Lessons Learned in Applying Formal Concept Analysis

Gabriela Arévalo, Stéphane Ducasse, Oscar Nierstrasz

Lessons Learned in Applying Formal Concept Analysis to Reverse Engineering *

Gabriela Arévalo, Stéphane Ducasse and Oscar Nierstrasz
Software Composition Group
University of Bern, Switzerland
www.iam.unibe.ch/∼scg

Abstract

A key difficulty in the maintenance and evolution of complex software systems is to recognize and understand the implicit dependencies that define contracts that must be respected by changes to the software. Formal Concept Analysis is a well-established technique for identifying groups of elements with common sets of properties. We have successfully applied FCA to complex software systems in order to automatically discover a variety of different kinds of implicit, recurring sets of dependencies amongst design artifacts. In this paper we describe our approach, outline three case studies, and draw various lessons from our experiences. In particular, we discuss how our approach is applied iteratively in order to draw the maximum benefit offered by FCA.

1 Introduction

One of the key difficulties faced by developers who must maintain and extend complex software systems, is to identify the *implicit dependencies* in the system. This problem is particularly onerous in object-oriented systems, where mechanisms such as dynamic binding, inheritance and polymorphism may obscure the presence of dependencies [DRW00, Dek03]. In many cases, these dependencies arise due to the application of well-known programming idioms, coding conventions, architectural constraints and design patterns [SG95], though sometimes they may be a sign of weak programming practices.

On the one hand, it is difficult to identify these dependencies in non-trivial applications because system documentation tends to be inadequate or out-of-date and because the information we seek is not explicit in the code [DDN02, SLMD96, LRP95]. On the other hand, these dependencies play a part in implicit contracts between the various software artifacts of the system. A developer making changes or extensions to an object-oriented system must therefore understand the dependencies among the classes or risk that seemingly innocuous changes break the implicit contracts they play a part in [SLMD96]. In short, implicit,

*In Proceedings of ICFCA 2005 (3^{rd} International Conference on Formal Concept Analysis) pp.95-112, Springer LNAI series (LNAI 3403), Springer Verlag

undocumented dependencies lead to *fragile systems* that are difficult to extend or modify correctly.

Due to the complexity and size of present-day software systems, it is clear that a software engineer would benefit from a (semi-)automatic tool to help cope with these problems.

Formal Concept Analysis provides a formal framework for recognizing groups of elements that exhibit common properties. It is natural to ask whether FCA can be applied to the problem of recognizing implicit, recurring dependencies and other design artifacts in complex software systems.

From our viewpoint, FCA is a metatool that we use as tool builders to build *new* software engineering tools to analyze the software. The *software engineer* is considered to be the end user for our approaches, but his knowledge is needed to evaluate whether the results provided by the approaches are meaningful or not.

Over the past four years, we have developed an approach based on FCA to detect undocumented dependencies by modeling them as recurring sets of properties over various kinds of software entities. We have successfully applied this approach to a variety of different reverse engineering problems at different levels of abstraction. At the level of classes, FCA helps us to characterize how the methods are accessing state and how the methods commonly collaborate inside the class [ADN03]. At the level of the class hierarchy, FCA helps us to identify typical calling relationships between classes and subclasses in the presence of late binding and overriding and superclass reuse [Aré03]. Finally, at the application level, we have used FCA to detect recurring collaboration patterns and programming idioms [ABN04]. We obtain, as a consequence, views of the software at a higher level of abstraction than the code. These high level views support *opportunistic* understanding [LPLS96] in which a software engineer gains insight into a piece of software by iteratively exploring the views and reading code.

In this paper we summarize our approach and the issues that must be taken into consideration to apply FCA to software artifacts, we briefly outline our three case studies, and we conclude by evaluating the advantages and drawbacks of using FCA as a metatool for our reverse engineering approaches.

2 Overview of the Approach

In this section we describe a general approach to use FCA to build tools that identify recurring sets of dependencies in the context of object-oriented software reengineering. Our approach conforms to a pipeline architecture in which the analysis is carried out by a sequence of processing steps. The output of each step provides the input to the next step. We have implemented the approach as an extension of the *Moose* reengineering environment [DLT00].

The processing steps are illustrated in Figure 1. We can briefly summarize the goal of each step as follows:

– *Model Import:* A model of the software is constructed from the source code.

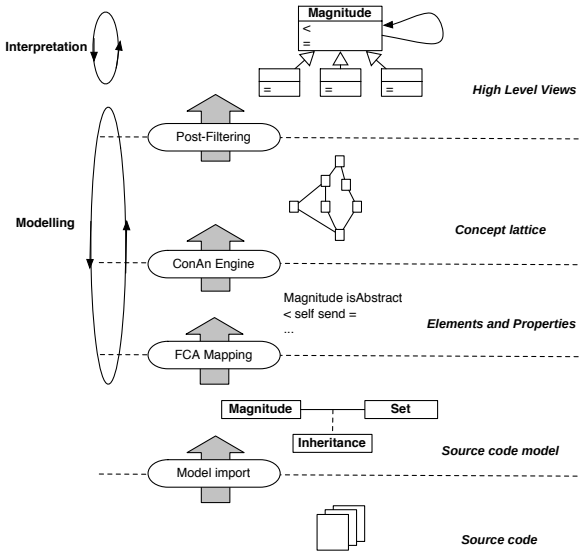

Figure 1: The overall approach

- *FCA Mapping:* A FCA Context (Elements, Properties, Incidence Table) is built, mapping from metamodel entities to FCA elements (referred as *objects* in FCA literature) and properties (referred as *attributes* in FCA literature) [1].
- *ConAn Engine:* The concepts and the lattice are generated by the ConAn tool.
- *Post-Filtering:* Concepts that are not useful for the analysis are filtered out.
- *Analysis:* The concepts are used to build the high level views.

A key aspect of our approach is that one must iterate over the modeling and the interpretation phases (see Figure 1). The *modeling* phase entails a process of experimentation with smaller case studies to find a suitable mapping from the source code model to FCA elements and properties. A particular challenge is to find a mapping that is efficient in terms of identifying meaningful concepts while minimizing the quantity of data that must be processed.

The *interpretation* phase is the other iterative process in which the output of the modeling phase is analyzed in order to interpret the resulting concepts in the context of the application domain. The useful concepts can then be flagged so that future occurrences can be automatically detected. As more case studies are analyzed, the set of identifiably useful concepts typically increases up to a certain point, and then stabilizes.

[1] We prefer to use the terms *element* and *property* instead of the terms *object* and *attribute* in this paper because the terms *object* and *attribute* have a very specific meaning in the object oriented programming paradigm.

We must remark that, from our experiences, there are two main participants in the approach: the *tool builder* and *the software engineer*. The *tool builder* builds the FCA-based tool to generate the high level views, and the *software engineer* uses the results provided by the tool to analyze a piece of software. Both of them work together in the *modeling* and *interpretation* phase of the approach, because *software engineer* has the knowledge of analyzing a system and the *tool builder* can represent this knowledge in the tool.

We will now describe each processing step in detail and outline the key issues that must be addressed in order to apply the approach.

Model Import

Description: Our first step is to build a *model* of the application from the source code. For this purpose we use the *Moose* reengineering platform, a research vehicle for reverse and reengineering object-oriented software [DDN02]. Software models in *Moose* conform to the FAMIX metamodel [TDDN00], a language-independent metamodel for reengineering. *Moose* functions both as a software repository for these models, and as a platform for implementing language-independent tools that query and manipulate the models.

Issues: In this step the most important issue is how to map the source code to metamodel entities. The main goal of this step is to have a language-independent representation of the software. In our specific case, we use the FAMIX metamodel, which includes critical information such as method invocations and attribute accesses. The tool builder can, however, choose any suitable metamodel.

FCA Mapping

Description: In the second step, we need to map the model entities to *elements* and *properties*, and we need to produce an *incidence table* that records which elements fullfil which property. The choice of elements and properties depends on the view we want to obtain.

Issues: This is a critical step because several issues must be considered. Each of these issues is part of the iterative *modeling* process.

- *Choice of Elements:* First we must decide which metamodel entities are mapped to FCA elements. This is normally straightforward. In most cases there are some particular metamodel entities that are directly adopted as FCA elements (*e.g.*, classes, methods). Alternatively, a FCA element may correspond to a set of entities (*e.g.*, a set of collaborating classes).
- *Compact Representation of Data:* In some cases, a naïve mapping from metamodel entities to FCA elements may result in the generation of massive amounts of data. In such cases, it may well be that many elements are in fact redundant. For example, if method invocations are chosen as FCA elements, it may be that multiple invocations of the same method do not add any new information for the purpose of applying FCA.

By taking care in how FCA elements are generated from the metamodel, we can not only reduce noise, but we can also reduce the cost of computing the concepts.

- *Choice of Properties:* Once the issue of modeling FCA elements is decided, the next step is to choose suitable properties. Well-chosen properties achieve the goal of distinguishing groups of similar elements. This means that they should neither be too general (so that most elements fulfill them) nor be too specific (so only few elements fulfill them).
- *Use of negative properties:* Nevertheless, in some cases the developer needs still more properties to distinguish the elements. But simply adding more properties may only increase the complexity of the approach. The use of "negative" information (built by negating existing properties) may help.
- *Single or Multiple FCA Contexts:* In some cases, multiple FCA contexts may be required. For example, in the XRay views case study, one context was used to analyze state and another to analyze behavior.
- *Computation of properties or elements:* When building the FCA context of a system to analyze, there are two alternatives for the FCA mapping. In an *one-to-one* mapping, the developer directly adopts metamodel entities and metamodel relationships as FCA elements and properties respectively. In a *many-to-one* the developer builds more complex FCA elements and properties by computing them from the metamodel entities and relationships, meaning for example that an FCA element can be composed of several metamodel entities, or an FCA property must be calculated based on metamodel relationships between different entities. This issue is one of the bottlenecks in the total computation time of the approach, because the incidence table must be computed in this step and if the FCA property must be calculated, this time can also compromise the total computation time.

ConAn Engine

Description: Once the elements and properties are defined, we run the ConAn engine. The ConAn engine is a tool implemented in VisualWorks 7 which runs the FCA algorithms to build the concepts and the lattice. ConAn applies the Ganter algorithm [GW99] to build the concepts and our own algorithm to build the lattice [Aré05].

Issues: In this step, there are two main issues to consider

- *Performance of Ganter algorithm:* Given an FCA Context $C = (E, P, I)$, the Ganter algorithm has a time complexity of $O(|E|^2|P|)$. This is the second bottleneck of the approach because in our case studies the number of FCA elements is large due to the size of the applications. We consider that |P| is not a critical factor because in our cases studies the maximum number of properties is 15.
- *Performance of Lattice Building Algorithm:* Our algorithm is the simplest algorithm to build the lattice but the time complexity is $O(n^2)$ where n is the number of concepts calculated by Ganter algorithm. This is the last bottleneck of the approach.
- *Unnecessary properties:* It may happen that certain properties are not held by any element. Such properties just increase noise, since they will percolate to the bottom

concept of the lattice, where we have no elements and all properties of the context.

Post-Filtering

Description: Once the concepts and the lattice are built, each concept constitutes a potential *candidate* for analysis. But not all the concepts are relevant. Thus we have a *post-filtering* process, which is the last step performed by the tool. In this way we filter out meaningless concepts.

Issues: In this step, there are two main issues to consider:

- *Removal of Top and Bottom Concepts:* The first step in *post-filtering* is to remove the *top* and *bottom* concepts. Neither provides useful information for our analysis because each one usually contains an empty set. (The intent is empty in the top concept and the extent is empty in the bottom concept).
- *Removal of meaningless concepts:* This step depends on the interpretation we give to the concepts. Usually concepts with only a single element or property are candidates for removal because the interesting characteristic of the approach is to find *groups* of elements sharing *common* characteristics. Concepts with only a single element occur typically in nodes next to the bottom of the lattice, whereas concepts with only one property are usually next to the top of the lattice.

Analysis

Description: In this step, the software engineer examines the candidate concepts resulting from the previous steps and uses them to explore the different *implicit* dependencies between the software entities and how they determine or affect the behavior of the system

Issues: In this step, there are several issues to consider. All of them are related to how the software engineer interprets the concepts to get meaningful or useful results.

- *Concept Interpretation based on Elements or Properties:* Once the lattice is calculated, we can interpret each concept $C = (\{E_1 \dots E_n\}, \{P_1 \dots P_m\})$ using either its elements or its properties. If we use the properties, we try to associate a meaning to the conjunction of the properties. On the other hand, if we focus on the elements, we essentially discard the properties and instead search for a domain specific association between the elements (for example, classes being related by inheritance).
- *Equivalent Concepts:* When we interpret the concepts based on their properties, we can find that the meaning of several concepts can be the same. This means that for our analysis, the same meaning can be associated to different sets of properties.
- *Automated Concept Interpretation:* The interpretation of concepts can be transformed into an automatic process. Once the software engineer establishes a meaning for a given concept, this correspondence can be stored in a database. The next time the analysis is performed on another case study, the established interpretations can be retrieved and automatically applied to the concepts identified. Once this process is

finished, the software engineer must still check those concepts whose meaning has not been identified automatically.

- *Using Partial Order in the Lattice:* The concepts in the lattice are related by a partial order. During analysis, the software engineer should evaluate if it is possible to interpret the partial order of the lattice in terms of software relationships. This means that the software engineer should only not interpret the concepts but also the relationships between them.
- *Limit of using FCA as a grouping technique:* When additional case studies fail to reveal new meaningful concepts, then the application of FCA has reached its limit. At this point, the set of recognized concepts and their interpretations can be encoded in a fixed way, for example, as logical predicates over the model entities, thus fully automating the recognition process and bypassing the use of FCA.

3 High Level Views in Reverse Engineering

Using the approach we have introduced in the previous section, we have generated different views at different abstraction levels for object oriented applications. We present each analysis according to a simple pattern: *Reengineering Goal*, *FCA Mapping*, *High Level Views*, *Validation*, *Concrete Examples* and *Issues*. *Reengineering Goal* introduces the analysis goal of the corresponding high level view. *FCA Mapping* explains which are the chosen elements and properties of the context for the analysis. *High level view* presents how the concepts are read to interpret which of them are meaningful or not for our analysis and how we use them to build the corresponding high level views. *Validation* summarizes some results of the case studies. *Concrete Examples* presents some examples found in the specific case study. *Issues* summarizes briefly which were the main issues taken into account to build the high level view.

3.1 Understanding a Class: XRay Views

Reengineering Goal: The goal is to understand the internal workings of a class by capturing how methods call each other and how they collaborate in accessing the state. We focus on analyzing a class as an isolated development unit [ADN03].

FCA Mapping: The elements are the *methods* (labelled m or n) and the *attributes* (labelled a) of the class, and the properties, are:

- m reads or writes the value of a
- m calls via-self n in its body
- m is abstract in its class
- m is concrete in its class
- m is an "interface" (not called in any method defined in the same class)
- m is "stateless" (doesn't read or write any attribute defined in the class)

High Level Views: A XRay view is a combination of concepts that exposes specific aspects of a class. We have defined three XRay views: State Usage, External/Internal Calls and Behavioural Skeleton. These three views address different but logically related aspects of the behavior of a class.

State Usage focuses on how the behavior accesses the state of the class, and what dependencies exist between groups of methods and attributes. This view helps us to measure the class cohesion [BDW98] revealing whether there are methods using the state partially or totally and whether there are attributes working together to provide different functionalities of the class.

Behavioural Skeleton focuses on methods according to whether or not they work together with other methods defined in the class or whether or not they access the state of the class. The way methods form groups of methods that work together also indicates how cohesive the class is [BDW98].

External/Internal Calls focuses on methods according to their participation in internal or external invocations. Thus, this view reveals the overall shape of the class in terms of its internal reuse of functionality. This is important for understanding framework classes that subclasses will extend. Interface methods, for example, are often generic template methods, and internal methods are often hook methods that should be overridden or extended by subclasses.

Validation: We have applied the XRay views to three Smalltalk classes: *OrderedCollection* (2 attributes, 54 methods), *Scanner* (10 attributes, 24 methods) and *UIBuilder* (18 attributes, 122 methods). We chose these particular three classes because they are different enough in terms of size and functionality and they address well-known domains. In general terms, we discovered that in *OrderedCollection* most of the methods access all the state of the class, that there is little local behavior to be reused or extended by subclasses because the class works with inherited methods, and there are few collaborations among the methods. *UIBuilder* is a class where most of the methods are not invoked in the class itself, meaning the internal behavior is minimal, and we have a large *interface*. This class offers a lot of functionality to build complex user interface and also several ways to query its internal state. In *Scanner*, the collaboration between methods occurs in pairs and there are no groups of methods collaborating with other groups.

Concrete Examples: We illustrate this case study with the XRay State Usage found in the Smalltalk class *OrderedCollection*. This view is composed of several concepts, and it clusters attributes and methods according to the way methods access the attributes. The motivation for this view is that, in order to understand the design of a class, it is important to gain insight into how the behaviour accesses the state, and what dependencies exist between groups of methods and attributes. This view helps us to measure the cohesion of the class [BDW98], thereby revealing any methods that use the state partially or totally and any attributes that work together to provide different functionalities of the class.

Some of the concepts occurring in State Usage are the following:

- {before, removeAtIndex:, add:beforeIndex:, first, removeFirst, removeFirst:, addFirst } *reads or writes* {firstIndex } represents the Exclusive Direct Accessors of firstIndex.

- {after, last, removeIndex:, addLastNoCheck:, removeLast, addLast:, removeLast: } *reads or writes* {lastIndex } represents the Exclusive Direct Accessors of lastIndex
- {makeRoomAtFirst, changeSizeTo:, removeAllSuchThat:, makeRoomAtLast, do:, notEmpty:, keysAndValuesDo:, detect:ifNone:, changeCapacityTo:, isEmpty, size, remove:ifAbsent:, includes:, reverseDo:, find:, setIndices, insert: before:, at:, at:put:, includes: } *reads or writes* {firstIndex, lastIndex } represents the Collaborating Attributes
- Stateful Core Methods = the same set as Collaborating Attributes

Before analysing the concepts identified by this view, we posed the hypothesis that the two attributes maintain an invariant representing a memory zone in the third anonymous attribute. From the analysis we obtain the following points:

- By browsing Exclusive Direct Accessors methods, we confirm that the naming conventions used help the maintainer to understand how the methods work with the instance variables, because we see that the method removeFirst accesses firstIndex and removeLast: accesses lastIndex.
- The numbers of methods that exclusively access each attribute are very similar, however, we discover (by inspecting the code) that firstIndex is mostly accessed by readers, whereas lastIndex is mostly accessed by writers.
- It is worth noting that Collaborating Attributes are accessed by the same methods that are identified as Stateful Core Methods. This situation is not common even for classes with a small number of attributes, and reveals a cohesive collaboration between the attributes when the class is well-designed and gives a specific functionality, in this specific case, dealing with collections.
- We identified 20 out of 56 methods in total that systematically access *all* the state of the class. By further inspection, we learned that most of the accessors are readers. There are only five methods, makeRoomAtFirst, makeRoomAtLast, setIndices, insert:before:, and setIndicesFrom:, that read and write the state at the same time. More than half of the methods (33 over 56) directly and indirectly access both attributes. This confirms the hypothesis that the class maintains a strong correlation between the two attributes and the anonymous attribute of the class.

All these facts confirm the hypothesis that the class maintains a strong correlation between the two attributes and the anonymous attribute of the class.

Issues: We mention some important issues about this approach.

- *Choice of elements and properties.* Elements and properties are mapped directly from the metamodel: the elements are attributes and methods, and the properties are accesses to attributes and calls to methods.
- *Compact representation of data.* Supposing you have two methods m and n and one attribute a, if we have several calls to the method n or accesses to the attribute a in the method body of m, we just keep one representative of n and a related to the method m.

- *Multiple FCA contexts.* In this case, we have used two lattices. The first one is used to analyze the state of the class and the second one is used to analyze the invocations of the class. We did not combine this information in a single lattice because we consider them to be completely different aspects of the class.
- *Unnecessary properties.* In some classes, the following properties *isAbstract*, *isStateless*, *isInterface* are discarded. This is normal because in any given class, it commonly occurs that all methods are concrete, or that all the methods access the state, or that most methods are called inside the class.
- *Meaningless concepts.* We discarded concepts with a single method in the set of elements, because we were more focused on groups of methods (represented in the elements) collaborating with another group of methods (represented in the properties).

3.2 Analyzing Class Hierarchies: Dependency Schemas

Reengineering Goal: Using the state and behavior of a class we analyze the different recurring dependencies between the classes of a hierarchy. They help us to understand which are the common and irregular design decisions taken when the hierarchy was built, and possible refactorings that were carried out [Aré03].

FCA Mapping: The elements are the accesses to any attribute defined in any classes of the hierarchy, and the called methods in any class of the hierarchy. If i is an invoked method or accessed attribute, and C, C_1, C_2 are classes, the properties are grouped as follows:

- Kind of calls: *C invokes i via self* and *C invokes i via super*
- Location of accessed state: *i accesses* { *local state*, *state in Ancestor* C_1, *state in Descendant* C_1 }
- Kind and Location of invoked method: { *is abstract*, *is concrete*, *is cancelled* } × { *locally*, *in ancestor* C_1 *of* C , *in descendant* C_1 *of* C }

High Level Views: A *dependency schema* is a recurring set of dependencies (expressed with the properties of the concepts) over methods and attributes in a class hierarchy. We have identified 16 dependency schemas that are classified as:

Classical schemas representing common idioms/styles that are used to build and extend a class hierarchy.

Bad Smell schemas representing doubtful designs decisions used to build the hierarchy. They are frequently a sign that some parts should be completely changed or even rewritten from scratch.

Irregularity schemas representing irregular situations used to build the hierarchy. Often the implementation can be improved using minimal changes. They are less serious than *bad smell* schemas.

Thus we see that the *dependency schemas* can be a good basis for identifying which parts of a system are in need of repair. These schemas can be used in two different ways: Either we obtain the global view of the system and which kinds of dependencies and practices occur

when analyzing a complete hierarchy, or we get detailed information about how specific classes are related to others in their hierarchy by restricting the analysis to just those classes.

Validation: We have validated *dependency schemas* in three Smalltalk class hierarchies: *Collection* (104 classes distributed in 8 inheritance levels, 2162 methods, 3117 invocations, 1146 accesses to the state of the classes), *Magnitude* and *Model*. *Collection* is an essential part of the Smalltalk system and it makes use of subclassing for different purposes. In this class hierarchy, the most used *classical* schemas are (1) the reuse of superclass behavior, meaning concrete methods that invokes superclass methods by *self* or *super*, (2) local behavior, meaning methods defined and used in the class that are not overridden in the subclasses and (3) local direct access, meaning methods that directly access the class state. In general terms, this means that the classes define their own state and behavior but they exhibit heavy superclass reuse. Within *bad smell* schemas, the most common is the ancestor direct state access meaning methods that directly access the state of an ancestor, bypassing any accessors. This is not a good coding practice since it violates class encapsulation. Within *irregularity* schemas the most common case is that of inherited and local invocations where methods are invoked by both *self* and *super* sends within the same class. This may be a problem if the super sends are invoked from a method with a different name. This is an *irregular* case because the class is overriding the superclass behavior but is indirectly using the superclass behavior.

Concrete Examples: We illustrate this case study with the *schema* named *Broken super send Chain*, which is categorized as a *Bad Smell* schema. It was found in the analysis of the Smalltalk class OrderedCollection.

Within the *"Bad Smell"* category, we have the schema *Broken* super *send Chain* (shown in Figure 2). It is composed of the following elements and properties:

- *C invokes i via super*: {representBinaryOn:, =} are *super-called* in SortedCollection
- *i is concrete locally*: {representBinaryOn:, =} has concrete behavior in SortedCollection.
- *i is concrete in ancestor* C_1 *of* C : {representBinaryOn:, =} has concrete behavior in ancestor SequenceableCollection of SortedCollection.
- *i is concrete in descendant* C_1 *of* C : {representBinaryOn:, =} has concrete behavior in descendant SortedCollectionWithPolicy of SortedCollection.

This schema identifies methods that are extended (*i.e.*, performing a *super* send) in a class but redefined in their subclasses without calling the overridden behavior, thus giving the impression of breaking the original extension logic. In SortedCollection the methods = and representBinaryOn: invoke hidden superclass methods. But the definitions of these methods in the subclass SortedCollectionWithPolicy do not invoke the super methods defined in SortedCollection. Such a behavior can lead to unexpected results when the classes are extended.

Issues: We mention some important issues about this approach.

- *Choice of Elements and Properties.* We map the attribute accesses and method calls directly from the metamodel. The choice of properties requires some analysis, because we need to cover the different possible inheritance relationships of the elements. The

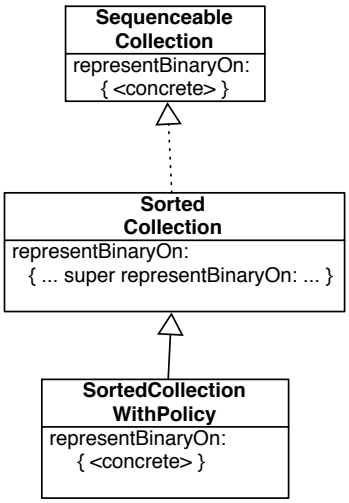

Figure 2: Broken *super* send Chain

- *Compact representation of data.* Supposing you have one method m and one attribute a in a class C. If we have several calls to the method m or accesses to the attribute a in several methods of the class C, we just keep one representative of the call to n and of the access to a related to the class C.
- *Use negative properties.* We define three negative properties because they help us to complement the information of the elements considering the three inheritance relationships used in the approach: *local*, *ancestor* and *descendant* definitions.
- *Meaningless concepts.* All the meaningful concepts must show either a positive or negative information about the 3 relationships: *local*, *ancestor* and *descendant*, and have at least one property of category *kinds of calls* (invokes i *via self* or *via super*). The rest of the concepts are discarded.

properties in the category *Kinds of calls* are mapped directly from the metamodel the rest of the properties are calculated based on the relationships expressed in the metamodel.

- *Single context.* In this case, we just use only one lattice, because we analyze only one aspect of classes: inheritance relationships.

3.3 Collaborations Patterns on Applications

Reengineering Goal: We analyze the different patterns used in a system using structural relationships between classes in a system. We call them *Collaboration Patterns* and they show us not only classical design patterns [GHJV95] but any kind of repeated patterns of *hidden contracts* between the classes, which may represent design patterns, architectural constraints, or simply idioms and conventions adopted for the project. With these patterns, we analyze how the system was built and which are the main constraints it respects [ABN04]. This approach refines and extends that which was proposed by Tonella and Antoniol [TA99] for detecting classical design patterns.

FCA Mapping: The elements are tuples composed of classes from the analyzed application. The order refers to length of the tuples. In our cases, we have experimented with order 2, 3 and 4. The properties are relations inside one class tuple, and are classified as *binary* and *unary* properties. The *binary* property characterizes a relationship between two classes inside the tuple, and the *unary* property gives a characteristic of a single class in the tuple. Given a tuple composed of classes $(C_1, C_2, ...C_n)$, the properties are:

- Binary property: For any i, j, $1 \leq i, j \leq n$, C_i is subclass of C_j, C_i accesses C_j, C_i has as attribute C_j, C_i invokes C_j, C_i uses locally C_j
- Unary property: For any i, $1 \leq n$, C_i is abstract, C_i is root, C_i is singleton, C_i has local defined method.

High Level Views: The concepts are composed of groups of tuples of classes that share common properties. The conjunction of these properties characterizes the pattern that all the tuples in the concept represent. Each *meaningful* concept represents a candidate for a pattern. Based on our case studies using tuples of classes of length 3 and 4, we have identified 8 collaboration patterns. 4 of 8 patterns exhibit the structural relationships of known design patterns (*Facade, Composite, Adapter* and *Bridge*). The rest of the identified patterns called *Subclass Star*, *Subclass Chain*, *Attribute Chain* and *Attribute Star* show the relationships among classes based on inheritance or on invocations, but not combined at the same time. The frequency of the patterns helps the developer to identify coding style applied in the application. The most interesting contribution of this high level view is the possibility of establishing a called *pattern neighbourhoods* over detected patterns. With the *neighbours* of the patterns, we can detect either missing relationships between classes needed to complete a pattern, or excess relationships between classes that extend a *pattern*. We can also analyze the connections of the identified patterns with the classes implemented in the analyzed application.

Validation: We have investigated the *collaboration patterns* in three Smalltalk applications: (1) *ADvance* [2](167 classes, 2719 methods, 14466 lines of code) is a multidimensional OOAD-tool for supporting object-oriented analysis and design, reverse engineering and documentation, (2) *SmallWiki* [3] (100 classes, 1072 methods, 4347 lines of code) is a new and fully object-oriented wiki implementation in Smalltalk and (3) *CodeCrawler* [4] (81 classes, 1077 methods, 4868 lines of code) is a language independent software visualization tool. These three applications have different sizes and complexity. We briefly summarize some results related to coding styles. We have seen that frequency and the presence of a pattern in a system is besides domain specific issues of coding style. In our case studies we have seen that *CodeCrawler* has a lot of *Subclass Star* and *Facade* patterns, whereas *SmallWiki* has a lot of *Attribute Chain* and *Attribute Star* patterns. *Advance* is the only application with the *Composite* pattern.

Concrete Examples: We illustrate this case study showing two *collaboration patterns* named *Attribute Chain* and *Subclass Chain* that were found in the tool *CodeCrawler* (shown in the Figure 4).

Pattern *Subclass Chain* in order = 3 is described with the following properties: C_3 *is subclass of* C_2, $\{C_2$ *is subclass of* C_1, C_1 *is abstract*$\}$, and in order = 4 the property C_4 *is subclass of* C_3 is added. Pattern *Attribute Chain* in order = 3 is described with the following properties: $\{C_1$ *invokes* C_2, C_2 *invokes* C_3 and C_1 *is abstract*$\}$, and in order = 4 the property C_3 *invokes* C_4 is added. Pattern *Composite* (shown in the Figure 3) is described with the following properties: $\{C_2$ *is subclass of*C_1, C_3 *is subclass of* C_1, C_3 *invokes* C_1, C_1 *is abstract*$\}$.

[2]http://www.io.com/ icc/
[3]http://c2.com/cgi/wiki?SmallWiki
[4]http://www.iam.unibe.ch/ scg/Research/CodeCrawler/index.html

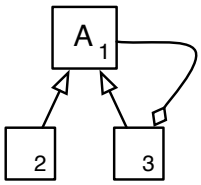

Figure 3: Composite Pattern

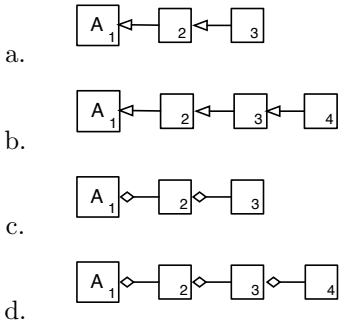

Figure 4: Subclass Chain (a)(b) and Attribute Chain (c)(d) in order = 3 and order = 4. Number in the classes indicate position in the tuple

The first two patterns help us to analyze some coding styles used in *CodeCrawler*. We have found 25 instances of 3 classes (order = 3) of *Attribute Chain* but no instances of 4 classes (order = 4). This means that in this application the developer has used at most 3 levels of delegation between the classes. Thus, if we detect a case of delegation in a method in a class in *CodeCrawler*, we are sure that this delegation will relate 2 or 3 classes. In the case of *Subclass Chain*, we have found 11 instances of 3 classes (order = 3) but only 3 instances of 4 classes (order = 4). This means that in this application most of hierarchies have 3 levels of inheritance and only some of them have 4 levels of inheritance. This is a symptom that the application uses *flat hierarchies* in its implementation. Related to the idea of *pattern neighbourhood*, as we have said exploring the *neighbours* of a pattern is important to detect all candidates for a classical pattern. For example, we have found the *Abstract Composite* pattern of order $o = 3$ of the ADvance application. We have detected two *Abstract Composite* patterns, but in the neighborhood we find four more *Composite* patterns without an abstract composite root.

Issues: We mention some important issues about this approach.

- Choice of elements and properties. Elements are tuples of classes built from the metamodel. As our case study refines the work of Tonella and Antoniol [TA99] we use the same idea to build the elements. The choice of properties is the set of the structural relationships to characterize *Structural Design Patterns* [GHJV95]. Except the properties *is subclass of* and *is abstract* that are mapped directly from the metamodel, the rest of properties are computed from the metamodel.
- Compact representation of data. This issues is related to how the tuples of classes are generated. We avoid generating all permutations of class sequences in the tuples. For example, if the tuple (C A P) is generated, and we subsequently generate (A P C) or (C P A), we only keep one of these as being representative of all three alternatives.
- Multiple Contexts. In this case, we have used 3 lattices. Each lattice is for one order of the elements: 2, 3 and 4. All use the same set of properties.
- Performance of the algorithm. With the tuples of classes of order more than 4, we

are not able to have a reasonable computation time of the algorithm. With one of the applications using tuples of classes of order 4, the computation time took around 2 days and this is not acceptable time from our viewpoint to software engineering.

- Meaningless concepts. As we said previously, each concept is a candidate for a pattern. In this case, each concept represents a graph in which the set of the elements is the set of nodes, and the set of properties define the edges that should connect the nodes, *e.g.*, if (A P C) has the properties *A is subclass of P* and *P uses C*, then we have a graph of 3 nodes with edges from A to P and from P to C. Thus we discarded all the concepts that represent graphs in which one or several nodes are not connected at all with any node in the graph.
- Mapping Partial Order of the Lattice. In this case, we map the partial order of the lattice to the definition of *neighbours* of a pattern. We can detect either missing relationships between classes needed to complete a pattern, or excess relationships between classes that extend a pattern.
- Limits of *Collaboration Patterns*. In this high level view, we consider that there are still possible new collaboration patterns to detect when applying the approach in other applications.

4 Lessons Learned

In general terms, we have seen that Formal Concept Analysis is a useful technique in reverse engineering. From our experiencies [Aré03, ABN04, ADN03] in developing this approach, several lessons learned are worthwhile mentioning.

Lack of a general methodology. The main problem we have found in the state of the art is the lack of a general methodology for applying FCA to the analysis of software. In most publications related to software analysis, the authors only mention the FCA mapping as a trivial task, and how they interpret the concepts. With our approach, we achieved not only to identify clear steps for applying FCA to a piece of software but where we have identified different bottlenecks in using the technique.

Modelling software entities as FCA components. The process of modelling software entities as FCA components is one of most difficult tasks and we consider it as one of the critical steps in the approach. Modelling is an iterative process in which we map software entities to FCA components, and we test whether they are useful enough to provide meaningful results. This task is not trivial at all, because it entails testing at least 5 small case studies (which should be representative of larger case studies). Each case study should help to refine the building of FCA components.

Performance of FCA algorithms. Part 1 The performance of the algorithms (to build the concepts and lattice) was one of main bottlenecks. In small case studies, this factor can be ignored because the computation time is insignificant. But in large case studies this factor can cause the complete approach to fail because computing the concepts and the lattice may take several hours (eventually days).

Performance of FCA algorithms. Part 2 The computation of the FCA algorithms is also affected by how they are implemented in a chosen language. A direct mapping of

the algorithms (in pseudo-code, as they are presented in books) to a concrete programming language is not advisable. In our specific case, we took advantage of efficient data structures in *Smalltalk* language to represent the data and improve the performance.

Supporting Software Engineers. The result of our experiences must be read by software engineers. One positive issue in this point is that with the high level views the software engineer is not obliged to read the concept lattices , meaning that he need not be a FCA expert.

Interpretation of the concepts. Although we can have an adequate choice of FCA elements and properties, the interpretation of the concepts is a difficult and time-consuming task. In most of the cases, we have tried to associate a meaning (a name) to each concept based on the conjunction of its properties. This task must be done by a software engineer applying *opportunistic code reading* to get meaningful interpretations. This process is completely subjective because it depends on the knowledge and experience of the software engineer.

Use of the Complete Lattice. Not all the concepts have a meaning in our approach, so we do not use the complete lattice in our analysis. In most of the cases, we remove meaningless concepts because from software engineering viewpoint they did not provide enough information for the analysis. We hypothesize that in certain cases it may be possible to use the complete lattice, but we did not find any.

Use of the Partial Order. Another critical factor is the interpretation of the partial order of the lattice in terms of software relationships. Only in *Collaboration Patterns* we were able to obtain a satisfactory interpretation of the partial order. So far, the interpretation is not a trivial task.

5 Related Work

Several researchers have also applied FCA to the problem of understanding object oriented software. Dekel uses CA to visualize the structure of the class in Java and to select an effective order for reading the methods and reveal the state usage [Dek03]. Godin and Mili [GMM+98] use concept analysis to maintain, understand and detect inconsistencies in the Smalltalk *Collection* hierarchy. They show how Cook's [Coo92] earlier manual attempt to build a better interface hierarchy for this class hierarchy (based on interface conformance) could be automated. In C++, Snelting and Tip [ST98] analysed a class hierarchy making the relationship between class members and variables explicit. They were able to detect design anomalies such as class members that are redundant or that can be moved into a derived class. As a result, they propose a new class hierarchy that is behaviorally equivalent to the original one. Similarly, Huchard [HDL00] applied concept analysis to improve the generalization/specialization of classes in a hierarchy. Tonella and Antoniol [TA99] use CA to detect the *structure* of Gamma-style design patterns using relationships between classes, such as *inheritance* and *composition*.

6 Conclusions and Future Work

In this paper we present a general approach for applying FCA in reverse engineering of object oriented software. We also evaluate the advantages and drawbacks of using FCA as a metatool for our reverse engineering approaches. We also identify the different bottlenecks of the approach. Thus, we are able to focus clearly on solving which and where the limitations appear (if there are some possible solutions) to draw the maximum benefit offered by FCA. From our tool builder viewpoint, we have proven that FCA is an useful technique to identify *groups* of software entities with hidden dependencies in a system. With FCA, we have built different software engineering tools that help us to generate *high level views* at different levels of abstraction of a system. We generate the *high level views* because without them, the *software engineer* should be obliged to read the lattice. This can represent a problem because in most of the cases, besides the useful information, the lattice can also have useless information that can introduce *noise* in analyzing a system.

Our future work is focused on several research directions that consist of: (1) the development of new case studies to analyze how useful the approach is or if there are still some refinements and improvements to do, (2) Testing with other concept and lattice building algorithms to see if we can improve the computation time of the ConAn engine and(3) Analysis of the partial order to get possible mappings in terms of software engineering relationships.

Acknowledgments. We gratefully acknowledge the financial support of the Swiss National Science Foundation for the projects "Tools and Techniques for Decomposing and Composing Software" (SNF Project No. 2000-067855.02), and Recast: Evolution of Object-Oriented Applications (SNF 2000-061655.00/1).

References

[ABN04] Gabriela Arévalo, Frank Buchli, and Oscar Nierstrasz. Detecting implicit collaboration patterns. In *Proceedings of WCRE 2004*. IEEE Computer Society Press, November 2004. to appear.

[ADN03] Gabriela Arévalo, Stéphane Ducasse, and Oscar Nierstrasz. X-Ray views: Understanding the internals of classes. In *Proceedings of ASE 2003*, pages 267–270. IEEE Computer Society, October 2003. Short paper.

[Aré03] Gabriela Arévalo. Understanding behavioral dependencies in class hierarchies using concept analysis. In *Proceedings of LMO 2003*, pages 47–59. Hermes, Paris, January 2003.

[Aré05] Gabriela Arévalo. *High Level Views in Object Oriented Systems using Formal Concept Analysis*. PhD thesis, University of Berne, January 2005. forthcoming.

[BDW98] Lionel C. Briand, John W. Daly, and Jürgen Wüst. A unified framework for cohesion measurement in object-oriented systems. *Empirical Software Engineering: An International Journal*, 3(1):65–117, 1998.

[Coo92] William R. Cook. Interfaces and specifications for the Smalltalk-80 collection classes. In *Proceedings OOPSLA '92, ACM SIGPLAN Notices*, volume 27, pages 1–15, October 1992.

[DDN02] Serge Demeyer, Stéphane Ducasse, and Oscar Nierstrasz. *Object-Oriented Reengineering Patterns*. Morgan Kaufmann, 2002.

[Dek03] Uri Dekel. Revealing java class structures using concept lattices. Diploma thesis, Technion-Israel Institute of Technology, February 2003.

[DLT00] Stéphane Ducasse, Michele Lanza, and Sander Tichelaar. Moose: an extensible language-independent environment for reengineering object-oriented systems. In *Proceedings of CoSET 2000*, June 2000.

[DRW00] Alastair Dunsmore, Marc Roper, and Murray Wood. Object-oriented inspection in the face of delocalisation. In *Proceedings of ICSE 2000*, pages 467–476. ACM Press, 2000.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, Reading, Mass., 1995.

[GMM+98] Robert Godin, Hafedh Mili, Guy W. Mineau, Rokia Missaoui, Amina Arfi, and Thuy-Tien Chau. Design of class hierarchies based on concept (galois) lattices. *Theory and Application of Object Systems*, 4(2):117–134, 1998.

[GW99] Bernhard Ganter and Rudolf Wille. *Formal Concept Analysis: Mathematical Foundations*. Springer Verlag, 1999.

[HDL00] M. Huchard, H. Dicky, and H. Leblanc. Galois lattice as a framework to specify algorithms building class hierarchies. *Theoretical Informatics and Applications*, 34:521–548, 2000.

[LPLS96] David Littman, Jeannine Pinto, Stan Letovsky, and Elliot Soloway. Mental models and software maintenance. In Soloway and Iyengar, editors, *Empirical Studies of Programmers, First Workshop*, pages 80–98, 1996.

[LRP95] John Lamping, Ramana Rao, and Peter Pirolli. A focus + context technique based on hyperbolic geometry for visualising larges hierarchies. In *Proceedings of CHI '95*, 1995.

[SG95] Raymie Stata and John V. Guttag. Modular reasoning in the presence of subclassing. In *Proceedings of OOPSLA '95*, pages 200–214. ACM Press, 1995.

[SLMD96] Patrick Steyaert, Carine Lucas, Kim Mens, and Theo D'Hondt. Reuse contracts: Managing the evolution of reusable assets. In *Proceedings of OOPSLA '96*, pages 268–285. ACM Press, 1996.

[ST98] Gregor Snelting and Frank Tip. Reengineering class hierarchies using concept analysis. In *ACM Trans. Programming Languages and Systems*, 1998.

[TA99] Paolo Tonella and Giuliano Antoniol. Object oriented design pattern inference. In *Proceedings ICSM '99*, pages 230–238, October 1999.

[TDDN00] Sander Tichelaar, Stéphane Ducasse, Serge Demeyer, and Oscar Nierstrasz. A meta-model for language-independent refactoring. In *Proceedings ISPSE 2000*, pages 157–167. IEEE, 2000.