An Iterative Requirements Engineering Framework based on Formal Concept Analysis and C-K theory

Jonas Poelmans¹, Guido Dedene¹,², Monique Snoeck¹, Stijn Viaene¹,³

¹ KULeuven, Faculty of Business and Economics, Naamsestraat 69, 3000 Leuven, Belgium
² Universiteit van Amsterdam Business School, Roeterstraat 11, 1018 WB Amsterdam, The Netherlands
³ Vlerick Leuven Gent Management School, Vlamingenstraat 83, 3000 Leuven, Belgium

{jonas.poelmans, monique.snoeck, guido.dedene}@econ.kuleuven.be
stijn.viaene@vlerick.be

In this paper, we propose an expert system for iterative requirements engineering using Formal Concept Analysis. The requirements engineering approach is grounded in the theoretical framework of C-K theory. An essential result of this approach is that we obtain normalized class models. Compared to traditional UML class models, these normalized models are free of ambiguities such as many-to-many, optional-to-optional or reflexive associations which cause amongst others problems at design time. FCA has the benefit of providing a partial ordering of the objects in the conceptual model based on the use cases in which they participate. The four operators of the C-K design square give a clear structure to the requirements engineering process: elaboration, verification, modification and validation. In each of these steps the FCA lattice visualization plays a pivotal role. We empirically show how the strategy works by applying it to a set of case studies and a modeling experiment in which 20 students took part.

1. Introduction

During the conceptual modeling phase, user requirements are represented in a specification of what the system does as if there were a perfect implementation technology available (McMenamin et al. 1984). This is not a model of how an implementation works but of what an implementation must accomplish. Use cases and the conceptual domain model are the most important artifacts resulting from this phase. A use case is a system usage scenario involving one or more actors and the purpose of a use case specification is to describe the flow of events in detail including how the use case starts, ends, modifies the system and interacts with actors. By analyzing the domain of interest, identifying and modeling relevant entities and relationships we obtain the conceptual domain model. In (Lindland et al. 1994) a framework is presented for evaluating the quality of conceptual models. A distinction is made between syntactic and semantic quality. Several methods have been introduced for detecting and handling syntactic problems such as inconsistencies. 75% of these techniques, such as model checkers, theorem provers, coherence checkers, etc. are formal (Lucas et al. 2009) and unfortunately not very popular in the industrial software development community (Beckert et al. 2006). This unpopularity is usually due to the fact that these approaches are difficult for modelers to use directly and that the feedback they offer, is usually poor and difficult for non-experts to understand. Semantic validation of requirements and conceptual models is a social rather than a technical process, which is inherently subjective and difficult to formalize (Vliet et al. 2000). While some errors can be detected automatically, most errors can only be detected with the involvement of humans (Moody 1998) since a conceptual model can only be evaluated against people's (tacit) needs, desires and expectations.

According to Wieringa et al. (2006) one of the problems is that we miss sound methodology that captures the essential elements of requirements engineering. The research on requirements engineering and conceptual modelling quality which has been done so far seems to have had little impact or practice (Moody 2005). Few of the proposals have been
widely accepted in practice and many have never been applied outside a research environment. Several authors claimed that researchers need to address the issue of practitioner acceptance (Kaindl et al. 2002, Moody 2003). According to the literature on quality management, the most effective way to improve quality of products is to improve the process by which they are produced (Evans et al. 2004). So far, conceptual modeling quality research has focused almost exclusively on product quality: very few proposals even mention the issue of process quality (Maier et al. 1999, Maier et al. 2001). In requirements engineering, we can distinguish 4 broad categories of activities: elaboration of requirements artifacts, syntactic verification, modification of the model and validation of the semantics of the model with the business users. We propose Formal Concept Analysis (FCA) as a human-centered and easily understandable instrument to support the modeling of a software system (Ganter et al. 1999, Wille 1999). It is a technique for mathematically describing and visualizing concepts and their interrelationships. In particular, the intuitive visual display was found to be of major importance during a number of case studies and a modeling case in which 20 students took part. The lattice helped in stimulating conscious reasoning over syntactic and semantic errors, inconsistencies and different modeling choices that were made. Amongst others, we gained insight in missing objects, missing or faulty assigned operations, wrong dependencies, alternative solutions, etc. FCA allows the user to reason over the semantics, consistency and relations among UML models. A lattice can automatically be derived from an object - use case interaction matrix and easily be transformed into a UML class diagram. This class model construction procedure based on FCA has the additional advantage of resulting in “normalized” class diagrams. These models contain no more many-to-many relations, no more optional-to-optional relations and no more reflexive relations, leading to less ambiguous class diagrams.

The requirements engineering process is framed in the C-K design science theory (Hatchuel et al. 2003) and each of the four categories of activities can be mapped to one of the four operators of the C-K design square. At the core of the method are multiple successive iterations through a learning loop. The actionable information in the K space, i.e. the use cases and conceptual model, are transformed to an FCA lattice which can be used for formal verification of the model and proposing modifications to the model. The results are fed back to the domain experts, and the semantic validity of the model is analyzed together with the business user. The FCA lattices serve here as a communication instrument.

The remainder of this paper is composed as follows. In section 2 we introduce the essentials of conceptual model quality and UML class model normalization. Section 3 discusses FCA, C-K theory and the relevance of these techniques in requirements engineering. Section 4 shows how C-K theory can be used as a framework for iterative requirements engineering and the relevance is showcased with multiple case studies. Section 5 describes a validation experiment. In section 6 related work is presented. Finally, section 7 concludes the discussion.

2. Requirements engineering artifacts

2.1 Use cases, conceptual domain model and quality

The quality of the end product depends greatly on the accuracy of the requirements specification and developers are more and more concentrating on how to improve the early stages of development. Both use cases and the conceptual model are important parts of early development of a software system. According to (Belgamo et al. 2004), these requirements models deserve special attention, since it is in the requirements engineering phase where
substantial communication difficulties concentrate and many defects may be introduced in the artifacts. Use Cases were introduced in OOSE (Jacobson et al. 1992) and describe the interactions between the system and the external actors. Such an interaction does not have to be atomic and is usually decomposed into steps indicated in the use case specification. An actor is a specific role played by a system user and represents a category of users that share similar behavior when using the system. By users we mean both human beings as well as external systems or devices communicating with the system. An actor is regarded as a class and users as instances of this class. A use case is a system usage scenario involving one or more actors and the purpose of a use case specification is to describe the flow of events in detail, including how the use case starts, ends, modifies the system and interacts with actors.

One of the major activities includes finding out which classes the software will need in order to satisfy the requirements described in the use cases. The behavior in a system should be exactly that which is required to provide the use case to the users of the system. A conceptual model is a collection of concepts linked together to form a model. Another important step is the allocation of the required functionality to an entity or entities in the conceptual model for each use case. For each step described in the use case specifications, a responsibility should be identified and allocated to an entity. This is a complex but unavoidable task (Insfran et al. 2002). One of the biggest challenges facing software projects is determining when and how to begin the transition from specifying requirements to working on a system design (Reed 2002). Incomplete or incorrect requirements carry the risk of formulating a design based on sketchy requirements. In (Lindland et al. 1994), a framework is presented for evaluating the quality of conceptual models. A distinction is made between syntactic and semantic quality. Semantic quality issues arise when the model lacks something that the domain contains, or it can include something the domain does not have. In other words, the more similar the model and the domain, the better the semantic quality. The two major semantic goals to be achieved are validity and completeness. Validity means that all statements made by the model are correct and relevant to the problem. Completeness means that the model contains all the statements about the domain that are correct and relevant. In the quality management literature the distinction is also often made between product and process quality (Checkland 1991): product quality focuses on the quality of the end product. Product quality criteria are used to conduct inspections of the finished product and to detect and correct defects. Process quality focuses on the quality of the production process. Process quality focuses on defect prevention rather than detection, and aims to reduce reliance on mass inspections as a way of achieving quality (Deming 1986).

2.2 UML class model normalization

We propose a new best practice for UML class diagrams called normalization. The goal of normalization is to reduce the ambiguity in conceptual models. Currently, there exists a lot of confusion in the literature about best practices in UML modelling and normalization of conceptual models. According to Frisendal et al. (2010), the biggest problem with UML is its complexity. In business concept modelling, intuition is obstructed by unnecessary complexity such as meta-constructs like aggregation, composition, many-to-many associations, inheritance, etc. which are not really necessary for business users to understand. Ambler (2009) defines class normalization as a process of applying simple rules to reduce coupling and increase cohesion within the object designs. A related approach for improving object diagrams is refactoring (Fowler 1999) which however is typically performed on source code instead of models. Falleri et al. (2008) define normalization as removing all redundancies from class models and finding abstractions. They use FCA and Relational Concept Analysis (RCA) to find possible class, association, attribute or method generalizations in models with
the aim of improving their abstraction level (Falleri et al. 2008b).

We start our discussion with 4 examples of problems typically associated with traditional UML class diagrams and how developers can benefit from normalization. Fig. 1 contains a reflexive association example. The model aims to represent a flow of activities. However, the UML diagram does not reveal which the start or ending activity is. Although, the UML standard allows to give a name to the start and end of an association, this is often forgotten by software developers resulting in an ambiguous diagram. Therefore, this flow should be modeled as a separate class. A flow is always characterized by two associations, one with the start activity and one with the end activity. These relationships are mandatory for the association-ends with cardinality 1. The normalized model is displayed in Fig. 2.

![Fig. 1. class diagram with a reflexive association.](image1)

![Fig. 2. normalized class diagram without reflexive association.](image2)

The second example in Fig. 3 shows an often encountered many-to-many association. Li et al. (2001) advise to model associative classes as a separate class and decompose the association into two associations between the two classes and the newly added class. This decomposition changes the many-to-many association into one-to-many associations that are much easier to realize than many-to-many associations. In object models, associations are instantiated as power sets. However, still an improvement is possible in their new model. The authors introduced an association between the 2 original classes which is superfluous since the new object was introduced which has a mandatory association with both entities. This association can be removed.

Also interesting to consider is the literature on Entity-Relationship modeling (Lanzerini et al. 1990) since UML diagrams can be derived from ER schemata. The standard binary association in UML and ER have the attribute unique on each end. Objects at unique ends are counted only once if they are connected to a particular object several times. At non-unique
ends every connection is counted even if several of them lead to the same object. If this property is set to unique, the instantiations of the association form a set, if they are non-unique, a bag. The authors of (University of Cape Town 2007) suggest to represent many-to-many relationships as two one-to-many relationships involving a new entity since it is difficult to implement a many-to-many relationship in a database system. This new structure can be implemented within a relational database system. Feinerer et al. (2007) study this unique and non-unique property. Standard ER does not allow non-unique, but UML 2.0 superstructure specification (Object Management Group 2005) does not make statements about instantiations of associations being a set or bag but the tools are standard on unique. Standard in all UML tools, this multiplicity property for many-to-many associations is set to unique (1) which is another argument for the normalization of many-to-many relationships. If isUnique is set to false, links carry an additional identifier apart from their end values. Many-to-many associations as a consequence make the diagrams and query definition unnecessarily complex and analysts cannot model more than 1 relation between the same objects if unique is set on true.

Each instantiation Rent_i of this association is a tuple (Person_i, Car_i). A problem with this representation is that the same person cannot rent the same car more than once since a tuple can only occur once in a set. This problem can be resolved by instantiating this association with an extra class. Again we see that the two association-ends are mandatory for Person and Car. The normalized model is displayed in Fig. 4.

Fig. 3. class diagram with many-to-many association.

![Fig. 3. class diagram with many-to-many association.](image)

Fig. 4. normalized class diagram without many-to-many association.

Figure 5 contains an example of an optional-to-optional association. In particular when it is important to keep a history record for the association and to perform querying afterwards, optional-to-optional associations should be instantiated by an extra class. For example, a Person can be enrolled in 0 or 1 Session. A Session can have 0 or more participants. The information related to an enrolment in a session cannot be kept by the class Person nor Session unless null values are allowed. The presence of null values may result in unpredictable behavior when queries are performed on the model. This association should be instantiated by an extra class and the information should be kept by this separate class.
Registration. This class Registration will contain all information about instances of this association which can be considered as a best practice. Again, this relationship is mandatory on the association-ends of Person and Session. The normalized model is displayed in Fig. 6.

Fig. 5. class diagram with optional-to-optional association

![Diagram](image1)

Fig. 6. normalized class diagram without optional-to-optional association.

![Diagram](image2)

Figure 7 contains an example of an association between 3 partners, indicated by a diamond in UML notation. UML defines an n-ary association as linking n classes, n > 2 and at each end is a multiplicity and uniqueness constraint. According to Genova et al. (2001), understanding n-ary associations is often very difficult for modellers and analysts. The multiplicity values typically specified for n-ary associations provide only partial understanding and are incompletely defined by UML. The authors reveal an ambiguity in the definition of UML minimum multiplicity for n-ary associations and three alternative interpretations are presented, each with its own problems and unexpected consequences. According to the author, many modellers use the ternary symbol in Fig. 7 as an abbreviated version of a ternary association with a hidden binary association. The limping links interpretation has ternary links that only link two instances and leave a blank for the third one. This option is however semantically weak and contradicts the UML definition of n-ary association. The other two interpretations are actual pairs which implies that minimum multiplicity must always be 1, which is not consistent with documentation and practice, the potential pairs interpretation seems correct but has a strange effect when value is 1. The authors propose a different notation similar to our normalization proposal. Based on the Merise method (Rochfeld 1986), the ternary association is replaced by a new entity and three binary associations that simulate the ternary association. This entity is called the intersection entity or associative entity (Song 1995). Each instantiation of this association is a triple (Designer_i, Tool_i, Project_i). Again this triple can only occur once in a set. By instantiating this association with a class, a second usage relation between a designer and a tool for a project becomes possible. Again, the association-ends with the weak entity type Usage are
mandatory for the classes Designer, Tool and Project. The normalized model is displayed in Fig. 8.

![Fig. 7. class diagram with association between more than 2 partners and UML diamond notation.](image1)

![Fig. 8. normalized class diagram without diamond notation.](image2)

The special cases of aggregation and composition can be interpreted as follows. In Fig. 9 we see that the filled diamond denoting composition in UML is a short notation for the diagram in Fig. 10. In Fig. 11, we see the same for the white diamond denoting aggregation. This diagram in Fig. 12 can be further normalized to the model underneath it.

![School](image3)

![Department](image4)
Fig. 9. class diagram with composition relationship.

Fig. 10. normalized class diagram.

Fig. 11. class diagram with aggregation relationship.
To summarize, each class diagram can be normalized. A normalized class diagram has the following properties. First, it only contains binary associations and no more associations between multiple parties. In other words, the diamond notation from UML is not needed anymore. Second, there are no more reflexive associations, i.e. from a class to itself. Third, each binary association has at least 1 mandatory side, i.e. with cardinality 1. On the other side, there can be cardinalities 1, 0..1, 0..*, etc. In other words, in a normalized class diagram there are no more many-to-many and optional-to-optional associations since 1 side is always mandatory.

Fig. 13 and Fig. 14 contains an example of a normalization procedure. Normalization has some advantages. First, normalization is a process that converges to a unique solution. Second, normalized class diagrams automatically lead to normalized database schema during implementation. Normalization does not need to happen at database level since it already was performed at class level.

The result of repeating this normalization procedure is a normalized class schema that is partially ordered to form a lattice structure. In this lattice structure, each association-end on the upper side has cardinality “1”.

2.3 UML associations and the use case-entity interaction matrix

A question that remains unanswered by UML is when we should model an association. Nor in UML, nor in ER-modeling there exist objective criteria to determine when an association
should be introduced. A possible answer was introduced in (Snoeck et al. 1996). An association should be introduced when 2 object types have something in common for a certain span in time. In other words, that they have a part of their life cycle in common.

We only consider business use cases, while IO or system use cases are not included in our discussion. We also require that the use cases are atomic, i.e. that they cannot be further decomposed. An atomic use case will change the system from one state into another and is defined in terms of how this use case will create or delete an object or form or break an association between two instances. Such an atomic use case is equivalent to a system operation in (Larman 1998). The notion of joint action (D' Souza 1998) can be used to represent an atomic use case.

For example, a use case like “manage customer” should be decomposed into “create customer”, “change customer” and “end customer”. We also do not consider extend and include operations on use cases. In the situation where two objects share a part of their lifecycle, there is an objective reason to introduce an association between them. After composing the use case-entity interaction matrix, we use FCA to come up with a clustering. Entities that participate in the same use cases are grouped in FCA concepts based on the use cases in which they participate. In section 4 we zoom in on FCA’s relevance in conceptual modeling.

3. FCA and C-K theory essentials

The Concept-Knowledge theory (C-K theory) was initially proposed and further developed by Hatchuel et al. (1996), Hatchuel et al. (1999) and Hatchuel et al. (2002). C-K theory is a unified design theory that defines design reasoning dynamics as a joint expansion of the Concept (C) and Knowledge (K) spaces through a series of continuous transformations within and between the two spaces (Hatchuel et al. 2003). C-K theory makes a formal distinction between Concepts and Knowledge: the knowledge space consists of propositions with logical status (i.e. either true or false) for a designer, whereas the concept space consists of propositions without logical status in the knowledge space. According to Hatchuel et al. (2004), concepts have the potential to be transformed into propositions of K but are not themselves elements of K. The transformations within and between the concept and knowledge spaces are realized by the application of four operators: concept $\rightarrow$ knowledge, knowledge $\rightarrow$ concept, concept $\rightarrow$ concept and knowledge $\rightarrow$ knowledge. These transformations form what Hatchuel calls the design square, which represents the fundamental structure of the design process. The last two operators remain within the concept and knowledge spaces. The first two operators cross the boundary between the Concept and Knowledge domains and reflect a change in the logical status of the propositions under consideration by the designer (from no logical status to true or false, and vice versa).
Design reasoning is modeled as the co-evolution of C and K. Proceeding from K to C, new concepts are formed with existing knowledge. A concept can be expanded by adding, removing or varying some attributes (a “partition” of the concept). Conversely, moving from C to K, designers create new knowledge either to validate a concept or to test a hypothesis, for instance through experimentation or by combining expertise. The iterative interaction between the two spaces is illustrated in Fig. 15. The beauty of C-K theory is that it offers a better understanding of an expansive process. The combination of existing knowledge creates new concepts (i.e. conceptualisation), but the activation and validation of these concepts may also generate new knowledge from which once again new concepts can arise.

### 3.1 FCA essentials

Formal Concept Analysis (Ganter et al. 1999, Wille 1982) is a recent mathematical framework that underlies many methods of knowledge discovery and data analysis. The starting point of FCA is triple of sets (M, F, T) called a formal context where $T \subseteq M \times F$ is a binary relation. This triple can be considered as a database table consisting of set of rows $M$ (called objects), columns $F$ (called attributes) and crosses representing relation $T$. An example of a cross table is displayed in Table 1 where objects are domain entities, attributes are use cases, and incidence relation shows which entities participate in which use cases.

<table>
<thead>
<tr>
<th></th>
<th>ENTER</th>
<th>LEAVE</th>
<th>LEND</th>
<th>APPLY</th>
<th>CLASS</th>
<th>LEND</th>
<th>CLASS</th>
<th>LEND</th>
<th>LEND</th>
<th>REWARD</th>
<th>REWARD</th>
<th>RETURN</th>
<th>RETURN</th>
<th>SELL</th>
<th>SELL</th>
<th>SELL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Member</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Book</td>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Loan</td>
<td>X</td>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Given a formal context, we then derive concepts and order them according to a subconcept-superconcept relation. This order makes a lattice which can be visualized by a line diagram.
The notion of concept is central to FCA. The way FCA looks at concepts is in line with the international standard ISO 704, that formulates the following definition: A concept is considered to be a unit of thought constituted of two parts: its extent and its intent. The extent consists of all objects belonging to the concept, while the intent comprises all attributes shared by those objects. Typically, one would think here about informational attributes but—in line with an object oriented approach—one can just as well consider behavioral attributes such as reaction to events or participation in processes. So let us illustrate the notion of concept of a formal context using the data in Table 1. For a set of objects $O \subseteq M$, the use cases that are common to all objects $o$ in the set $O$ can be identified, written $\sigma(O)$, via:

$$\sigma(O) = \{ f \in F \mid \forall o \in O : (o, f) \in T \}$$

Take for example the set $O \subseteq M$ consisting of objects Member, Book and Loan. This set $O$ of objects is closely connected to a set $A$ consisting of the attributes “borrow”, “renew”, “return” and “lose”, being the use cases shared by the objects in $O$. That is:

$$\sigma([\text{Member, Book, Loan}]) = \{\text{borrow, renew, return, lose}\}$$

Reversely, for a set of attributes $A$, we can define the set of all objects that share all attributes in $A$:

$$O = \tau(A) = \{ i \in M \mid \forall f \in A : (i, f) \in T \}$$

If we take as example the set of events of Loan, namely $\{\text{borrow, renew, return, lose}\}$, we get to the set $O \subseteq M$ consisting of the objects Member, Book and Loan. That is to say:

$$\tau(\{\text{borrow, renew, return, lose}\}) = \{\text{Member, Book, Loan}\}$$

As one can see, there is a natural relationship between $O$ as the set of all objects sharing all attributes of $A$, and $A$ as the set of all attributes that are valid descriptions for all the objects contained in $O$. Each such pair $(O, A)$ is called a formal concept (or concept) of the given context. The set $A = \sigma(O)$ is called the intent, while $O = \tau(A)$ is called the extent of the concept $(O, A)$.

Notice that concepts are always maximal in the sense that the set $O$ contains all objects that share the attributes of $A$ and that $A$ contains all shared attributes of the objects in $O$.

Moreover, there is a natural hierarchical ordering relation between the concepts of a given context that is called the subconcept-superconcept relation 1.

$$(O_1, A_1) \leq (O_2, A_2) \iff (O_1 \subseteq O_2 \wedge A_2 \subseteq A_1)$$

A concept $d = (O_1, A_1)$ is called a subconcept of a concept $e = (O_2, A_2)$ (or equivalently, $e$ is called a superconcept of a concept $d$) if the extent of $d$ is a subset of the extent of $e$ (or equivalently, if the intent of $d$ is a superset of the intent of $e$). For example, the concept with intent “enter,” “leave,” “lose,” “return,” “renew,” and “borrow” is a subconcept of the concept with intent “lose,” “return,” “renew,” and “borrow.” With reference to Table 1, the extent of the latter is composed of object types Loan, Member and Book, while the extent of the former is composed of object type Member.

---

1 The terms subconcept and superconcept are used here in an FCA-context and should not be confused with the notions of subclass and superclass as used in the OO-paradigm.
The set of all concepts of a formal context combined with the subconcept-superconcept relation defined for these concepts gives rise to the mathematical structure of a complete lattice, called the concept lattice $\beta(M,F,T)$ of the context. The latter is made accessible to human reasoning by using the representation of a (labeled) line diagram. The line diagram in Figure 16, for example, is a compact representation of the concept lattice of the formal context abstracted from Table 1. The circles or nodes in this line diagram represent the formal concepts. The shaded boxes (upward) linked to a node represent the attributes used to name the concept. The non-shaded boxes (downward) linked to the node represent the objects used to name the concept. The information contained in the formal context of Table 1 can be distilled from the line diagram in Figure 16 by applying the following reading rule: An object $g$ is described by an attribute $m$ if and only if there is an ascending path from the node named by $g$ to the node named by $m$. For example, Member is described by the attributes “enter”, “leave”, “lose”, “return”, “renew” and “borrow”.

Retrieving the extent of a formal concept from a line diagram such as the one in Figure 16 implies collecting all objects on all paths leading down from the corresponding node. In this example, the extent associated with the upper node is \{Loan, Book, Member\}. To retrieve the intent of a formal concept one traces all paths leading up from the corresponding node in order to collect all attributes. In this example, the second concept in row two is defined by the attributes “sell,” “classify,” “acquire,” “lose,” “renew,” “return,” and “borrow.” The top and bottom concepts in the lattice are special. The top concept contains all objects in its extent. The bottom concept contains all attributes in its intension. A concept is a subconcept of all concepts that can be reached by travelling upward. This concept will inherit all attributes associated with these superconcepts. In our example, the first node on the second row with extent \{Member\} is a subconcept of the top node with extension \{Loan, Member, Book\}.

In FCA, the concept generated by an object type $P$ is defined as $\gamma(P) = (\tau(\sigma(P)), \sigma(P))$ and the concept generated by an event type $b$ as $\lambda(b) = (\tau(b), \sigma(\tau(b)))$. In the line diagram, the nodes are labelled by the object types which generate the corresponding concept $C$. These are called the own object types of the concept $C$. An object class $A$ is called an owner class for a use case if this object class $A$ is involved in this use case and there is no class which is existence dependent of $A$ and which is also involved in this use case.

4. Iterative requirements engineering process using FCA

According to Wieringa et al. (2006), many of the papers published in the requirements engineering field describe techniques for use in requirements engineering practice: for
example, how to improve the process of negotiating requirements or how to build use case models, etc. Unfortunately, there are few research papers that investigate the properties of these techniques, or the problems to be solved by these techniques (Wieringa 2005a, Wieringa 2006b). According to the authors, the absence of such research prevents the transfer of results of requirements engineering research to practice. Companies will hesitate to adopt techniques of which the properties are not well known, not thoroughly investigated or for which it has not been investigated which problems they solve and under which conditions. The methodological framework we use is C-K theory, which gives a clear structure to the process of iterating back and forth between the human actor and the documents describing the system under development.

A problem is a difference between what is perceived and what is desired, that we want to reduce (Wieringa 2003). An action problem is a desire to change the world; a knowledge problem is a desire to increase our knowledge about the world. Action problems can be classified into two kinds. A design problem is a desire to specify a change and an implementation problem is a desire to implement a specified change. To solve a design problem, we must do two things: Analyze the problem and specify a solution. According to (Wieringa 2003), requirements engineering is the problem analysis part of a design problem. It is about a knowledge problem, which the engineer tries to solve by building a theory about the domain of this problem. This knowledge creation process can be seen as a special case of the unified theory on design, called C-K theory. The notion of design as an expansive process addressed in design theories such as C-K theory should not be confused with the software design phase; although a software design process can be seen as a special case of design reasoning in C-K theory. In this section, we discuss how the requirements engineering process can be framed using C-K theory.

4.1 C-K theory in requirements engineering

Modeling software systems contains both formal and non-formal steps. These non-formal steps should not be unpredictable or irrational, but should follow a systematic way of thinking (Marincic et al. 2008). In some papers on formal methods, the modeling process is presented as if modelers had all the knowledge about the system before they started modeling. In that case it is possible to build a model in a strictly top-down manner. But modelers usually do not know everything up front about the system that they are modeling. One of the essential aspects of the requirements modeling process is iteratively increasing the knowledge available about the system. The source of information can be technical documents or domain experts. Most likely, the modeler does not have a complete knowledge about the system. The need for a structured approach has been described in the many papers on the soundness issues in requirements engineering. In this section, we give a clear structure to this modeling process by using C-K theory. We particularly focus on the iterative refinement steps, which describe how the model grows from an initial, sketchy, general description to its final version. In our approach, the expert is the driving force behind the modeling of software systems.

Requirements engineering is basically a process of iterating back and forth between a concept and a knowledge space. The knowledge space contains the information available to the domain expert including initial sketchy requirements for the system under development. This knowledge is then conceptually organized and visualized using the FCA lattices. We perform this conceptualization to put the actionable knowledge available in the K space under scrutiny. In the C space, these lattices are used for verification of the model and to detect inconsistencies, anomalies, missing entities, missing use cases, etc. These newly discovered concepts, anomalies and concept gaps are then activated and used to improve the current model. In the K space, these findings are fed back to the domain expert and the lattices are
used for validation of the requirements model. They serve as a communication instrument between the software modeler and the business user, for whom the technical jargon of the software engineer is often difficult to understand. Unspoken assumptions and desires on semantics of the model should be made explicit and communicated to all stakeholders (Wieringa 2001).

![Diagram](image)

**Fig. 17. Iterative C-K requirements engineering process**

This process is graphically described in Figure 17. During the K → C step, the FCA lattices are constructed that form the core artifacts of our requirements model construction, verification and validation method. These lattices are built from the entity - use case interaction matrix, but can also be based on a use case - operations or entity - operations interaction matrix. The entity - use case interaction lattice partially orders the entities in the conceptual model based on the use cases in which they participate. This lattice may reveal missing concepts, entities and use cases, but also issues such as use case participations that should be added or removed. Also anomalies in the behavioral side of the model can more easily be detected because of the non-hierarchical partial order relation. The use case - operations interaction lattice gives insight into the operations needed to complete the use cases. Missing operations, use cases that should or should not have certain operations in their execution scenarios, use cases that should have certain operations in common, etc. can be identified. The entity - operations lattice gives additional insight into the behavior of entities in the conceptual model. In the special case of atomic use cases, i.e. the use cases are not further decomposable, every use case corresponds to an operation in the conceptual model and vice versa. In this case only one lattice, namely the entity-use case lattice is needed. These lattices are used during the C → C step for formal verification of the model, as a human-centered instrument that facilitates the detection of inconsistencies, anomalies, etc. The original model, the discovered anomalies and the proposed modifications are returned to and discussed with the domain expert during the C → K step. The lattices serve as a communication instrument, between the developer and the domain expert. During the K → K step, these lattices are used for semantic validation by the domain expert. This may result in the addition, modification or deletion of use cases, modifications in the conceptual model, etc. These artifacts may be used as input for a new iteration through the C-K loop.
4.2 FCA lattice properties and relation with software artifacts

A conceptual model is a dual structure of concepts and their instances called objects and behavioral elements of the model (use cases) in which they participate. These concepts or classes are related through associations. In this section we intend to formalize this dual conceptual model structure with FCA. FCA has a well established mathematical foundation whereas conceptual modeling is to some extent still an ambiguous discipline.

After composing the use case-entity interaction matrix, we use FCA to come up with a clustering. Entities that participate in the same use cases are grouped in FCA concepts. These shared use case participations indicate a shared lifecycle of the objects and the lines interconnecting the concepts in an FCA lattice can be used as associations in a UML class diagram between the objects belonging to the interconnected concepts. From this FCA lattice, a UML class diagram can be automatically derived since an FCA lattice based on a correct entity-use case interaction matrix is isomorphic to the correct UML model. Each line interconnecting 2 concepts can be seen as a direct association between the own objects in the extent of the corresponding concepts. A line between 2 concepts in the lattice means that between 2 instantiations exactly 2 own classes in the extent of the 2 concepts, one tuple can be created. This follows from the object-use case interaction matrix.

One of the main contentions of this section is that FCA leads to a normalized class diagram. One of the consequences is that FCA can be used to detect missing classes. An important benefit of FCA over other techniques is its non-hierarchical partial ordering of concepts. This is more expressive than traditional hierarchical tree-like structures, which was already stated in (Christopher 1965). Hierarchical decomposition gives a distorted and simplified view that does not necessarily conform to reality.

Table 2 summarizes the properties of an FCA lattice based on a use case-entity interaction matrix and how it can be used to distill the correct model from this interaction matrix and the original UML model. In case there is a discrepancy between one of these columns for the model, solutions can be proposed based on these best practice guidelines. The first column contains an interesting observation made by looking at the FCA lattice. Since this lattice is based on the entity-use case matrix, the corresponding statement in the third column is true. The statements made in column 2 are best practices for the UML model that were introduced to make the use case–entity interaction matrix and UML model consistent with each other. Multiple possibilities for model revision paths can be associated with one lattice observation. For example, take the second row of Table 2 if there is no direct line between the concepts with A and B as own objects in their extent in the FCA lattice, then there should be no direct association between A and B in the UML model. This is required because the UML model is isomorphic with the FCA lattice unless of course the use case-entity interaction matrix contains an error. In the use case-entity interaction matrix both A and B should have use cases in which either A or B does not participate. If the domain expert however says that there should be a direct association between A and B which is mandatory for A in the UML model, then the FCA lattice flags an error in the use case-entity interaction matrix and use case participations should be propagated from A to B. This will have the consequence that there will appear a direct line in the FCA lattice between A and B. FCA can be used to flag errors in both the UML model and use-case entity interaction matrix.

Table 2. Normalization guidelines

<table>
<thead>
<tr>
<th>Entity-use case FCA lattice</th>
<th>Normalized UML model</th>
<th>Use case-entity matrix</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Entity A belongs to a concept</td>
<td>There should be a direct Entity A should participate</td>
<td></td>
</tr>
</tbody>
</table>

Table 2: Normalization guidelines
lower in the lattice than $B$'s concept and there is a direct line between both concepts. association between $A$ and $B$, which is mandatory (cardinality “1”) for $B$. $B$ is existence dependent of $A$.

2 There is no direct line between concepts with $A$ and $B$ as own objects in extent. There should be no direct association between $A$ and $B$. $A$ and $B$ have use cases in which either $A$ or $B$ doesn't participate.

3 There is a concept with 2 or more own use cases shared by $A$ and $B$ and no owner entity. Replace direct (many-to-many) association by contract entity with which association is mandatory for $A$ and $B$. $A$ and $B$ share 2 or more use cases and there is no entity which functions as a contract between $A$ and $B$ and which participates only in these 2 or more use cases.

4 There is no upward path from entity $A$’s concept to $B$’s concept. On implementation level this means $A$ can not call the proprietary operations of $B$. Entity $A$ does not participate in all the use cases in which $B$ participates.

5 There is a node $R$ with two use cases $a$ and $b$ as label and no own entities. Entities $A$ and $B$ lower in the lattice participate in these use cases and there is no upward path between $A$ and $B$ nor are there any entities on the path from $A$ to $R$ or $B$ to $R$.

There is no association between $A$ and $B$ nor a contract entity which interconnects them. There is no entity $S$ that participates in use cases $a$ and $b$. $A$ and $B$ do not participate in the creation and deletion cases of $S$.

Table 2 contains the different types of normalization steps that can be undertaken. A sequence of normalization steps is called a normalization path and each path converges to a unique solution. In each case study we will show the relevance of normalization. In section 4.5, we see an application of the rule in row 3 of Table 2. In section 4.6 we see a combination of applying rules in row 1 and 2. In section 4.4 we see an application of the rule in row 5.

The FCA lattice components that facilitate the detection of anomalies during semantic validation of the model are explained in Table 2. Table 3 summarizes the results of (Poelmans et al. 2010).

### Table 3. Model anomalies and FCA analysis components

<table>
<thead>
<tr>
<th>Model anomalies</th>
<th>FCA analysis component</th>
</tr>
</thead>
<tbody>
<tr>
<td>Missing or faulty use case parts</td>
<td>Duality of partial ordering based on entities and use cases</td>
</tr>
<tr>
<td></td>
<td>reveals faulty and incomplete participations</td>
</tr>
<tr>
<td>Missing proprietary use cases</td>
<td>We gain insight into the owned or proprietary use cases of</td>
</tr>
<tr>
<td></td>
<td>an entity by looking at the concept that owns this entity.</td>
</tr>
<tr>
<td></td>
<td>The</td>
</tr>
</tbody>
</table>
deletion of an entity owned use cases by this entity are attached as labels.

Missing entity A concept with two or more owned use cases and no own entity.

Superfluous entity A concept with more than one own entity indicates in the current model 2 entities have the same behavior and either one of them should be removed or one of them should participate in additional use cases.

The relevance of rule 1 is showcased in section 4.3 and 4.4. The relevance of rule 2 is showcased in section 4.3. The relevance of rule 3 is explained and demonstrated in section 4.5.

We first showcase our method on some toy examples to make the reader familiar with FCA-based conceptual modeling. Then we provide a real-life case study. We show the detection of some of the typical errors made by modelers using the rules of section 4.2.

4.3 Case study: Book Trader System

We now illustrate this process using the Book Trader system introduced in (Liang 2003). The collaboration diagrams and conceptual model displayed in Figures 18a – f contained some errors that remained undetected to the authors. These errors are:

- Entity Line does not participate in any use cases according to the Collaboration diagrams.
- Separate create and delete use cases are missing for entities Book, Order, Line, Customer and Invoice.
- In the collaboration diagrams, Order is involved in the provide-quantity operation which is in contradiction with the class diagram in which Line is having this operation.
- A better modelling option would be to allow new prospective customers to register. Currently, according to this model a Customer can only be registered as part of the use case Place Order.
- The class Invoice cannot be seen as an object in a business model since it has no further behaviour after its creation, however it should be included in the information architecture.

We found these errors while studying the three lattices and this analysis process will now be discussed in detail. During the K → C step, we constructed a cross table indicating which entities participate in which use cases. This table displayed in Table 4 is based on the collaboration diagrams and the conceptual model in the original paper and we will call it the entity-use case interaction matrix. Each collaboration diagram represents one use case. When an entity appears in the collaboration diagram, this is registered in the table with a C, M or E. The C indicates that the use case creates the entity, M indicates reads or modifies the entity and E indicates it terminates the entity.

| Table 4. Faulty entity-use case matrix |
The corresponding lattice is displayed in Fig. 19. During the C → C step we performed a syntactic verification of the model quality based on the lattice visualization. Using the lattice in Fig. 19, we found that the entity Line does not participate in any use cases according to the original collaboration diagrams. Second, when analyzing the attributes of the FCA concepts we found that separate create and delete use cases were missing for most entities defined. UML best practice guidelines advised the definition of both creation and deletion use cases (Rumbaugh et al. 2004). These are some clear modeling anomalies. In the C → K step we propose modifications to resolve these anomalies. In the K → K step the improved model is communicated to and validated with the domain expert.

Fig. 18a. Collaboration diagram (adapted from Liang 2003)

Fig. 18b. Collaboration diagram (adapted from Liang 2003)

Fig. 18c. Collaboration diagram (adapted from Liang 2003)
Fig. 18d. Collaboration diagram (adapted from Liang 2003)

Fig. 18e. Collaboration diagram (adapted from Liang 2003)

Fig. 18f. Class diagram (adapted from Liang 2003)
During the K → C step of our second iteration through the design square we constructed a matrix, displayed in Table 5 that maps the (non-atomic) use cases against the operations that instantiate them.

Table 5. Faulty use case-operations matrix

<table>
<thead>
<tr>
<th></th>
<th>CREATE</th>
<th>PROVIDE</th>
<th>PROVIDE</th>
<th>PROVIDE</th>
<th>PROVIDE</th>
<th>PROVIDE</th>
<th>PROVIDE</th>
<th>PROVIDE</th>
<th>PROVIDE</th>
<th>PROVIDE</th>
<th>PROVIDE</th>
<th>PROVIDE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Place order</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Check stock</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Deliver Book</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Generate invoice</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Check credit</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The corresponding lattice is displayed in Figure 20. During the C → C step the operations - use case lattice helped us to identify some of the semantic discrepancies between the original conceptual model and the real world, for example, we see that the record-new-customer operation only occurs in the use case Place_Order. New prospective customers cannot be created without an order. This is an unrealistic constraint that is also incorporated in the original domain model. One could consider to make each mandatory association into an optional association would not better reflect reality. During C → K and K → K, modifications are proposed and their semantics are discussed.
The third matrix, displayed in Table 6, maps the entities against the operations in which they are involved.

Table 6. Faulty entity-operations matrix

<table>
<thead>
<tr>
<th></th>
<th>CREATE</th>
<th>PROVIDE-ALL FEATURES</th>
<th>PROVIDE-TOTAL-COST</th>
<th>PROVIDE-QUANTITY</th>
<th>PROVIDE-NAME</th>
<th>PROVIDE-CREDIT-BALANCE</th>
<th>PROVIDE-ADDRESS</th>
<th>PROVISION-NEW-ORDER</th>
<th>REDUCE-ORDER</th>
<th>RECORD-NEW-ORDER</th>
<th>ASSIGN-CUST</th>
<th>NAME-CHANGE</th>
<th>CREDIT-BALANCE</th>
<th>REDUCE-CREDIT-BALANCE</th>
<th>REDUCE-NEW-Customer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Order</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Line</td>
<td></td>
<td>X</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Book</td>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Invoice</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Customer</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The corresponding lattice is displayed in Figure 21. When we look at the concept with entity Invoice in its extent, we see that Invoice only participates in a create event. The Invoice entity has no further behavior and should not be modeled as a separate class. Business modeling analysts will typically not consider this to be an object however it should of course be integrated within the data and information architecture. The creation of an invoice should be modeled as an operation or an event from which a document is generated. Again, creation and deletion operations are missing for most entities. Moreover, Line is involved in the provide-quantity operation which is in contradiction with the use case - entity interaction matrix.

Fig. 21. Entity – operations interaction matrix
4.4 Case study: Hotel Administration System

In this section, we showcase how the FCA lattices were used for the semantic validation of a hotel administration system model. Customers can make reservations for a particular room type. Reservations must be confirmed by a letter. If such letter is not received in time, the reservation is cancelled. When a guest checks in for the first time, his details are registered. At the end of the stay, an invoice is sent to the customer who made the reservation. Suppose a business analyst would come up with Table 7. Fig. 22 shows an excerpt of the initial UML model that was developed for this hotel administration system.

Table 7. Incorrect entity-use case matrix for the Hotel Administration

<table>
<thead>
<tr>
<th></th>
<th>CUSTOMER</th>
<th>TYPE</th>
<th>ROOM</th>
<th>VATION</th>
<th>RESER</th>
<th>ROOM</th>
<th>STAY</th>
<th>IDN</th>
<th>CONSUMPT</th>
<th>PAYMENT</th>
</tr>
</thead>
<tbody>
<tr>
<td>cr_customer</td>
<td>C</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>e_customer</td>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cr_room_type</td>
<td>C</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>e_room_type</td>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>reserve</td>
<td>M</td>
<td>M</td>
<td>C</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>confirm</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cancel</td>
<td>M</td>
<td>M</td>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>check_in</td>
<td>M</td>
<td>M</td>
<td>E</td>
<td>M</td>
<td>C</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>no_show</td>
<td>M</td>
<td>M</td>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cr_room</td>
<td>M</td>
<td>C</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>e_room</td>
<td>M</td>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>e_stay</td>
<td>M</td>
<td>M</td>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>consume</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>C</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>bill</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>invoice</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>C</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pay</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>remind</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>e_invoice</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

At first this UML model seems to be alright. During the $K \rightarrow C$ step the lattice in Fig. 22 was obtained from the use case – entity interaction matrix. In this example, the use cases were not further decomposable and every use case corresponds to an operation in the conceptual model. Such a use case can also be called an event. In this case, only one matrix namely the one that maps entities versus use cases is needed. The interplay between use cases and entities and the additional partial ordering relation helped however to reveal some semantic issues in the original UML model during the $C \rightarrow C$ step.
Fig. 22. Incorrect hotel administration model and entity-use case interaction lattice based on original incorrect matrix

We used the lattice for semantic validation of the model. First, the person or company who made the reservation for the guest, may be someone else than the person who is staying in the hotel. Second, we see that there is no upward path from the concept with entity Customer to the concept with use cases bill and consume. In other words, the entity Customer does not participate in the use cases bill and consume although guests of the hotel should be able to make consumptions. Second, Customer does not participate in the use case e-stay, whereas the guest of the hotel should be able to check out of the hotel at the end of his stay. Based on these observations we decided to add an entity Guest to the conceptual model during the C → K step. The observation that Customer is not associated with Room is perfectly alright. The Customer makes a Reservation for a Room type, a specific Room is only relevant for the guest of the hotel. Fig. 23 contains the correct UML model that was obtained after discussion with and validation by the domain experts during the K → K step and contains the entity-use case interaction lattice corresponding to this correct model.
Fig. 23. Correct hotel administration lattice and UML model which is isomorphic to this lattice

4.5 Case Study: Ordering system

This section discusses the development of the ordering system for computer hardware, office material, etc. of the university KULeuven in Belgium. This ordering process is a standardized process for KULeuven, where an order is placed when a request for ordering a computer is received from an employee. A request is sent to Dell to construct a computer. In a standard setting, payment is only made after the goods were delivered to the KUL. Fig 24 contains the initial UML class model for an excerpt of this system, which is not normalized and does not allow for the detection of potential conflicts in use case execution order. After building the use case-entity interaction matrix in Table 8 and lattice we obtain the lattice in Fig. 25.

During the C $\rightarrow$ C step we use FCA for formal verification of the models, detection of missing entities, detection of use cases for which no responsible entity has been assigned, etc. We see that Dell computer and KULeuven jointly participate in 3 main use cases. If this joint participation is not coordinated, both KULeuven and Dell computer can impose sequence constraints on the order of execution for these 3 use cases. This may result in a situation of deadlock. Indeed, for the KULeuven these use cases have a fixed order of execution, namely:
1. Order  
2. Deliver  
3. Pay

However, for computer manufacturer Dell who also participates in these use cases, the ordering of the use cases is:

1. Order  
2. Pay  
3. Deliver

We see there is one node in the lattice with no entity label and 3 use case labels attached. We see that the entities KULeuven and Dell participate in these use cases, but an entity responsible for coordinating this joint participation in these use cases is missing. The business processes of KULeuven and Dell have to communicate with each other and the sequence of use case execution should be coordinated. During the C → K step an extra entity named Contract is added as a contract between these two actors to guide and coordinate the collaboration. Figure 25 contains the UML model of an excerpt of this system. During the K → K step this modification is explained to the business user, together with the constraints imposed by this “contract” object type on use case execution order.

Table 8. Faulty entity-use case matrix

<table>
<thead>
<tr>
<th></th>
<th>CONSTRUCT</th>
<th>ISSUE-REQ</th>
<th>PAY</th>
<th>DELIVER</th>
<th>ORDER</th>
</tr>
</thead>
<tbody>
<tr>
<td>KULeuven</td>
<td>C</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
</tr>
<tr>
<td>Dell</td>
<td>C</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
</tr>
</tbody>
</table>

![Diagram](image)

Fig. 25. FCA lattice with missing entity and correct ordering system model with contract entity added after normalization

4.6 Case study: Elevator repair system

In this case study an engineer works for exactly one office. This engineer is responsible for repairing broken elevators. When an elevator fails, an interrupt is sent to an office and an engineering is sent to the elevator. Fig. 26 contains the initial UML model and Table 9 the entity-use case interaction matrix.
Table 9. Initial entity-use case interaction matrix

<table>
<thead>
<tr>
<th>Elevator</th>
<th>Office</th>
<th>Engineer</th>
<th>Report</th>
<th>Close</th>
<th>Interrupt</th>
<th>Change</th>
<th>Repair</th>
<th>Action</th>
<th>Erect</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>C</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

There are however some issues with this model, the model is not normalized and this has the following consequences. During the C \rightarrow C step we analyzed the requirements artifacts. The UML model shows a direct association between Office and Elevator. Assuming that the unique property has been set (which is reasonable, as unique is “standard”), this means that the same elevator cannot send an interrupt more than once to the same office. The associations between instantiated objects form a set and in a set, a tuple can only occur once. This is an undesirable property of the non-normalized UML model containing the many-to-many relationship. In a many-to-many relationship no entity is foreseen to record more than one association and its properties between the same instantiated objects.

Fig. 26. Initial UML model and FCA lattice for initial model of elevator repair system

In the FCA lattice based on the use case-entity interaction matrix in Fig. 26 we see no direct line between the concepts of Elevator and Office. The lattice indicates there are 2 events not coordinated by an entity and in which both entities participate. Based on the lattice recommendations a new entity should be added during the C \rightarrow K step with which both Elevator and Engineer have a mandatory association. This entity replaces the original direct association and coordinates the 2 use cases. We call this entity “Interrupt”.

In the FCA lattice there is no direct line between the concept of Engineer and the concept of Elevator. In other words, there is no association in the FCA lattice based on the use case-entity interaction matrix that corresponds to the direct many-to-many association between Engineer and Elevator in the UML model. The consequence of the direct many-to-many
association in the UML model is that the same Elevator cannot get a second repair by the same Engineer. This is also an undesired property which can be solved by normalization during the $C \rightarrow K$ step, i.e. in this case by replacing the association by a separate class associated to the entities Elevator and Engineer.

By studying the lattice during the $C \rightarrow C$ step we see there is no upward path from Office to Engineer. There is also no direct line from the concept with entity Office to the concept with entity Engineer. According to the FCA lattice based on the use case entity interaction matrix there should be no direct association between Office and Engineer or the use case-entity matrix should be modified. In the UML model, Engineer has a mandatory relationship with Office. When an Engineer is created, modified or archived, the corresponding Office should at least be notified (as part of the execution of these use cases) of these changes since Engineer is existence dependent of an office. Other alternatives are also possible, however for this case we chose this configuration, in practice the management will have to decide. If the Office object disappears the Engineer object should be terminated too since the mandatory association disappears. We see however that Office is not involved in all use cases in which Engineer is involved. This problem was solved by propagating the use case participations from Engineer to Office during the $C \rightarrow K$ step.

Interrupt use cases are not coordinated and Engineer is not involved in these use cases. Also repair use cases are not coordinated by an entity. FCA suggests to introduce Interrupt and RepairAction as 2 entities in the UML model. Then there is still no relation between RepairAction and Interrupt. This makes it impossible to know afterwards which Engineer worked on which Interrupt. There is no direct association between Interrupts and Repairs. As a consequence, Key Performance Indicators cannot be generated from this model and neither data mining nor business intelligence can be applied. The correct associations are missing fundamentally. The solution is to propagate all use case participations from RepairAction to Interrupt. In the FCA lattice, RepairAction will be right above Interrupt. In the UML diagram there is now a direct association which is mandatory for RepairAction. The correct UML model is displayed in Fig. 27, the correct entity-use case interaction matrix in Table 10 and the corresponding FCA lattice in Fig. 28.

![Normalized UML model](image)

**Fig. 27. Normalized UML model**

<table>
<thead>
<tr>
<th>Office</th>
<th>Elevator</th>
</tr>
</thead>
<tbody>
<tr>
<td>Engineer</td>
<td>RepairAction</td>
</tr>
</tbody>
</table>

**Table 10. Entity – use case interaction matrix for normalized model**
At first it seems the model contains a circular constraint, namely starting from RepairAction it is possible to navigate to Office by Interrupt, but it is also possible to navigate from Engineer to Office. In the original model both instantiations were considered to be the same, this invariant can be written in OCL as follows (Meyer 1997):

\[
\]

One of the consequences of this OTIS-based example is the following. If we impose this constraint, it is possible that some Offices receive too many requests and other Offices too little. This is an example in which the circular constraint should not be imposed. In the normalized model, engineers of no matter what Office can do a repair action, which solves the former problem. The result is that the notion of Office became a virtual entity in the system and not a physical office anymore.
5. Validation experiment

We empirically showed the relevance of our research with a modelling experiment in which 20 students took part.

5.1 Participants

The goal of the experiment was to evaluate the practical feasibility of the C-K design loop supported by FCA for software requirements engineering. The experiment conducted with the collaboration of students, consists of a modelling exercise in which they should distill entities and elementary processes from a textual description of a business process. Then, they should compose a matrix in which is indicated which entities participate in which processes. Their solution is then handed over to the data analyst who uses FCA to detect anomalies, missing object types, etc. and gives suggestions to the students for improving their original model. They then implement these changes. The first experiment is performed in collaboration with students of the course 'Ontwikkeling van Bedrijfstoepassingen' in March 2010. The experiment took place as part of an exercise session of the class.

5.2 Setup of experiment 1

The experiment was built around the Web Shop case.

**Web shop case:**
In the simple Web shop registered customers can create shopping carts. They can choose from various products, and select them to put them in the shopping cart. Once put in the shopping cart, a product can be confirmed (definitely, and hence archived) or removed from the cart. Of course, at the end of a shopping session, the cart must be paid, and is next delivered to the customer.

**Table 11. Formal context of web shop case**

<table>
<thead>
<tr>
<th></th>
<th>CT</th>
<th>CR PROD</th>
<th>CH PROD</th>
<th>CH Cust</th>
<th>MER</th>
<th>CR Cust</th>
<th>END Cust</th>
<th>CR CART</th>
<th>PAY CART</th>
<th>DELIVER CART</th>
<th>ADD ITEM</th>
<th>ARCH ITEM</th>
<th>REMOVE ITEM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Product</td>
<td>C</td>
<td>M</td>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Customer</td>
<td></td>
<td>C</td>
<td>M</td>
<td>E</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td></td>
<td>M</td>
<td>M</td>
<td></td>
</tr>
<tr>
<td>Shopcart</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cartitem</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Fig. 29. Web shop case

Table 11 contains the use case-entity interaction matrix and Fig. 29 the correct normalized class model. The C indicates that the use case creates the entity, M indicates that it modifies the entity and E that it deletes the entity. Fig. 30 contains the FCA lattice based on this correct matrix.

Fig. 30. FCA lattice based on correct matrix

5.3 Results of experiment 1

Starting from the use case-entity interaction matrix provided by the students, we derived an FCA lattice. None of the students succeeded in making a fully correct model for the system. The different solutions of students could however be categorized in four broad categories of structurally similar models containing some typical modeling errors. These FCA lattices and in particular the partial ordering of entities made it easy to identify faulty or missing associations and missing or faulty use cases and use case participators. During the semantic analysis of the students’ solutions the lattices made it easy to identify the different modeling options students chose and where they confused different options or mixed them together. This reasoning over models and model choices helped in achieving a uniform software model. We
found it more straightforward to distill this information on syntactic and semantic correctness from the FCA lattices than from the UML model and entity-use case interaction matrix. In these traditional artefacts information is more shattered and no partial ordering is available, whereas FCA provides a condensed and complete overview of the model's syntax and semantics.

The following errors were regularly found using the FCA lattices:

- For each entity the proprietary use cases are made visible as the labels connected to the concept corresponding to the entity. 16/20 students forgot one or more terminate entity use cases, whereas, only 3 students forgot a create entity use case. An excerpt of the lattices is displayed in Fig. 31 where the student forgot the EndCUSTOMER and DeliverCART use cases.

![Fig. 31. Termination use case omission](image)

- By following the lines upward, we can easily see in which use cases an entity participates. By following the lines downward one can see which objects participate in a use case. We found that for use case “RemoveITEM from cart”, “AddITEM to cart”, students tend to forget that also the initiator of this use case, i.e. the Customer, also participates in these use cases together with Product and ShopCart. Only 1/20 students modeled this correctly furthermore only 7/20 students did not forget that the initiator of “CrCART”, the Customer also participates in this use case. Fig. 32 contains an example of lattice excerpts showing that Customer does not participate in these 3 use cases.
Multiple use case participations shared by multiple entities but not coordinated by a contract entity can be found in the lattice as a node with own use cases but no own objects. For 17/20 students the lattice contained such a node where entities may impose different sequence constraints on use case execution order resulting in deadlock. Fig 33 contains as example of such a node.

Finally, the FCA models helped in reasoning over modeling choices made by students. From this lattice, the corresponding UML class diagram was distilled and fed back to the student with the found anomalies.

6. Related work
FCA has been used in various application domains including knowledge discovery, software engineering, information retrieval, etc. (Poelmans et al. 2010a). Notorious applications in knowledge discovery include the identification of domestic violence from statements made by victims of a violent incident (Poelmans et al. 2009, Poelmans et al. 2010c), human trafficking suspects (Poelmans et al. 2011) and radicalizing subjects (Elzinga et al. 2010) from observational police reports and quality of care issues from patient treatment data (Poelmans et al. 2010b). One of the first papers applying concept lattices to software analysis (Krone et al. 1996) analyzed the relationships between source code pieces and preprocessor variables in Unix system software. Later on multiple papers appeared on identifying modules or classes in legacy system code (e.g. Siff et al. 1997). More recently dynamic code analysis gained interests, e.g. Ammons et al. (2003) analyzed execution traces which they clustered with FCA to debug specifications in temporal logic. In Snelting (2005), Hesse et al. (2005) and Tilley et al. (2005) an overview of FCA applications in software engineering published in 2003 or earlier can be found. Also in requirements engineering there are some applications of FCA. Typically, use case descriptions are written in natural language although sometimes controlled vocabularies are used. Duwel (1999) and Duwel et al. (1998) used FCA to identify class candidates in use case descriptions. The authors considered the use cases as objects of a formal context and the nouns identified within the text were considered as a starting point for a class hierarchy. Tilley et al. present a case study applying Duwel’s approach to an Object-Z specification (Tilley et al. 2003). Richards et al. (2002) and Richards et al. (2002b) also applied FCA to use cases in an attempt to reconcile descriptions written by different stakeholders using a controlled vocabulary and grammar. According to the authors, the formal nature of this controlled language facilitates the analysis of use cases to identify misunderstandings, inconsistencies and conflicts. Moreover, similar concepts and differences in terminology were identified using concept lattices. FCA has also been used during the design phase of the software engineering process. In this section we focussed on early requirements engineering.

7. Conclusions

In this paper we showed the relevance of normalized class models in early requirements engineering. FCA was used to derive a concept lattice from the use case-entity interaction matrix. This lattice was used for syntactic verification and semantic validation of the UML model and use case – entity interaction matrix. After a successive number of normalization steps, a normalized UML class model is obtained with desirable properties such as the absence of many-to-many, optional-to-optional and reflexive associations. The iterative process of analyzing and improving the requirements artifacts was framed in C-K theory. The C-K loop consists of 4 main phases, elaboration, verification, modification and validation. In each phase, the visualization of FCA plays a pivotal role.

Acknowledgements

Jonas Poelmans is aspirant of the “Fonds voor Wetenschappelijk Onderzoek – Vlaanderen” or “Research Foundation Flanders”.

References


