Software Reengineering Based on Concept Lattices

Gregor Snelting
Universität Passau
Lehrstuhl Softwaresysteme
snelting@fmi.uni-passau.de

1 Overview

Concept analysis provides a way to identify groupings of objects that have common attributes. The mathematical foundation was laid by G. Birkhoff [1], who proved that for every binary relation between certain “objects” and “attributes”, a lattice can be constructed which allows remarkable insight into the structure of the original relation. The relation can always be reconstructed from the lattice, hence concept analysis is similar in spirit to Fourier analysis.

Later, Wille and Ganter elaborated Birkhoff’s result and transformed it into a data analysis method [10, 3]. Since then, it has found a variety of applications, such as analysis of Rembrandt’s paintings, classification of algebraic structures, and behaviour of drug addicts. In 1993, work on the application of concept analysis in the area of program understanding and reengineering was initiated. Concept analysis has been used for modularization of legacy code [5, 6, 2], finding interferences between configurations [4, 7], and transformation of class hierarchies [8, 9].

2 Concept lattices

In this overview, we will not present the elaborated and beautiful mathematical and algorithmic background (the interested reader should consult [3]), but will merely present an example explaining concept lattices.

Concept analysis starts with a relation, or boolean table, between a set of objects and a set of attributes. As an example, consider the table describing properties of the planets of the solar system (figure 1). The corresponding lattice offers insight not obvious from the original table. In particular,

- each lattice element (called a “concept”) corresponds to a maximal rectangle in the table;
- the lattice is however independent of row or column permutations in the table;
Figure 1: A table describing properties of planets and the corresponding concept lattice

- lattice elements are labelled with attributes and objects;
- object \( o \) has attribute \( a \) in the table if and only if \( o \) appears below \( a \) in the lattice;
- the lattice presents a hierarchical clustering of objects and attributes;
- suprema factor out common attributes, e.g. “Mars and Venus are both near”;
- infima factor out common objects, e.g. “Pluto is both small and far”;
- upward arcs are implications, e.g. “Any planet without moon is also near and small”.

From a large table, such insights are hard to obtain manually. Lattice construction can be exponential in the worst case, but is almost linear in practice.

3 Assessing modular structures

In this section, we want to show how concept analysis can be used to assess the modular structure of legacy code and perhaps modularize old systems. We try to find modules in legacy code by analysing the relation between procedures and global variables. Hence the objects are the procedures of a program, the attributes are the global variables, and the variable usage table has entry \((p, v)\) if procedure \( p \) uses variable \( v \).

A module consists of a set of procedures \( P \) and a set of variables \( V \) such that all procedures in \( P \) use only variables in \( V \) and all variables in \( V \) are only used by procedures in \( P \). This definition captures the essence of information hiding. In the table, a module shows up as a maximal rectangle. This rectangle, however, need not be completely filled – not every procedure in a module uses all module variables, and not all module variables are used by all procedures.
We say that two sets of procedures (resp. their modules) are coupled if they use the same global variable(s). Similarly, two sets of variables (resp. their modules) interfere, if they are used by the same procedure. Although coupling via global variables is undesirable, in a reengineering setting coupling might be acceptable if there are nested local modules or procedures. Interferences however prevent a modularization, as there is a procedure which uses variables from two different modules – a violation of the information hiding principle.

Figure 2 presents the variable usage table for a Modula-2 program from a student project. The program is about 1500 lines long and divided into 8 modules; there are 33 procedures which use 16 module variables. The corresponding lattice is horizontally decomposable: it consists of independent substructures (each for one module), connected only via top and bottom element. Any program sticking to modularization and information hiding must generate a horizontally decomposable lattice.

In case there are only a few interferences between horizontal summands, the modular structure is still good (see lower right part in figure 2). Interferences can be detected automatically and removed by simple program transformations such as encapsulation of global variables.

We examined several legacy systems written in Fortran and Cobol. One example is an aerodynamics system used for airplane development in a national research institution. The system is about 20 years old, and has undergone countless modifications and extensions.
The source code is 106000 lines long, consists of 317 subroutines, and uses 492 global variables in 46 COMMON blocks. One of the goals of the analysis is to reshape COMMON blocks such that each module corresponds to one COMMON block. Several manual restructuring efforts had not been very successful, so it was decided to try concept analysis.

After the variable usage table was built, the lattice was constructed\(^1\). It contains no less than 2249 elements. The number of elements in itself is not the problem (after all, it is a large program), but unfortunately the lattice is so full of interferences that it is impossible to reveal any structure (Figure 3). There is no way to make the lattice horizontally decomposable by removing just a small number of interferences. We also tried to analyse just parts of the system, with no better results either.

Generally speaking, the presence of module candidates must correspond to some partitioning of the variables, and such partitionings can be found by lattice decompositions such as horizontal decomposition. In the example, the overwhelming number of interferences

\(^{1}\)This required 11 seconds on a SparcStation20.
prevents a partitioning and hence a modularization. Based on these results, the national
institution decided to cancel a reengineering project for this system, and develop a new sys-
tem from scratch. Thus in this case, the concept lattice did not generate a modularization,
but served as a quality metrics.

Another approach was studied by Siff and Reps [6]. They not only consider the use
of global variables, but also use of types, or the fact that a procedure does not use a
variable or type. A modularization is obtained by finding lattice elements which provide
a partition of the attribute space. Siff reports good results on small C programs. Van
Deursen and Kuipers use concept analysis for analysing records in Cobol programs in
order to identify object candidates [2]; they report good success on a real-life Cobol legacy
system. According to van Deursen, it is important to filter out "objects" and "attributes"
which are not application specific (in fact, they only analysed records associated with
persistent file structures). In general, a system can be modularized automatically only if
it is not too degenerate.

4 Exploring configuration spaces

Our next application is the analysis of configuration spaces. We concentrated our efforts on
UNIX source files, where variants and versions are often managed using the C preprocessor
CPP.

Using CPP, objects are code pieces (consecutive source line intervals), while the at-
tributes are derived from the CPP expressions governing each code piece. Figure 4 presents
a simple example which shows how a configuration table is derived from a source file. In
the corresponding lattice, a concept represents a specific configuration thread, namely a
set of code pieces selected by the same CPP expressions. The example lattice also displays
an interference between two configuration threads, namely a code piece governed by two
supposedly independent, or orthogonal, CPP symbols. Interferences show up as infima not
labelled with a CPP symbol. In the example, X_win and DOS are – as everybody knows –
even mutually exclusive, hence the interference indicates that code piece IV is dead code.

But governing conditions can be arbitrary boolean expressions; furthermore, #ifdefs
and #ifs may be nested. Thus it is not so obvious what the "attributes" should be. The
handling of complex governing expressions is explained in detail in [7]: governing expres-
sions are transformed into conjunctive normal form; then additional columns for elementary
disjunctions and negations are introduced. As an example, consider the right part of figure
4: the lattice displays an interference between elementary disjunctions DOS||X_win and
UNIX||DOS. In the source text, it seems that code piece II is governed by the simple ex-
pression DOS, but since DOS also appears in the governing expression for code piece V,
there is a subtle interdependency between the corresponding configuration threads - visible
in the lattice. In general, a good configuration lattice is horizontally decomposable (just
as a good module lattice, see last section). If there are not too many interferences between
sublattices, they can be removed by modification of the CPP files.

One of the source files we analysed was the stream editor rcseedit from the RCS system.
...I...
#ifdef DOS
...II...
#endif
#ifdef OS2
...III...
#endif
#endif defined(DOS) || defined(X_win)
...IV...
#endif
#ifdef X_win
...V...
#endif
#endif VI...

#ifdef UNIX
...I...
#endif
#ifdef DOS
...II...
#endif
#endif defined(DOS)
|| defined(X_win)
...III...
#endif
#endif defined(UNIX)
...IV...
#endif
#endif defined(X_win)
|| (defined(DOS) && defined(X_win))
...V...
#endif

Figure 4: Two simple CPP files and their lattices. Disjunctions can cause interference.

This program is 1656 lines long and uses 21 CPP variables for configuration management. Its configuration lattice, together with the labelling of the lattice elements is shown in Figure 5. The top element C1 represents the code pieces not governed by anything. The left-hand side of the lattice is quite flat (C18 to C4) which means that there are many configurations which do not influence each other. From a software engineering viewpoint, this is desirable, as it indicates low coupling between configuration threads.

There are, however, some interferences in the right-hand side. For example, C27, representing the source line 1426, is the infimum of C3 and C26. The latter are labelled hasrename resp. hasNFS; hasrename has to do with the file system, while hasNFS is concerned with the network. These should be independent (transparency of the network), but the lattice reveals that they are not. A look into the source code reveals the following comment for line 1426: “An even rarer NFS bug can occur when clients retry requests. ... This not only wrongly deletes B’s lock, it removes the RCS file! ... Since this problem affects scads of Unix programs, but is so rare that nobody seems to be worried about it, we won’t worry either.”

5 Transforming class hierarchies

Recently, we combined concept analysis with dataflow analysis and type inference in order to improve object-oriented systems. Our goal is to analyse Java or C++ programs with respect to their member access patterns. We want to infer a transformed class hierarchy which is semantically equivalent to the original one, but reflects actual member accesses in the program (in contrast to the static hierarchy as defined in the program text). Technically, for every variable a new type is computed which contains exactly the data members and methods which must be visible to the variable due to its actual behaviour. The new
types are arranged in a new class hierarchy, obtained as a concept lattice. The new hierarchy is usually more fine-grained, indicating that the old classes may be split or refactored. Average object size has usually decreased, hence the approach can also be considered a hyper-aggressive space optimization. The program statements are unchanged!

The method is quite complex, and the interested reader can find details in [8, 9]. As an example, consider figure 6. A small class hierarchy is presented together with a main program using it. The upper right table summarizes all member accesses occurring in the source text. “Objects” are all variables and pointers from the program, including this pointers. “Attributes” are all data members and methods from the program. Note that a conservative approximation for dynamic binding must be used during table construction, since the exact target of a method call is unknown at analysis time. This approximation is achieved via points-to analysis (an analysis which for every pointer computes a – hopefully
class A {
    public:
    virtual int f() { return g(); };
    virtual int g() { return x; };
    int x;
};
class B : public A {
    public:
    virtual int g() { return y; };
    int y;
};
class C : public B {
    public:
    virtual int f() { return g() + z; };
    int z;
};

Figure 6: Sample C++ program; initial member access table and tables after propagating assignment and dominance type constraints; final lattice.
small – set of objects it can point to at runtime). There are also some subtleties in connection with class-typed data members, nonvirtual methods and this pointers, which will not be explained here.

The original program generates several type constraints which are essential for program behaviour. For example, in any assignment the type of the left-hand side must be a superclass of the type of the right-hand side. Such constraints are incorporated into the table in form of *implications* [3]. Implications are indicated as arrows between rows in the upper right table in figure 6; an implication from row $x$ to row $y$ means that all entries in $x$ must be copied to $y$ and will enforce $x \leq y$ in the lattice. Furthermore, certain sub-/superclass relations must be retained in order to preserve visibility properties of members which have been redefined in subclasses. The latter constraints are more complex to determine; they are indicated as implications between columns in the lower left table. A fix-point iteration applies implications until the table has stabilized, and finally the concept lattice is computed.

The resulting lattice can directly be interpreted as a class hierarchy: lattice elements are classes, “attribute labels” are class members, and “object labels” are class-typed variables. We have proven that the new hierarchy is operationally equivalent to the old hierarchy. But usually it contains more classes, and objects are smaller. In the example, we see that $b$ does not access $A::f$ and $A::x$, and $c$ does not access any of $A$'s original members. Hence the new types of $b$ resp. $c$ are not a subclass of the new type of $a$ anymore, resulting in
Figure 9: Refactoring proposal for regular expression subhierarchy.

reduced space requirements for objects b resp. c.

Instead of discussing more details of this powerful analysis method, we would like to conclude with a realistic example. The "JEdit" text editor (ca. 9000 LOC) uses a Java reimplementation of the GNU regular expression library, and we will now show how our method generates restructuring proposals for this library. Figures 7 and 8 present the original and the transformed class hierarchy, as displayed by our implementation KABA. Most of the classes are just reproduced by KABA, indicating that the original class structure was good.

But there is a subhierarchy "gnu.regexp.REToken" which implements regular expression search. The transformed regular expression search is much more fine-grained (figure 9); for example, KABA discovers a distinction between "regular expression search without substitution" and "regular expression search with substitution". Note that the lattice displays the finest possible splittings and refactorings of classes according to actual program behaviour. For reengineering purposes, the lattice should therefore be simplified in order to reflect software design principles. Semantics-preserving simplifications based on the structure theory of concept lattices are discussed in [9].
References


