Concept Analysis – A New Framework for Program Understanding

Gregor Snelting
Technische Universität Braunschweig
Abteilung Softwaretechnologie
`snelting@ips.cs.tu-bs.de`

Abstract

Concept analysis transforms any relation between "objects" and "attributes" into a complete lattice. This concept lattice can be studied by algebraic means and offers remarkable insight into properties and structure of the original relation. As relations between "objects" and "attributes" occur all the time in software technology, concept analysis is an attractive foundation for a new class of program analysis tools. The article presents a short overview of the underlying theory, as well as applications for software component retrieval, analysis of configuration spaces, and modularization of legacy code.

1 Overview

Concept analysis provides a way to identify groupings of *objects* that have common *attributes*. The mathematical foundation was laid by G. Birkhoff in 1940 [1]. Birkhoff 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 to Fourier analysis.

Later, R. Wille and B. Ganter elaborated Birkhoff's result and transformed it into a data analysis method [15, 19]. 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 finding interferences between configurations [5, 14], improving software component retrieval [6, 7], learning from databases [4], and modularization of legacy code [8, 13]. It is the aim of this article to demonstrate that concept analysis is an elegant and powerful tool, and a useful framework for a new class of program analysis algorithms.

2 Mathematical Background

2.1 Relations and their lattices

Concept analysis starts with a relation, or boolean table, T between a set of objects $\mathcal{O}$ and a set of attributes $\mathcal{A}$, hence $T \subseteq \mathcal{O} \times \mathcal{A}$. The triple

$$\mathcal{C} = (\mathcal{O}, \mathcal{A}, T)$$

is called a formal context.

For any set of objects $O \subseteq \mathcal{O}$, their set of common attributes is defined by

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

Similarly, for any set of attributes, their set of common objects is

$$\tau(A) = \{o \in \mathcal{O} \mid \forall a \in A : (o, a) \in T\}$$

In fact, σ and τ form a Galois connection (a pair of two antimonotone functions), and both $\sigma \circ \tau$ and $\tau \circ \sigma$ are closure operators: e.g., $\sigma \circ \tau(O)$ determines the biggest set of objects which have the same attributes as O. A pair (O, A) is called a concept, if

$$A = \sigma(O) \text{ and } O = \tau(A)$$

Informally, such a concept corresponds to a maximal rectangle in the table T. Note that concepts are invariant against row or column permutations. For a concept $c = (O, A)$, $O = \mathit{ext}(c)$ is called the *extent* and $A = \mathit{int}(c)$ is called the *intent* of c.

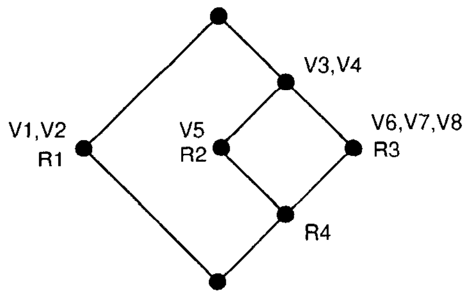
```
SUBROUTINE R1(...)
COMMON /C1/ V1,V2
...
END
```

```
SUBROUTINE R3(...)
COMMON /C2/ V3,V4
COMMON /C4/ V6,V7,V8
...
END
```

```
SUBROUTINE R2(...)
COMMON /C2/ V3,V4
COMMON /C3/ V5
...
END
```

```
SUBROUTINE R4(...)
COMMON /C2/ V3,V4
COMMON /C3/ V5
COMMON /C4/ V6,V7,V8
...
END
```

	V1	V2	V3	V4	V5	V6	V7	V8
R1	×	×						
R2			×	×	×			
R3			×	×		×	×	×
R4			×	×	×	×	×	×



$V8$	$\rightarrow$	$V7\ V6\ V4\ V3$
$V7$	$\rightarrow$	$V8\ V6\ V4\ V3$
$V6$	$\rightarrow$	$V8\ V7\ V4\ V3$
$V5$	$\rightarrow$	$V4\ V3$
$V4\ V3\ V2\ V1$	$\rightarrow$	$V8\ V7\ V6\ V5$
$V4$	$\rightarrow$	$V3$
$V3$	$\rightarrow$	$V4$
$V2$	$\rightarrow$	$V1$
$V1$	$\rightarrow$	$V2$

Figure 1: A formal context, its concept lattice, and its minimal implication base; extracted from a source text.

The set of all concepts of a given table forms a partial order via

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

G. Birkhoff discovered in 1940 that it is also a complete lattice, the concept lattice

$$\mathcal{L}(\mathcal{C}) = \{(O, A) \in 2^{\mathcal{O}} \times 2^{\mathcal{A}} \mid A = \sigma(O) \wedge O = \tau(A)\}$$

In this lattice, the infimum (or join) of two concepts is computed by intersecting their extents:

$$(O_1, A_1) \wedge (O_2, A_2) = (O_1 \cap O_2, \sigma(O_1 \cap O_2))$$

Note that $A_1 \cup A_2 \subseteq \sigma(O_1 \cap O_2)$, as $O_1 \cap O_2$ has at least common attributes $A_1 \cup A_2$. Thus an infimum describes the set of attributes common to two sets of objects. Similarly, the supremum (or meet) is computed by intersecting the intents:

$$(O_1, A_1) \vee (O_2, A_2) = (\tau(A_1 \cap A_2), A_1 \cap A_2)$$

Again, $O_1 \cup O_2 \subseteq \tau(A_1 \cap A_2)$. Thus a supremum describes a set of common objects which fit to two sets of attributes.

If the concepts are labelled with their extent and intent, the lattice is hard to understand. Fortunately, there is a much simpler way to indicate extent and intent of the concepts in a lattice. A lattice element is labelled with attribute $a \in \mathcal{A}$, if it is the largest concept having a in its intent; it is labelled with an object $o \in \mathcal{O}$, if it is the smallest concept having o in its extent. The (unique) lattice element labelled with a is thus

$$\mu(a) = \bigvee\{c \in \mathcal{L}(\mathcal{C}) \mid a \in int(c)\}$$

The element labelled with o is

$$\gamma(o) = \bigwedge\{c \in \mathcal{L}(\mathcal{C}) \mid o \in ext(c)\}$$

All concepts greater than $\gamma(o)$ have o in its extent, and all concepts smaller than $\mu(a)$ have a in its intent.

The following remarkable property establishes the connection between a table and its lattice, and shows that they can be reconstructed from each other:

$$(o, a) \in T \iff \gamma(o) \leq \mu(a)$$

Hence the attributes of object o are just those which show up above o in the lattice, and the objects for attribute a are those which show up below a.

Utilizing the labelling, suprema in the lattice indicate that certain objects have common attributes, while infima show that certain attributes fit to common objects. Another way to express this is to say that suprema factor out common attributes, and infima factor out common objects. Thus the lattice uncovers a hierarchy of conceptional clusters implicit in the original table. This observation explains part of the power of concept lattices.

2.2 Interpretation of concept lattices

Figure 1 gives a very small example of a formal context and its concept lattice. The context table is generated from a (fictious) FORTRAN source file and captures the use of global variables by subroutines. The corresponding lattice shows that all subroutines below μ(V3) (namely R2, R3, R4) use V3 (and no other subroutines use V3). All variables above γ(R4) (namely V3, V4, V5, V6, V7, V8) are used by R4 (and no other variables are used by R4). Thus the concept labelled R4 is in fact

$$c_1 = \gamma(\mathrm{R4}) = (\{\mathrm{R4}\}, \{\mathrm{V3,V4,V5,V6,V7,V8}\})$$

The concept labelled V5/R2 is in fact

$$c_2 = \mu(\mathrm{V5}) = \gamma(\mathrm{R2}) = (\{\mathrm{R2,R4}\}, \{\mathrm{V3,V4,V5}\})$$

Hence $c_1 \leq c_2$, as c_1 has fewer procedures and more variables. This can be read as an implication: "Any variable used by subroutine R2 is also used by R4". Similarly, $\mu(\mathrm{V5}) \leq \mu(\mathrm{V3}) = \mu(\mathrm{V4})$, which translates to "All subroutines which use V5 will also use V3 and V4". The infimum of V5/R2 and V6,V7,V8/R3 is labelled R4, which means that R4 (and all subroutines below γ(R4),[1] but no other) uses both V5 and V6,V7,V8. The supremum of the same concepts is labelled V3,V4, which means that V3 and V4 (and all variables above μ(V2), but no other) are used by both R2 and R3. Such knowledge is not easy to obtain manually from big source files!

The example already demonstrates several possibilities to interpret the lattice. Wille [18] presents the following list:

1. Concepts determine maximal object sets with identical attributes. E.g., procedures which use the same global variables, or students which solved the same exercises.

2. The lattice displays a hierarchical classification of (sets of) objects. E.g., R4 uses more variables than R2 or R3.

3. Irreducible elements (i.e., not obtainable as infimum or supremum) in the lattice correspond to "fundamental" objects resp. attributes. In Figure 1, all elements are either inf- or sup-irreducible.

4. Repeated patterns (sublattices) in the lattice show that certain groups of attributes are in fact different instances of an "abstract" attribute.

5. Congruences or weak congruences (so-called block relations) allow to partition the table into "independent" subtables. In the example, R1, V1,V2 can be separated from the rest of the table, as R1,V1,V2 and R2,R3,R4,V3,V4,V5,V6,V7,V8 are both congruence classes.

6. Algebraic decompositions of the lattice, such as subdirect or subtensorial decomposition, reveal hidden structure in the objects or attributes themselves.

7. The table is the "logarithm" of the lattice, as algebraic operations on the lattice have a counterpart in the table.

8. Table or lattice can also be represented by a complete, minimal set of implications between attributes. An implication between attribute sets, written $A \rightarrow B$, means "any object which has all attributes in A also has all attributes in B". In Figure 1, there is e.g., the implication $V5 \rightarrow V4\ V3$, which corresponds to an upward arc in the lattice.

9. An implication base can also be constructed by an incremental, interactive knowledge aquisition process.

[1] there are none in the example

In this overview, we cannot explain all these topics in detail. For a discussion of elementary concept analysis, see [2]. The book [19] treats the mathematical theory in depth. There are fast algorithms for the construction and algebraic decomposition of concept lattices. Construction of concept lattices and implication bases has typical time complexity $O(n^3)$ for an $n \times n$ table, but can be exponential in the worst case.

3 Improving software component retrieval

We will now turn to applications of concept analysis in software technology. Our first application is rather simple: we will demonstrate how to improve software component retrieval. We wish to state in advance that concept analysis should not replace other retrieval techniques, but should be considered an "amplifier".

We assume that software components are indexed by keywords. It is not important how the indexing has been obtained, manually or automatically. The indexing, written as a table, is of course a formal context, and the corresponding concept lattice can be computed (Figure 2).

What do we get for our money, namely the computational investment into the lattice? First of all, the lattice shows a hierarchy of keywords which is only implicit (or, more precisely, invisible) in the original table. For example, the lattice reveals that the keywords *read* and *write* have a common superattribute, namely the keyword *file*. As mentioned in the theory part, this can be read as an implication: "any component which

	access	*change*	*check*	*create*	*directory*	*file*	*get*	*group*	*input*	*mode*	*new*	*open*	*output*	*owner*	*permission*	*process*	*read*	*remove*	*status*	*write*
access	×		×			×														
chdir		×			×															
chmod		×				×				×					×					
chown		×				×		×						×						
creat				×		×					×									
fork				×							×					×				
fstat						×	×												×	
mkdir				×	×						×									
open				×		×						×					×			×
read						×			×								×			
rmdir					×	×												×		
write						×							×							×

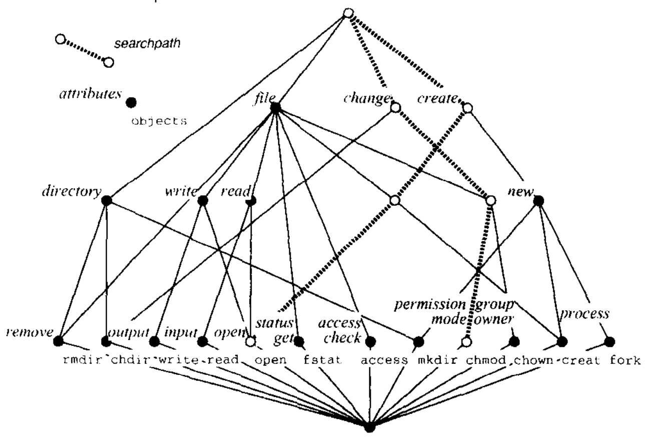


Figure 2: Components indexed by keywords

is indexed with *read* or *write* is indexed with *file* as well". *write*, in turn, is the superattribute for keywords *output* and *open*, and the lattice shows that component `write` is indexed by the keywords above `write`, namely *input*, *write*, and *file*. The reader should keep in mind that the hierarchy of keywords is solely derived from the table, and looks different if more entries are added to the table.

The lattice can also be used to check or disprove background knowledge. If the user believes that everything which is *create*d also is *new*, then the lattice shows that this believe is wrong, as *new* is a subconcept of *create* and not vice versa.

How can the lattice support searching? Any set of keywords Q may be used as a search key to determine a lattice element c, which is the infimum of the elements labelled with the keywords: $c = \bigwedge\{\mu(q) \mid q \in Q\}$. For example, the search key $Q = \{file, new\}$ identifies the component `creat`[2], as there is no other component below both *file* and *new*. The key $\{file, create\}$ selects the unlabelled center element. Only components below this element will fit to both *file* and *create*. Hence any search key *narrows the search space* (Figure 3). More search keys may be added incrementally, constraining

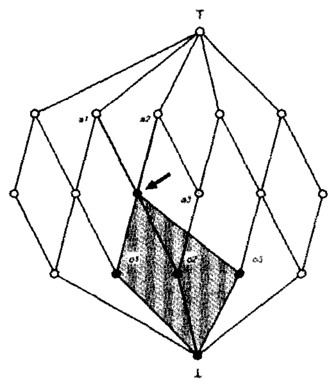


Figure 3: Narrowing the search space

the possible components even further. This behaviour is much more flexible than e.g., faceted search [12], where keywords must be given in a fixed order, and incomplete queries are not possible.

As a side effect, any search key also determines the set of still possible keywords, namely those which are consistent with the preceding search keywords. After selecting *file* and *create*, search key *new* is no longer possible, because there is no element below *file* and *create*, which is also below *new*. In general, a keyword k is inconsistent with a (partial) query Q, if $\mu(k) \wedge \bigwedge\{\mu(q) \mid q \in Q\} = \perp$. Thus not only can the set of possible search results be narrowed incrementally, the set of still possible keywords is narrowed as well. Using the precomputed lattice, this context-sensitive support can be presented to the user very fast, while the lattice itself need not be visible [6, 7].

4 Exploring configuration spaces

Our next application is the analysis of configuration spaces. Our work was motivated by Parnas, who pointed out: "When a large and important family of products gets out of control, a major effort to restructure it is appropriate. The first step must be to reduce the size of the program family. One must examine the various versions to determine why and how they differ" [10]. The application of concept analysis to this problem is quite obvious, because configuration management systems typically select and compose software components (objects) according to certain features (attributes). We concentrated our efforts on UNIX source files, where variants and versions are often managed using the C preprocessor CPP. A lot of source code sticking to the "configuration selection by preprocessing" scheme is around, which makes it an ideal target for reengineering studies.

[2]not to be confused with the element labelled *create*

```
...I...
#ifdef DOS
...II...
#endif
#ifdef OS2
...III...
#endif
#if defined(DOS)
    && defined(X_win)
...IV...
#endif
#ifdef X_win
...V...
#endif
...VI...
```

	DOS	OS2	X_win
I			
II	×		
III		×	
IV	×		×
V			×
VI			

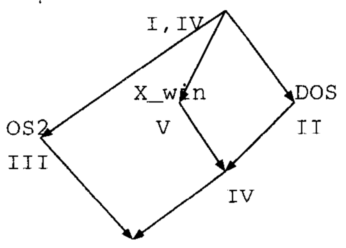


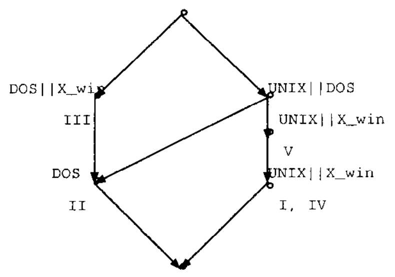
Figure 4: A simple CPP file and its configuration lattice

4.1 The configuration table and its lattice

Using CPP, objects are code pieces (consecutive source line intervals), while the attributes 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 [14]: governing expressions are transformed into conjunctive normal form; then additional columns for elementary disjunctions and negations are introduced, as a formal context cannot express negated or disjunctive attributes directly. In order that this transformation be correct, additional implications have to be introduced as well. As an example, consider Figure 5. 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 expression 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.

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



	DOS	UNIX	DOS||X_win	UNIX||X_win	UNIX||DOS
I		×		×	×
II	×		×		×
III			×		
IV		×			
V				×	×

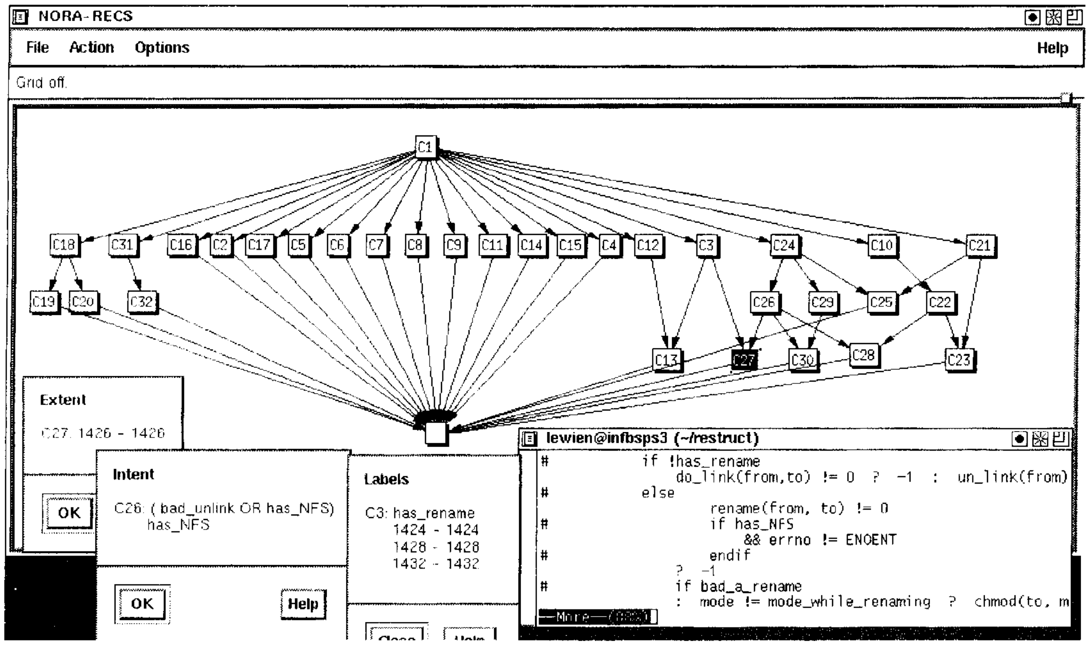
Figure 5: Disjunctions can cause interference

4.2 Interference analysis

One of the source files we analysed was the stream editor rcsedit from the RCS system. 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 6; it has been computed and layouted by the tool NORA/RECS.[3] 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 source line 1426, is the infimum of $C3$ and $C26$. The latter are labelled has_rename resp. has_NFS; has_rename has to do with the file system, while has_NFS 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 afflicts scads of Unix programs,

[3]NORA/RECS offers display of concept labels and corresponding code pieces upon mouse click, display of irreducible elements, sublattices, congruences, subdirect decomposition, subtensorial decomposition, horizontal decomposition, and automatic interference detection. Still, it is an experimental implementation not fit for field use.



Extent

C27: 1426 - 1426

Intent

C26: (bad_unlink OR has_NFS)
has_NFS

Labels

C3: has_rename
1424 - 1424
1428 - 1428
1432 - 1432

lewien@infbsps3 (~/restruct)

```
#           if !has_rename
                do_link(from,to) != 0  ?  -1  :  un_link(from)
#           else
                    rename(from, to) != 0
#                   if has_NFS
                        && errno != ENOENT
#                   endif
                ?  -1
#               if bad_a_rename
                :  mode != mode_while_renaming  ?  chmod(to, m
```

```
C1:  1 - 164
     169 - 179
     210 - 210
     238 - 252
     413 - 427
     486 - 488
     532 - 598
     654 - 659
     666 - 667
     674 - 684
     690 - 691
     695 - 702
     708 - 713
     725 - 727
     731 - 731
     747 - 749
     753 - 754
     758 - 1049
     1096 - 1121
     1128 - 1142
     1150 - 1228
     1234 - 1316
     1322 - 1328
     1347 - 1370
     1377 - 1385
     1396 - 1396
     1401 - 1402
     1406 - 1408
     1419 - 1420
     1434 - 1543
     1549 - 1656
C2:  !has_setuid
     1547 - 1547
C3:  has_rename
     1424 - 1424
     1428 - 1428
     1432 - 1432
C4:  !bad_a_rename
     1394 - 1394
C5:  !has_prototypes
     1375 - 1375
C6:  !has_mktemp
     1320 - 1320
     1336 - 1345
C7:  open_can_creat
     1232 - 1232
C8:  !has_readlink
     1125 - 1126
C9:  has_setuid
     1545 - 1545
C10: (!has_rename OR bad_b_rename)
     1410 - 1417
C11: has_fchmod
     1398 - 1399
     1404 - 1404
C12: bad_a_rename
     1387 - 1392
     C13:1430 - 1430
C14: has_prototypes
     1372 - 1373
C15: has_mktemp
     1318 - 1318
     1330 - 1334
C16: !open_can_creat
     1230 - 1230
C17: has_readlink
     1051 - 1094
     1123 - 1123
     1144 - 1148
C18: large_memory
     254 - 254
     277 - 400
     490 - 490
     608 - 652
     661 - 661
     693 - 693
     715 - 715
     729 - 729
     751 - 751
C19: !has_memmove
     258 - 275
C20: has_memmove
     256 - 256
C21: !has_NFS
C22: !has_rename
     1422 - 1422
C23: 213 - 213
C24: (bad_unlink OR has_NFS)
     181 - 188
     208 - 208
C25: 206 - 206
C26: has_NFS
     204 - 204
C27: 1426 - 1426
C28: 215 - 235
C29: bad_unlink
     190 - 193
     198 - 201
C30: 195 - 196
C31: !large_memory
     166 - 167
     402 - 405
     410 - 411
     429 - 484
     492 - 530
     600 - 606
     663 - 664
     669 - 672
     686 - 688
     704 - 706
     717 - 723
     733 - 745
     756 - 756
C32: bad_fopen_wplus
     407 - 408
C33:
```

Figure 6: Configuration lattice for `rcsedit`

but is so rare that nobody seems to be worried about it, we won't worry either."

A good configuration lattice is *horizontally decomposable*: it consists of independent sublattices, which are connected only via the top and bottom elements. Figure 7 presents a table which leads to a horizontally decomposable lattice, but also contains an interference. Since row and column permutations do not influence the lattice, horizontal decomposition (if possible) is a simple and natural way to discover independent configuration subspaces, which would be difficult in the table (or source file) directly. If all CPP symbols in a sublattice deal with the same configuration aspect, the principles of high cohesion and low coupling are satisfied. Interferences destroy horizontal decomposability, but can be discovered automatically. Interferences and their discovery using concept analysis are described in detail in [14] and [3].

4.3 Background knowledge and minimal governing expressions

Often the user has some background knowledge, such as "X_win requires UNIX" or "X_win and DOS are incompatible". Such knowledge can be coded as implications: $X\_win \rightarrow UNIX$ resp. $X\_win \wedge DOS \rightarrow \mathcal{A}$ (because a contradiction implies everything). It can easily be checked whether background implications are respected by a configuration table and its lattice. If not, the source file is inconsistent with the background knowledge, and the lattice displays those code pieces

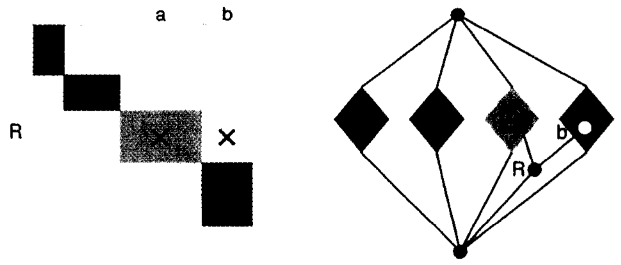


Figure 7: A horizontal decomposition and an interference

which violate background implications.

Background knowledge can also be used to simplify governing expressions in a way not achievable by boolean simplification alone. Lindig [9] not only found a simple and efficient way to determine the total number of configurations from the lattice; he also demonstrated how minimal governing expressions can be generated from irreducible lattice elements. Background knowledge can be added to this process: governing expressions are minimized with respect to program-specific or background-specific mutual dependencies. This results in a powerful tool for "software geriatrics", as demanded by Parnas.

5 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 $\mathcal{O}$ are the procedures of a program, the attributes $\mathcal{A}$ are the global variables, and the *variable usage table* has entry (p, v) if procedure p uses variable v.

5.1 Modules and lattices

A module consists of a set of procedures $P \subseteq \mathcal{O}$ and a set of variables $V \subseteq \mathcal{A}$ 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.

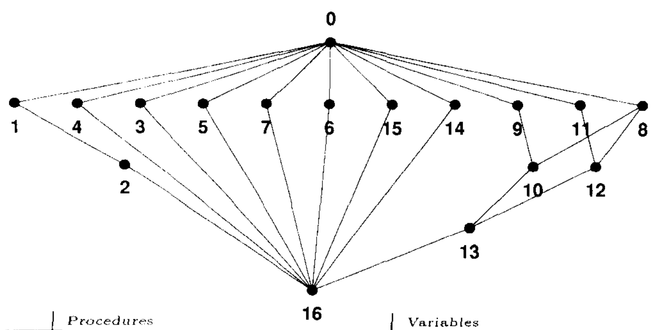
	angelegt	gelöscht	speicherverbrauch	colors	maxstrlength	namehashtab	phonehashtab	hashtabsize	error1	error2	esc
allocate	×										
init	×	×									
analyse	×	×									
initsp			×								
ausgabesp			×								
fuegespein			×								
changeadr				×							
readdata					×						
readline					×						
lookup						×					
exists						×					
rlookup							×				
remove						×	×				
insert						×	×				
calchasvalue								×			
savehashtab						×		×			
partsearch						×		×			
clearhashtabs						×		×			
inithashtabs						×	×	×			
printmessage									×	×	
setbackground											×
settextcolor											×
setattribute											×
clrsrc											×
gotoxy											×

Figure 8: Variable usage table of student Modula-2 program (excerpt)

Figure 8 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 (Figure 9) is of course horizontally decomposable[4]. Note that there are more horizontal summands than modules in the program. Thus the modularization proposal generated from the variable usage does not agree with the actual module structure in the program. Manual inspection confirms that some modules have low cohesion and should be split. For example, elements 3 and 4 have been one module in the original program, but the lattice indicates that element 4 implements an abstract data type, while element 3 deals with some low-level memory management.

In case there are only a few interferences between horizontal summands, the modular structure is still good. Interferences can be detected automatically and removed by simple program transformations such as encapsulation of global variables. If there are too many interferences, one might still try to modularize the source code by using so-called block relations. Block relations (also called weak congruences) correspond to rectangle shapes in the table and induce a factor lattice. Every

[4]For modular languages, this is a consequence of the theory. Unfortunately, Modula-2 allows to export module variables, which can lead to coupling and interferences.



	Procedures	*Variables*
0		
1	allocate	angelegt
2	init, analyse	geloescht
3	initsp, ausgabesp, fuegespein	speicherverbrauch
4	ermittelsp, loeschenicht, loeschealles, loesche, einfuegen, erzeuge, elemanz	pointerliste
5	addressinput	done, helpadr, help, eingabe
6	changeadr	colors
7	readdata, readline	maxstrlength
8	lookup, exists	namehashtab
9	rlookup	phonehashtab
10	remove, insert	
11	calchashvalue	hashtabsize
12	savehashtab, partsearch, clearhashtabs	
13	inithashtabs	
14	printmessage	error2, error1
15	setbackground, settextcolor, setattribute, clrscr, gotoxy	esc
16		

Figure 9: Module structure of a student Modula-2 program

element in the factor lattice corresponds to a rectangle shape in the original table and thus a module candidate. While block relations are difficult to detect manually in a table or its lattice, an efficient algorithm for dicovering block relations exists.

Figure 10 presents an example. The table entries can be grouped into three rectangle shapes, as can be seen from the additional bullet entries. The lattice resulting from the "enriched" table (original entries plus bullets) therefore has just three elements, which can be considered a "skeleton" of the original lattice. Indeed, the original lattice can be grouped into three overlapping "congruence classes". The factor lattice is just the three-element skeleton, hence there are three module candidates in the source code.

Another approach was studied by Siff and Reps [13]. 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.

5.2 A case study

We examined several legacy systems written in FORTRAN and COBOL. One example is an aerodynamics system used for airplane development in a national re-

	$V1$	$V2$	$V3$	$V4$	$V5$	$V6$
$R1$	●	×	×	×	×	×
$R2$	×	●	×	×	●	×
$R3$	×	●	×	●	×	×
$R4$	×	×	×	●	●	×
$R5$		×	●	×	×	×
$R6$		●		×	×	●

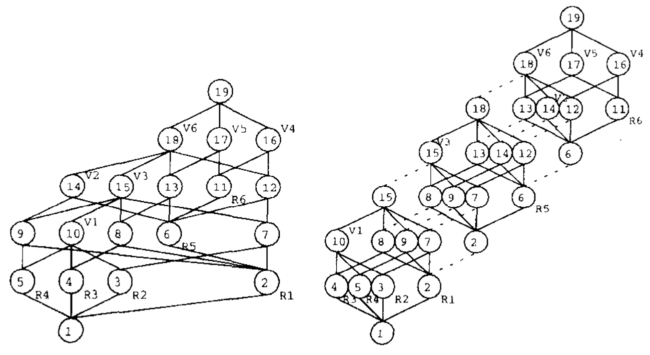


Figure 10: A context table, its lattice, a block relation, and its corresponding congruence classes

search 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[5]. 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 11). There is no way to make the lattice horizontally decomposable by removing just a small number of interferences.

Several experiments tried to analyse just part of the system. The program contains a particularly intricate COMMON block called "CNTL", which contains 26 variables. These variables are used in 192 subroutines, and the resulting lattice does not look very encouraging either. Another experiment examined the "OUTPUT"-subsystem, which consists of 50 subroutines using 278 global variables from 26 COMMON blocks; the resulting lattice still has 259 elements and is full of fine-grained interferences.

We also tried to determine block relations. Unfortunately, neither the lattice for the whole system nor the lattice for the "CNTL" COMMON block had usable block relations, hence no automatic modularization was pos-

[5]This required 11 seconds on a SparcStation20.

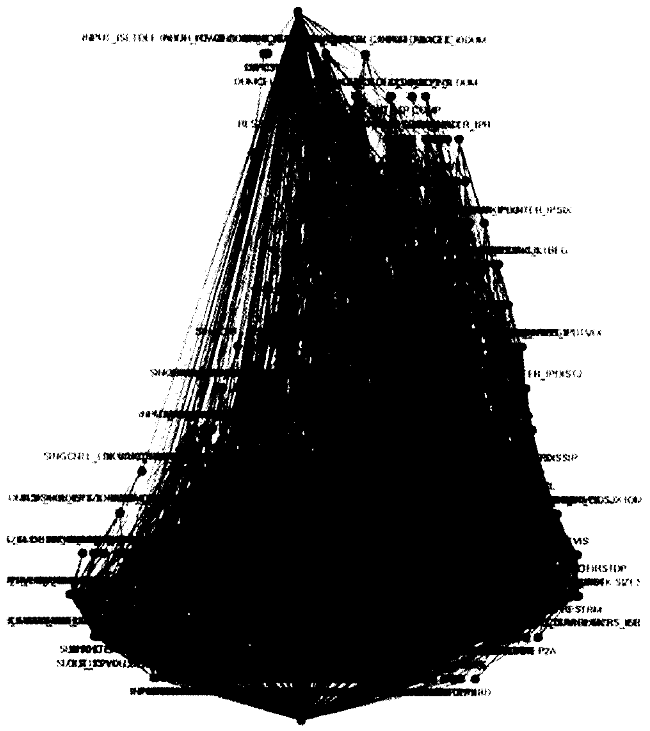


Figure 11: Module structure of aerodynamics system

sible. We also tried to apply subdirect decomposition [16] and subtensorial decomposition [17], as described in [3]. These decomposition techniques are motivated by algebraic rather than software engineering issues, and failed also.

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 or block decomposition. In the example, the overwhelming number of interferences 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 system from scratch.

6 Future work

The potential of concept analysis in program understanding has not yet been fully explored. Here are some plans for future work:

- A module can also be characterized by variables which are not used. Hence any module corresponds to a maximal unfilled rectangle. The lattice of the inverted context displays all these rectangles.
- The structure theory for concept lattices offers algebraic decompositions which have not been explored yet.
- Fuzzy contexts use values between 0 and 1 as table entries. For such tables, a concept lattice can again be computed [11].
- Concept analysis can be used for the analysis and perhaps reengineering of class hierarchies in old C++ programs.

References

[1] G. Birkhoff: Lattice Theory. American Mathematical Society, Providence, R.I., 1st edition, 1940.

[2] B. Davey, H. Priestley: Introduction to lattices and order. Cambridge University Press 1990.

[3] P. Funk, A. Lewien, G. Snelting: Algorithms for concept lattice decomposition and their application. Report 95-09, Computer Science Department, Technische Universität Braunschweig, 1995.

[4] R. Godin, R. Missaoui: An incremental concept formation approach for learning from databases. Theoretical Computer Science 133 (1994), pp. 387 – 419.

[5] M. Krone, G. Snelting: On the inference of configuration structures from source code. Proc. 16th International Conference on Software Engineering, Mai 1994, IEEE Comp. Soc. Press, pp. 49-57.

[6] C. Lindig: Concept-Based Component Retrieval. Proc. IJCAI-95 Workshop on Formal Approaches to the Reuse of Plans, Proofs, and Programs, Montreal, August 1995.

[7] C. Lindig: Komponentensuche mit Begriffen. Proc. Softwaretechnik '95, Braunschweig, Oktober 1995, S. 67-75.

[8] C. Lindig, G. Snelting: Assessing Modular Structure of Legacy Code Based on Mathematical Concept Analysis. Proc. International Conference on Software Engineering (ICSE'97), Boston 1997, pp. 349 – 359.

[9] C. Lindig: Analyse von Softwarevarianten. Report 98-03, Computer Science Department, Technische Universität Braunschweig, 1998.

[10] D. Parnas: Software Aging. Proc. International Conference on Software Engineering (ICSE'97), Boston 1997, pp. 279-290.

[11] S. Polland: Fuzzy Begriffe - formale Begriffsanalyse unscharfer Daten. Springer Verlag 1997.

[12] R. Prieto-Diaz: Implementing faceted classification for software reuse. Journal of the ACM 34 (5), 1991, pp. 89 - 97.

[13] M. Siff, T. Reps: Identifying Modules via Concept Analysis. Proc. International Conference on Software Maintenance, Bari 1997, pp. 170 - 179.

[14] G. Snelting: Reengineering of configurations based on mathematical concept analysis. ACM Transactions on Software Engineering and Methodology 5,2 (April 1996), pp. 146-189.

[15] R. Wille: Restructuring lattice theory: an approach based on hierarchies of concepts. In: I. Rival, (Ed.), Ordered Sets, pp. 445-470, Reidel 1982.

[16] R. Wille: Subdirect decomposition of concept lattices. Algebra Universalis 17 (1993), pp. 275-287.

[17] R. Wille: Tensorial decomposition of concept lattices. Order 2 (1985), pp. 81-95.

[18] R. Wille: Bedeutungen von Begriffsverbänden. In B. Ganter, R. Wille, K. Wolff (Ed.): Beiträge zur Begriffsanalyse. BI Wissenschaftsverlag 1996, pp. 161 - 212.

[19] B. Ganter, R. Wille: Formale Begriffsanalyse - Mathematische Grundlagen. Springer Verlag 1996.