Utilising refactoring to restructure use-case models. In: Proc. of the World Congress on Engineering, 2007. pp. 523–527. LNCS (2007)
11. Kruchten, P.: The Rational Unified Process: An Introduction, Second Edition. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (2000)

12. Richards, D., Böttger, K., Aguilera, O.: A controlled language to assist conversion of use case descriptions into concept lattices. In: Australian Joint Conference on Artificial Intelligence. pp. 1–11. LNCS, Springer (2002)
13. Rolland, C., Achour, C.B.: Guiding the construction of textual use case specifications. Data Knowl. Eng. 25(1-2), 125–160 (1998)
14. Rui, K., Butler, G.: Refactoring use case models : The metamodel. In: ACSC'03. CRPIT, vol. 16, pp. 301–308. ACS (2003)