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to identify these dependencies in non-trivial applications because system documentation tends to be inadequate or out-of-date and because the information we seek is not explicit in the code [DDN02, SLMD96, LRP95].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, 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.

B. Ganter and R. Godin (Eds.): ICFCA 2005, LNAI 3403, pp. 95–112, 2005.

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.
- *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].

[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.