Extending Formal concept Analysis with a Second Ordering Relationship on Concepts: a Software Engineering Case Study

Jonas Poelmans[1], Guido Dedene[1,4], Peter Eklund[3], Stijn Viaene[1,2], Monique Snoeck[1], Sergei Kuznetsov[5]

[1]K.U.Leuven, Faculty of Business and Economics, Naamsestraat 69, 3000 Leuven, Belgium
[2]Vlerick Leuven Gent Management School, Vlamingenstraat 83, 3000 Leuven, Belgium
[3]Centre for Digital Ecosystems, School of Information Systems and Technology, The University of Wollongong, Northfields Avenue, Wollongong NSW 2522, Australia
[4]Universiteit van Amsterdam Business School, Roetersstraat 11 1018 WB Amsterdam, The Netherlands
[5]State University Higher School of Economics (HSE), Moscow, Russia
{Jonas.Poelmans, Stijn.Viaene, Guido.Dedene, Monique.Snoeck}@econ.kuleuven.be
peklund@uow.edu.au
skuznetsov@hse.ru

Abstract. Formal Concept Analysis (FCA) has been used extensively in software engineering research. Traditional FCA theory has however a significant shortcoming when used for underpinning the construction of a conceptual domain model. Conceptual models typically contain two types of relations between classes, i.e. associations between classes and inheritance relationships, whereas FCA only imposes one ordering on concepts, namely the subconcept-superconcept relation. We eliminate this lack of expressive power of traditional FCA by introducing an extra dimension in FCA. Whereas triadic FCA imposes conditions on which objects participate in which attributes, we introduce a second ordering relation and we show how this expanded version of FCA can be used to model associations between classes and inheritance relationships in one mathematically sound model.

1 Introduction

Formal Concept Analysis (FCA) arose twenty-five years ago with the aim to restructure applied lattice and order theory (Wille 1982, Poelmans et al. 2010b). Since then, FCA has been used extensively in software engineering research (Tilley et al. 2005) and in particular in the areas of detailed design and software maintenance. It has been used for reorganizing existing class hierarchies and for refactoring and modifying existing code (Snelting 2000, Snelting 1998). FCA has also been used for identifying class candidates in legacy code (Tonella et al. 1999) and to mine legacy source code (Mens et al. 2005). In requirements analysis, FCA has been used to identify class candidates in use case descriptions (Duwel 1999, Duwel et al. 1998) and to reconcile descriptions written by different stakeholders using controlled vocabularies and grammars (Richards et al. 2002, Richards et al. 2002b). More recently FCA has also been used in combination with ontologies. Cimiano investigates how FCA and ontologies may complement one another from an application point of view (Cimiano et al. 2004). Bain applies FCA to identify structure in theories (Bain 2003). Rouane-Hacene et al. (2007) described an interesting adaptation of FCA named Relational Concept Analysis which takes a input a set of contexts and inter-context relations resulting in a set of lattices whose concepts are linked by relations. The authors use the approach amongst others to restructure class hierarchies in object-oriented software engineering (Huchard et al. 2007). Within the area of design, FCA has been applied to classes and methods (Godin et al. 1998). However, it has only rarely been applied to the earlier stage of conceptual modeling (Tilley et al. 2007).

Traditional FCA theory has an important shortcoming when we want to use it to formally underpin the construction of conceptual models. These models typically contain two types of relationships between classes, i.e. associations[1] between classes and the inheritance relationships between them. Traditional FCA has only one type of ordering on concepts, namely the subconcept-superconcept relation. In this paper, we extend traditional FCA theory with an extra dimension. The aim is to allow concepts to be ordered, fulfilling the requirements of two (or more) types of orderings. We also give some examples and applications in software requirements engineering and modeling. In particular, we show how the traditional subconcept-superconcept relationship can model Existence Dependency relationships (Poelmans et al. 2010a) and how this new dimension we introduce, allows for the integration of inheritance hierarchies. Triadic FCA (Lehmann et al. 1995), we did not find useful since this extended theory does not work with two ordering relationships, but rather imposes conditions whether or not a certain object has an attribute.

The remainder of this paper is composed as follows. In Section 2 we describe how UML associations can be expressed with an FCA lattice. In Section 3 we introduce inheritance lattices. In Section 4 we show how these 2 types of lattices can be merged to a lattice where the concepts obey the requirements imposed by 2 ordering relations instead of one subconcept-superconcept relation. Section 5 describes some applications and Section 6 concludes the paper.

2 Expressing UML associations with an FCA lattice

In (Poelmans et al. 2010a) we showed how special types of UML associations starting from an entity-use case interaction matrix which indicates the use cases in which the entities participate, an FCA lattice can be automatically derived. By letting the specialization object types (inheritance) out of the lattice construction, we obtain an FCA lattice containing the associations that should be in the class model according to the matrix. In contrast to UML, FCA imposes a partial order relation on all objects. In (Poelmans et al. 2010) we show that the set of lines which interconnect the FCA concepts form a lattice that is isomorphic to a MERODE class model. The MERODE methodology entails the notion of existence dependency, which superimposes a lattice structure (not to be confused with inheritance hierarchies) on objects. The concept of existence dependency (ED) is based on the notion of the “life” of an object (Snoeck et al. 1998). The life of an object is the span between the point in time of its creation and the point in time of its end. Existence dependency is defined at two levels: at the level of object types or classes and at the level of object occurrences. The existence

[1] UML identifies a number of supplementary constructs, such as aggregation and association as a class. These constructs can however be considered as special cases of the basic notion of association. They are (as opposed to inheritance) not fundamentally different from the concept of association.

dependency relation is a partial ordering on objects and object types. The life of the existence dependent object cannot start before the life of its master. Similarly, the life of an existence dependent object ends at the latest at the same time that the life of its master ends.

The notion of existence dependency is similar to the notion of weak entity in data modeling as introduced by Chen and the notion of master entity from OOSSADM. In the ER-notation (Chen 1977) we can use the notion of a weak entity to denote an existence dependent object type since the existence of a weak entity depends on the existence of the other entities it is related to by means of a weak relationship (Chen 1977). Existence dependency is equivalent to the notion of a weak relationship that is in addition mandatory for the weak entity type. MERODE requires all objects in the conceptual model to be related through existence dependency relationships only. The class diagram can therefore be represented as an existence dependency graph.

Formal Concept Analysis is a recent mathematical technique that can be used as an unsupervised clustering technique. Objects participating in the same set of events are grouped in concepts. The starting point of the analysis is a table consisting of rows M (i.e. objects), columns F (i.e. attributes) and crosses $T \subseteq M \times F$ (i.e. relationships between objects and attributes). The mathematical structure used to reference such a cross table is called a formal context.

Table 1. Example of a formal context

	ENTER	LEAVE	E_LOAN	PAY_FINE	RETURN	BORROW	CLASSIFY	CR_ITEM	E_ITEM	CR_TITLE	E_TITLE
Member	C	E	M	M	M	M					
Item							M	C	E		
Loan			E	M	M	C					
Title			M	M	M	M	M	M	M	C	E

Table 1 shows a formal context containing Existence Dependency relationships. In the latter, objects are related (i.e. the C, M and E) to a number of use cases (i.e. the attributes); here an object is related to a use case if the object participates in the use case. The C indicates the use case creates an instantiation of the entity type, M means it modifies an instantiation of the entity type, while E means the use case terminates an instantiation of the entity type. A single-valued formal context is obtained by replacing ‘C’, ‘M’ and ‘E’ by ‘X’. Given a formal context, FCA then derives all concepts from this context and orders them according to a subconcept-superconcept relation. This results in a line diagram (a.k.a. lattice).

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 extension and its intension (Ganter 1999, Wille 1982). The extension consists of all objects belonging to the concept, while the intension 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 and use cases. 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 or events that are common to all objects *o* in the set *O* can be identified, written $\sigma(O)$, via:

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

Take for example the set $O \subseteq M$ consisting of objects Member, Title and Loan. This set O of objects is closely connected to a set A consisting of the attributes "borrow", "e_loan", "return" and "pay_fine", being the use case participations shared by the objects in O. That is:

σ({Member, Title, Loan}) = {borrow, e_loan, return, pay_fine}

Conversely, 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 {borrow, renew, return, lose}, we get to the set $O \subseteq M$ consisting of the objects Member, Title and Loan. That is to say:

τ({borrow, e_loan, return, pay_fine}) = {Member, Title, 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[2].

$(O_1, A_1) \subseteq (O_2, A_2) \Leftrightarrow (O_1 \subseteq O_2 \wedge A_2 \subseteq A_1)$

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," "e_loan," "return," "pay_fine," and "borrow" is a subconcept of the concept with intent "e_loan," "return," "pay_fine," and "borrow." With reference to Table 1, the extent of the latter is composed of object types Loan, Member and Title, while the extent of the former is composed of object type Member.

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 1, 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. It displays only concepts that describe objects and is therefore a subpart of the concept lattice. 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 1 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

[2] 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.

described by the attributes “enter”, “leave”, “e_loan”, “return”, “pay_fine” and “borrow”.

Retrieving the extension of a formal concept from a line diagram such as the one in Figure 1 implies collecting all objects on all paths leading down from the corresponding node. In this example, the extension associated with the second node on the right is {Loan, Title, Member}. To retrieve the intension 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 “e_loan,” “return”, “pay_fine” and “borrow.” The top and bottom concepts in the lattice are special. The top concept contains all objects in its extension. 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 third row with extension {Title} is a subconcept of the second node in row 2 with extension {Loan, Member, Title}.

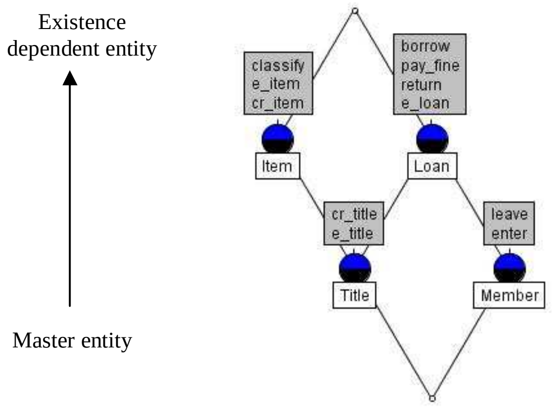

Fig. 1. Line diagram corresponding to the context from Table 1

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. In the next section we will add inheritance relationships to these class diagrams.

3 Expressing inheritance relationships with an FCA lattice

Generalization / specialisation is an abstraction principle that allows to define classes as a refinement of other classes. The more general class is also called supertype or parent class and the refined class is then called a subtype or child class. In this section we will only consider single inheritance. The principle of inheritance says that each object type inherits the features of its eventual parent object type. Such a rule can naturally be expressed using FCA. In our case, the FCA lattice is built based on object-use case participations. Using the principle of substitutability of Barbara Liskov, we can reformulate the rules of inheritance for Use Case participation. The principle of substitutability states that whenever a generalization object occurs it can be replaced by a specialized object, because subtypes are by definition of inheritance type-conformant to their supertype(s). Suppose for example that a Use Case "*register Item*" has been defined, and that Item has the subtypes CD-ROM and Book, then we assume that "*register_Item*" also applies to CD-ROMS and books. If however a *borrow* Use Case has been defined for books, we cannot assume that it also applies to items in general. So, entity-use case interactions of the generalized entity are inherited by the specialized entity, but not conversely. Table 2 shows a formal context containing inheritance relationships. If the inherited use case is applicable to the specialized object type (possibly after overriding the corresponding method), the I/ must be followed by C, M or E to indicate the nature of the involvement. If it is not, the use case is specialized for each subtype, which is indicated by S/ followed by C, M or E.

Table 2. Formal context containing inheritance relationships

	CR_ITEM	CR_CDROM	CR_ISSUE	CR_BORROWABL E_ITEM	CLASSIFY	E_ITEM	E_BORROWABLE _ITEM	E_CDROM	E_ISSUE	CR_VOLUME	E_VOLUME	CR_COPY	E_COPY
Item	C				M	E							
Borrowable Item	S\C			C	I\M	S\E	E						
CDRom	S\C	C			I\M	S\E		E					
Issue	S\C		C		I\M	S\E			E				
Volume	S\C			S\C	I\M	S\E	S\E			C	E		
Copy	S\C			S\C	I\M	S\E	S\E					C	E

The FCA lattice obtained from the subpart of the entity-use case interaction matrix, denoting the inheritance relationship is displayed in Fig. 2. For creating the lattice the many-valued context was turned into a single-valued context by replacing all table entries by 'X'.

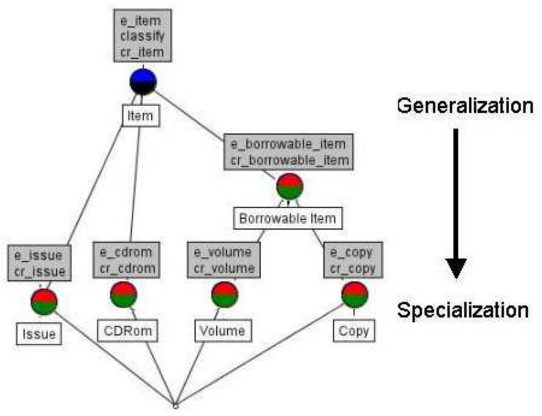

Fig. 2. Inheritance lattice

Object types lower in the lattice are specializations of object types higher in the lattice. The inheritance relationships between object and event types can be expressed using an FCA lattice. In (Snoeck et al. 1996), the generalization-specialization hierarchy is defined as a tree (without multiple inheritance) and specialization types inherit the features of their subtype. This single inheritance hierarchy can be defined as a special type of lattice $\underline{K}$ (*O*, *A*, *I*) with the extra constraint that each concept *S* has at most one superconcept *G* (with the exception of the infinum of the lattice, which must have an empty extent):

$$\forall G, G', S \in \underline{K}(O, A, I) : G <_I S \text{ and } G' <_I S$$
$$\Rightarrow G = G'$$

G, *G'* and *S* are concepts. G is called a super concept of S provided ext (*G*) $\subset$ ext (*S*) and int (*S*) $\subset$ int (*G*). This is denoted as $G <_I S$, Then, (*K* (*O*, *A*, *I*), $<_I$) is a partially ordered set that is denoted by $\underline{K}$(*O*,*A*,*I*) and is called the concept lattice of the context.

To obtain the set of parent types of an object type *P*, we take the extent of the concept that owns *P* (which is called the concept *L*)

$\gamma_+(P) = ext(L)$

To obtain the set of descendant types of a type *P*, we take the extent of all subconcepts of the concept that owns *P* (namely *L*):

$\gamma_-(P) = \{ext(C \in \underline{K}(O, A, I)) \mid C <_I L\}$

To obtain the topmost parent type of a type P, we take the extent of the supremum concept of the lattice.

$\gamma_{\max}(P) = ext(\sup(\underline{K}(O, A, I))$

4 Merging the existence dependency and inheritance lattices

An extended lattice is defined as a combination of a main lattice, inheritance lattices and sublattices. In this extended lattice, the concepts are ordered according to two ordering relations. The vertical dimension is used for the existence dependency relationships between the concepts. The existentially dependent object type will always be above its master in the lattice. The horizontal dimension is used to model the inheritance relationships between the concepts. The specialized object will always be to the right of its generalization in the merged lattice. Both orderings are based on the subconcept-superconcept relationship from traditional FCA but have a different semantic meaning.

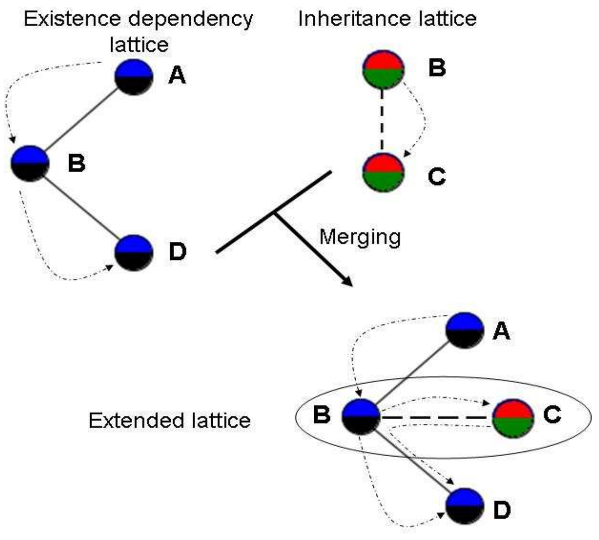

Fig. 3. How the existing dependency lattice and the inheritance lattice are combined to form one extended lattice.

There will always be at least one object type owned by a concept of an inheritance lattice which belongs to the extent of a concept from the main lattice. When merging the existence dependency and inheritance lattices, we need to distinguish between the concepts from the ED and inheritance lattices. We make use of the notion of specialized concept. Note that we do not show the infinum of the inheritance lattices in the extended lattice. This concept is not useful in this section because we only consider single inheritance, and can thus be omitted from the visualization. Figure 3 visually represents the process of merging existence dependency and inheritance lattices. Figure 4 explains the structural properties of the result of this process.

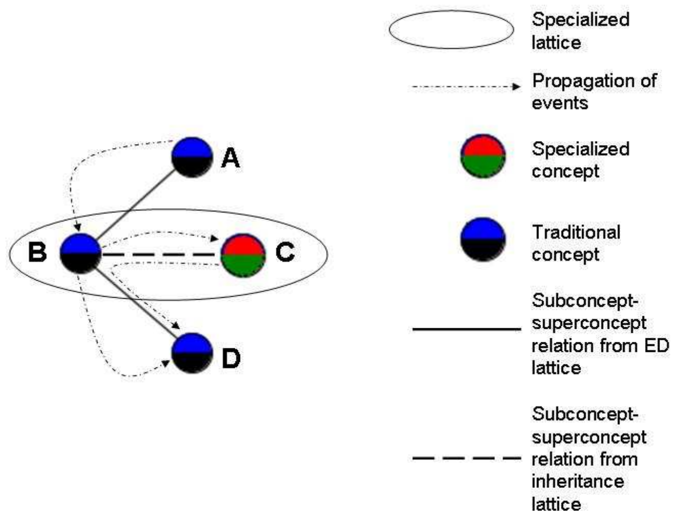
A
B
C
D
Specialized lattice
Propagation of events
Specialized concept
Traditional concept
Subconcept-superconcept relation from ED lattice
Subconcept-superconcept relation from inheritance lattice

Fig. 4. Properties of the extended lattice

The relationships among the concepts of these lattices after propagation of use case participations through the dashed arrows are as follows:

1. int (B) $\subset$ int (C) and ext (C) $\subset$ ext (B)
2. int (C) $\subset$ int (D) and ext (D) $\subset$ ext (C)
3. int (B) $\subset$ int (D) and ext (D) $\subset$ ext (B)
4. int (A) $\subset$ int (B) and ext (B) $\subset$ ext (A)
5. int (A) $\subset$ int (C) and ext (C) $\subset$ ext (A)
6. int (A) $\subset$ int (D) and ext (D) $\subset$ ext (A)

The propagation of use case participations from *A* to *B* and *B* to *D* is required because of ED, as explained in Section 2. The propagation of A to D is the consequence of the transitive ED from A to D via B. This explains properties 3, 4 and 6. The propagation from *B* to *C* is required because of inheritance as explained in Section 3. This explains property 1.

The combination of the existence dependency between A and B and the inheritance relationship between B and C, allows us to naturally infer a relationship between A and C: C inherits from B the fact that it has a relationship with C. Moreover as A's attributes are a subset of B's attributes, the attributes of A are a subset of those of C, thereby implying a similar superset subset relation relationship as between B and C. This explains property 5.

Property 2 needs some more explanation. C inherits from B the fact that it is ED from D. However, D has more attributes than B, so the additional attributes of D are not naturally inherited by C. In the opposite direction, we know that int(B) $\subset$ int(C), however, C has more attributes than B. We will now discuss why we postulate that int(C) $\subset$ int (D). Although the Liskov principle of substitution states that in a relationship between D and B, a B-object can be replaced by a C-object, this can only be done in so far the C-object is capable of adhering to the same attributes as B. The extensions of C are not visible, nor known by D. The reason why the non-trivial property 2 was introduced is to include the benefits of the accumulation rule as introduced by MERODE in the FCA software model (thoroughly discussed in section 5).

We can see that if this is done, concepts *B* and *C* have an identical relationship to the concepts *A* and *D* from the main lattice. We can say that from the point of view of *A* and *D*, the inheritance constellation consisting of concepts *B* and *C* is seen as one "expanded node". This materializes the Liskov principle of substitutability: from the perspective of *A* and *D*, *B* can be substituted by *C* at any time. Propagation of use cases is performed through the dashed arrows. The propagation of use case participations from *C* to *D* is necessary to fulfill the requirements of the OO principle of substitutability. An object type acquires the use case participations of its existence dependent object types through propagation plus all the use case participations of the object types that are directly or indirectly a specialization of any of the existence dependent object types (discussed in more detail in Section 5). On an implementation level these propagations can also be interpreted as follows. The dotted arrows indicate the possibility for unique navigation from one object type to another. This lattice structure is in accordance with the principle of substitutability from OO.

4.1 Properties of extended lattice

There is at least one concept in the inheritance lattice with the same extent and intent as a concept in the ED lattice. Each object type that is owned by concept in the inheritance lattice is also owned by the corresponding concept from the ED lattice. In the extended lattice, these 2 concepts are displayed as one concept. They form the hinges between the ED lattice and inheritance lattice. In other words, a concept in the inheritance lattice is also a part of the ED lattice. This concept is the link between the two lattices.

Proposition 1.
Given: $(\beta(O,A,I),<)$ and $\{(K(X,Y,S),<_I)_1,...,(K(X,Y,S),<_I)_n\}$
Proposition (required):
$\forall n \in \mathrm{N} : \exists L \in (\beta(O,A,I),<) : (\exists E \in (K(X,Y,S),<_I)_n) :$
$ext(L) = ext(E) \wedge \mathrm{int}(L) = \mathrm{int}(E)$

The vertical ordering of concepts according to the ED subconcept / superconcept relation is not violated by introducing inheritance lattices that are attached to the main lattice. The specialized concepts of a concept *L* of a main lattice have the same subconcept-superconcept relations with concepts from the main lattice as *C*. For each specialized concept in an inheritance lattice, the following relationships with the main ED are valid:

Proposition 2.
Given:
$(\beta(O,A,I),<),(K(X,Y,S),<_I)$ and concept *L* as defined in proposition 1
Proposition (required):

$\forall T \in (\beta(O, A, I), <)$:

- $(T > L) \Leftrightarrow (\text{int}(T) \subset \text{int}(L))$ and $(ext(L) \subset ext(T))$

  and $(\forall W <_I L : \text{int}(T) \subset \text{int}(W)$ and $ext(W) \subset ext(T))$

- $(T < L) \Leftrightarrow (\text{int}(L) \subset \text{int}(T))$ and $(ext(T) \subset ext(L))$

  and $(\forall W <_I L : \text{int}(W) \subset \text{int}(T)$ and $ext(T) \subset ext(W))$

5 Applications and accumulation rule

The accumulation rule was introduced to reveal the implicit inheritance relationships in a model, hence allowing verifying the correctness of a software model consisting of 2 ordering relations on concepts. The rule allows for the detection of inherited existence dependency relationships that are not explicitly modelled. This helps in identifying places in the model where such implicit links are erroneous and hence should be removed by removing existence dependency or inheritance relationships or by refactoring the model. The accumulation principle says that an object type not only acquires the use case participations of its existence dependent object types through propagation but also all the use case participations of the object types that are directly or indirectly a specialization of any of the existence dependent object types. It should be noted that this propagation of use case participations through the accumulation principle results in an implicit participation of the master in the use cases of the specialization of its dependent.

Intuitively, conceptual modelers tend to assume covariance, which allows us to define more stringent preconditions for specializations than for generalized object types. Covariance is however almost nowhere supported for design and coding purposes. Contravariance is most used by software designers and states that the properties of a contract or association between the generalized object types cannot be made more stringent at the specialized object type level, only loosened. The specialized object types inherit the properties of the contracts of their supertype and the properties of the associations their supertypes participate in. In Fig. 6 the accumulation principle is used to detect the erroneous link between Borrowable Item and Loan, and in Fig. 7 between Product and Sale. When we observe the structure from Figure 5 in a conceptual domain model expressed as an FCA lattice, the model contains an anomaly.

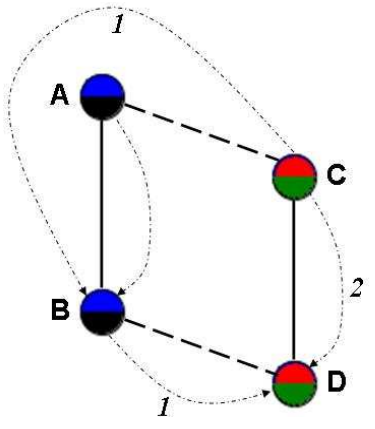

Fig. 5. Detecting a violation of the accumulation rule with the extended lattice

The model contains two links from the dependent Object type *C* to its master *D*. Events are propagated from *A* to *B* by two paths, namely 1) and 2), while one of them is superfluous, In case such an anomaly is observed, there are two options. Either one deletes the ED link between the generalization object types *A* and *B* or one deletes the ED link between the specialization types *C* and *D*. Option 1 is preferred when there is an object type *E* that is a specialization of *B* and that should not inherit the (implicit) ED relationship with *C* from *B*. This is illustrated using the following two real life examples (taken from (Snoeck 1999)).

Example 1:

Our first example comes from a library environment. In a library, different types of items are available: CD-ROMS, single issues of journals, bound volumes of journals, and copies of books. CD-ROMS and single issues of journals must not leave the library. The other types of items can be borrowed (and returned). Only loans of copies can be renewed. A first extended lattice for this example is given in Fig. 6. The feature-object table is given in Table 3 and for readability purposes, we have placed the entities in the columns and the use cases in the rows.

Table 3. Formal context of library environment

	TITLE	ITEM						LOAN		MEMBER
			CDROM	ISSUE	BORROWABLE ITEM				RENEWABLE LOAN	
						VOLUME	COPY			
cr_title	C									
e_title	E									
classify	M	M	I/M	I/M	I/M	I/M	I/M			

cr_item	M	C	S/C	S/C	S/C	S/C	S/C			
cr_CDrom	M		C							
cr_issue	M			C						
cr_bo_item	M				C	S/C	S/C			
cr_volume	M					C				
cr_copy	M						C			
e_item	M	E	S/E	S/E	S/E	S/E	S/E			
e_CDrom	M		E							
e_issue	M			E						
e_bo_item	M				E	S/E	S/E			
e_volume	M					E				
e_copy	M						E			
borrow	M				M			C	S/C	M
cr_re_loan	M				M		M		C	M
renew	M				M		M		M	M
return	M				M		M	M	I/M	M
pay_fine	M				M		M	M	I/M	M
e_loan	M				M		M	E	I/E	M
enter										C
leave										E

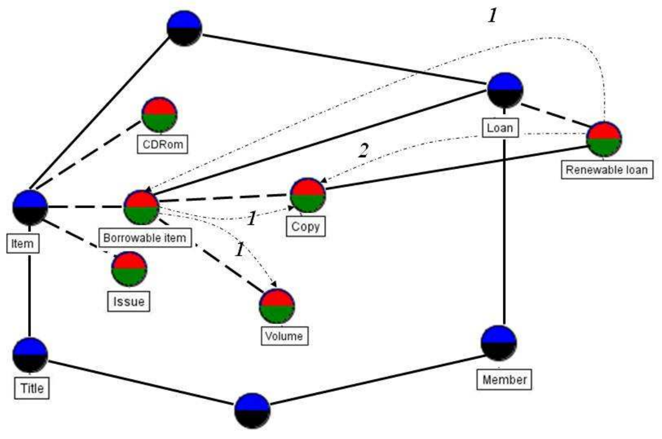

Fig. 6. Detecting anomalies with the extended lattice of a library system

Notice that in Fig. 6 because of the accumulation principle, the renew event was also propagated from Renewable loan to Borrowable item. If inheritance is applied again, Volume will now inherit the renew event from Borrowable item. But this is not what we specified: loans of volumes cannot be renewed. The error stems from the erroneous existence dependency relation between Borrowable item and Loan. A generalization/ specialization hierarchy must always be interpreted as “a generalization object or a specialisation1 object or a specialisation2 object or...”. The erroneous existence dependency relation therefore means: "a loan and by inheritance a renewable loan are existence dependent on a volume or on a Copy.” But a renewable loan cannot be existence dependent on a Volume. The library schema must be corrected by removing the ED relationship between Borrowable item and loan and by adding an entity being a specialisation of Loan named “not renewable loan” with an ED relationship to Volume.

Example 2:

In a pharmacy, some products can be sold freely, others only under doctor's perscription. This is represented in the extended lattice of Fig. 7.

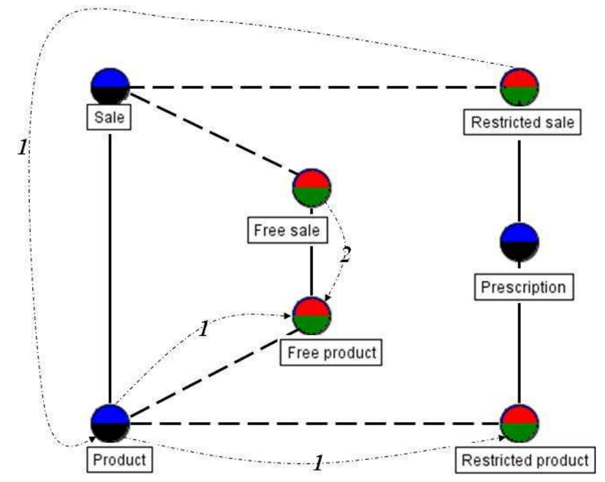

Fig. 7. Detecting anomalies using the extended lattice of a pharmacy system.

The original model contained an existence dependency relation between product and sale since the general case is that a sale is always existence dependent on a product. But again, this dotted existence dependency relation must be interpreted as: "a sale, and by inheritance a free sale and a restricted sale are existence dependent on a product, which is a free product OR a restricted product." Since a restricted sale is existence dependent on a restricted product only (by the intermediary of a prescription), the existence dependency relation between product and sale is wrong and should be removed.

6 Conclusions

In this paper we have shown how traditional FCA theory can be extended with a second ordering relation on concepts. Two partial orderings, one in the horizontal direction and one in the vertical direction increase the expressiveness of the theory thereby making it usable in situations where it was previously not. We particularly showcased the possibilities of the extended theory for software engineering activities. Both inheritance and the traditional ordering based on existence dependency can now be modeled and mathematically underpinned with FCA.

7 References

1. Bain, M. (2003) Inductive Construction of Ontologies from Formal Concept Analysis. Proc. of AI, LNAI 2903, 88-99.
2. Chen, P.P. (1977) The entity relationship approach to logical database design. QED information sciences, Wellesley (Mass.).
3. Cimiano, P., Hotho, A., Stumme, G., Tane, J. (2004) Conceptual Knowledge Processing with Formal Concept Analysis and Ontologies. P. Eklund (Ed.): ICFCA, LNAI 2961, 189-207. Springer.
4. Düwel, S. (1999) Enhancing system analysis by means of formal concept analysis. Proc. conference of Advanced information systems engineering 6th doctoral consortium, Heidelberg, Germany.
5. Düwel, S., Hesse, W. (1998) Identifying candidate objects during system analysis. Proc. CAiSE'98/IFIP 8.1 3rd Int. Workshop on Evaluation of Modeling Methods in System Analysis and Design (EMMSAD).
6. Ganter, B., Wille, R. (1999) Formal Concept Analysis. Mathematical foundations. Springer.
7. Godin, R., Mili, H., Mineau, G.W., Missaoui, R., Arfi, A., Chau, T.-T. (1998) Design of class hierarchies based on concept (Galois) lattices. Theory and Application of Object Systems (RAPOS), 4(2), 117-134.
8. Huchard, M., Rouane Hacene, M, C. Roume, and P. Valtchev. Relational concept discovery in structured datasets. Annals of Mathematics and Artificial Intelligence, 49(1-4):39–76, 2007.
9. Lehmann, F., Wille, R. (1995) A triadic approach to formal concept analysis. In: G. Ellis, R. Levinson, W. Rich, J. F. Sowa (eds). Conceptual structures: Applications, Implementation and theory, Vol. 954 LNAI, Springer-Verlag.
10. Mens, K., Tourwé, T. (2005) Delving source code with formal concept analysis. Computer Languages, Systems & Stuctures, 31, 183-197. Elsevier.
11. Poelmans, J., Dedene, G., Snoeck, M., Viaene, S. (2010a) Using Formal Concept Analysis for the verification of process, data matrices in conceptual domain models. IASTED International Conference on Software Engineering, Innsbruck, Austria, 17-19 february.
12. Poelmans, J., Elzinga, P., Viaene, S., Dedene, G. (2010b), Formal Concept Analysis in Knowledge Discovery: a Survey. LNCS, 6208, 139-153, 18th international conference on conceptual structures (ICCS): from information to intelligence. 26 - 30 July, Kuching, Sarawak, Malaysia**.** Springer.
13. Richards, D., Boettger, K. (2002) A controlled language to assist conversion of use case descriptions into concept lattices. Proc. of 15th Australian joint conference on artificial intelligence, LNAI, vol. 2557, 1-11.
14. Richards, D., Boettger, K., Fure, A. (2002) Using RECOCASE to compare use cases from multiple viewpoints. Proc. of the 13th Australian Conference on Information Systems ACIS, Melbourne.
15. Rouane-Hacene, M. , M. Huchard, A. Napoli, and P. Valtchev. A proposal for combining formal concept analysis and description logics for mining relational data. In S. Schmidt S. Kuznetsov, editor, Intl. Conf. on Formal Concept Analysis (ICFCA'2007), volume 4390 of LNCS, pages 51–65, Clermont Ferrand, France, 2007. Springer-Verlag.
16. Snelting, G., Tip, F. (1998) Reengineering class hierarchies using concept analysis. Proc. of ACMSIGSOFT Symposium on the Foundations of Software Engineering, 99-110.
17. Snelting, G., Tip, F. (2000) Understanding class hierarchies using concept analysis. ACM Transactions on programming languages and systems, 540-582, May.
18. Snoeck, M., Dedene, G. (1998) Existence dependency. The key to semantic integrity between structural and behavioral aspects of objects types. IEEE Transactions on Software Engineering, 24(4), 233-251.
19. Tilley, T., Cole, R., Becker, P., Eklund, P. (2005) A survey of Formal Concept Analysis Support for Software engineering activities. G. Stumme (Ed.) ICFCA, LNCS 3626, 250-271, Springer.

20. Tilley, T., Eklund, P. (2007) Citation analysis using Formal Concept Analysis: A case study in software engineering. 18th int. conf. on database and expert systems applications (DEXA).
21. Tonella, P., Antoniol, G. (1999) Object-oriented design pattern inference. Proc. of CSM, 230-240.
22. Wille, R. (1982). Restructuring lattice theory: an approach based on hierarchies of concepts. I. Rival (Ed.): Ordered sets, 445-470. Reidel. Dordrecht-Boston.

Ackowledgements

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