On the Inference of Configuration Structures from Source Code

Research Paper

Maren Krone and Gregor Snelting
Arbeitsgruppe Softwaretechnologie
Technische Universität Braunschweig
Gaußstraße 17, D-38106 Braunschweig

Abstract

We apply mathematical concept analysis *to the problem of infering configuration structures from existing source code. Concept analysis has been developed by German mathematicians over the last years; it can be seen as a discrete analogon to Fourier analysis. Based on this theory, our tool will accept source code, where configuration-specific statements are controlled by the preprocessor. The algorithm will compute a so-called concept lattice, which – when visually displayed – allows remarkable insight into the structure and properties of possible configurations. The lattice not only displays fine-grained dependencies between configuration threads, but also visualizes the overall quality of configuration structures according to software engineering principles. The paper presents a short introduction to concept analysis, as well as experimental results on various programs.*

1 Introduction

A simple and widely used technique for configuration management is the use of the C preprocessor. Configuration-dependent source code pieces are enclosed in "#ifdef ... #endif" brackets, and by defining preprocessor symbols during compiler invocation (e.g. "cc –Dultrix prog.c"), a configuration thread is determined and the appropriate code pieces are selected and compiled. Although much more sophisticated configuration management systems have been developed recently (see e.g. [2]), a lot of code sticking to "configuration management by preprocessing" is around, and a reverse engineering tool which allows to extract the underlying structure from such sources is certainly useful.

As an example, consider some code pieces from the X-Window tool "xload"; this tool displays various machine load factors (figure 1). The 724–line program is quite platform dependent: 43 preprocessor symbols are used to control a variety of configuration threads (e.g. SYSV, macII, ultrix, sun, CRAY, sony). A code piece may depend not only on simple preprocessor symbols, but on arbitrary boolean combinations of such symbols. Furthermore,

```
#if (!defined(SVR4) || !defined(__STDC__)) && !defined(sgi) &&
!defined(MOTOROLA)
    extern void nlist();
#endif
#ifdef AIXV3
    knlist( namelist, 1, sizeof(struct nlist));
#else
    nlist( KERNEL_FILE, namelist);
#endif
#ifdef hcx

    if (namelist[LOADAV].n_type == 0 &&
#else
    if (namelist[LOADAV].n_type == 0 ||
#endif /* hcx */
        namelist[LOADAV].n_value == 0) {
    xload_error("cannot get name list from", KERNEL_FILE);
    exit(-1);
    }
    loadavg_seek = namelist[LOADAV].n_value;
#if defined(umips) && defined(SYSTYPE_SYSV)
    loadavg_seek &= 0x7fffffff;
#endif /* umips && SYSTYPE_SYSV */
#if (defined(CRAY) && defined(SYSINFO))
    loadavg_seek += ((char *) (((struct sysinfo *)NULL)->avenrun))
- ((char *) NULL);
#endif /* CRAY && SYSINFO */
    kmem = open(KMEM_FILE, O_RDONLY);
    if (kmem < 0) xload_error("cannot open", KMEM_FILE);
#endif
```

Figure 1: X-Window tool "x_load.c"

"#ifdef"s and "#define"s may be nested, resulting in rather incomprehensible source texts. Even experienced programmers will have difficulties to obtain some insight into the configuration structure, and when a new configuration variant is to be covered, the introduction of errors is very likely.

Fortunately, there is a method, called *formal concept analysis* [13,15], which allows to reconstruct semantic structures from raw data as given in our case. This method has been developed at the universal algebra group in the Department of Mathematics at the Technical University of Darmstadt, and has been applied to various problem domains such as classification of finite lattices, analysis of Rembrandt's paintings, or behaviour of drug addicts. The method computes a so-called *concept lattice*, where a concept is a pair, consisting – in our case – of a set of code pieces (so-called *objects*) and a set of preprocessor symbols (so-called *attributes*). Such concepts represent semantic properties of

the underlying problem domain. The lattice structure imposes a partial order on concepts (more specific vs. more general), and for two concepts, there exist supremum (generalization) and infimum (unification).

A concept lattice which arises from a source text similar to “x_load.c” is presented in figure 2[1]. It reveals simple facts e.g. that the CRAY configuration comprises source lines 21–28, 29–40, 201–207, and 11–20. But it also displays less obvious information, e.g. that there are three main configuration schemes (macII, SYSV, sun); that lines 11–20 appear in all configurations except sequent, alliant, and 386 platforms; that apollo and ultrix configurations have lines 126–200, 201–207, 11–20 in common; and that source lines valid for sony or ultrix are valid for sun as well. Furthermore, violations of software engineering principles like high cohesion or low coupling show up immediately.

[1] Figure 2 and figure 3 are isomorphic copies of an instructive example presented in [15]

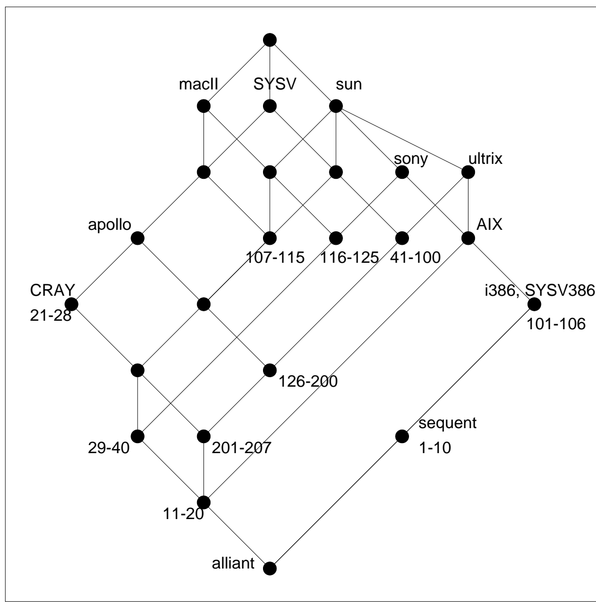


Figure 2: A concept lattice

All that is a consequence of the concept lattice structure, as explained in the paper.

2 Basic Notions of Concept Analysis

2.1 The concept lattice

Formal concept analysis has been introduced by R. Wille about ten years ago. For beginners, it is not that easy to understand, hence we restrict ourselves to the absolute minimum of the theory. Formal concept analysis starts with a triple $C = (O, A, P)$, called a (formal) *context*, where O is a finite set (the so-called *objects*), A is a finite set (the so-called *attributes*), and P is a relation between O and A, hence $P \subseteq O \times A$. If $(o, a) \in P$, we say object o has attribute a. Figure 3 gives an example of a formal context, namely a characterization of source lines by governing preprocessor symbols, as extracted from a program's source text.

For a set of objects $X \subseteq O$, we define the set of *common attributes* $\sigma(X) := \{a \in A \mid \forall o \in X : (o, a) \in P\}$. Similarly, for a set of attributes $Y \subseteq A$ the *common objects* are defined by $\tau(Y) := \{o \in O \mid \forall a \in Y : (o, a) \in P\}$. The mappings $\sigma : 2^O \to 2^A$ and $\tau : 2^A \to 2^O$ form a *Galois connection* and can be characterized by the following conditions: for $X, X_1, X_2 \subseteq O,\ Y, Y_1, Y_2 \subseteq A$

$$X_1 \subseteq X_2 \Longrightarrow \sigma(X_2) \subseteq \sigma(X_1)$$

and

$$Y_1 \subseteq Y_2 \Longrightarrow \tau(Y_2) \subseteq \tau(Y_1)$$

that is, both mappings are *antimonotone*;

$$X \subseteq \tau(\sigma(X)) \text{ and } \sigma(X) = \sigma(\tau(\sigma(X)))$$

as well as

$$Y \subseteq \sigma(\tau(Y)) \text{ and } \tau(Y) = \tau(\sigma(\tau(Y)))$$

	SYSV	SYSV386	macII	i386	ultrix	sun	AIX	CRAY	apollo	sony	sequent	alliant
1 - 10		X		X	X	X	X			X	X	
11 - 20	X		X		X	X	X	X	X	X		
21 - 28	X		X					X	X			
29 - 40	X		X			X		X	X	X		
41 - 100	X				X	X						
101 - 106		X		X	X	X	X			X		
107 - 115	X		X			X						
116 - 125			X			X				X		
126 - 200	X		X		X	X			X			
201 - 207	X		X		X	X		X	X			

Figure 3: Sample classification of source lines according to governing preprocessor symbols

that is, both mappings are *extensive*, in particular the common objects of the common attributes of an object set are a superset of this object set, and their common attributes are equal; finally, for an index set I and $X_i \subseteq O, Y_i \subseteq A$

$$\sigma\big(\bigcup_{i\in I} X_i\big) = \bigcap_{i\in I} \sigma(X_i) \quad \text{and} \quad \tau\big(\bigcup_{i\in I} Y_i\big) = \bigcap_{i\in I} \tau(Y_i)$$

A (formal) *concept* is a pair (X, Y), where $X \subseteq O, Y \subseteq A$ and $Y = \sigma(X), X = \tau(Y)$. Hence, a concept is characterized by a set of objects (called its *extent*) and a set of attributes (called its *intent*) such that all objects have all attributes and all attributes fit to all objects. The set of all concepts is denoted by $B(O, A, P)$. Intuitively, a concept is a maximal filled rectangle in a table like figure 3, where permutations of lines or columns of course do not matter.

A concept (X_1, Y_1) is a *subconcept* of another concept (X_2, Y_2) if $X_1 \subseteq X_2$ (or, equivalently, $Y_1 \supseteq Y_2$). It is easy to see that this definition imposes a partial order on $B(O, A, P)$, thus we write $(X_1, Y_1) \leq (X_2, Y_2)$. Moreover, $\underline{B}(O, A, P) = (B(O, A, P), \leq)$ is a complete lattice, due to the

Basic Theorem for Concept Lattices [13]: Let $C = (O, A, P)$ be a context. Then $\underline{B}(O, A, P)$ is a complete lattice, called the *concept lattice* of C, for which infimum and supremum are given by

$$\bigwedge_{i\in I}(X_i, Y_i) = \Big(\bigcap_{i\in I} X_i, \sigma\big(\tau\big(\bigcup_{i\in I} Y_i\big)\big)\Big)$$

and

$$\bigvee_{i\in I}(X_i, Y_i) = \Big(\tau\big(\sigma\big(\bigcup_{i\in I} X_i\big)\big), \bigcap_{i\in I} Y_i, \Big)$$

This remarkable theorem says that in order to compute the infimum (greatest common subconcept) of two concepts, their extents must be intersected and their intents must be joined; the latter set of attributes must then be "blown up" in order to fit to the object set of the infimum. Analogeously, the supremum (smallest common superconcept) of two concepts is computed by intersecting the attributes and joining the objects.

The lattice structure allows a *labelling* of the concepts: a concept is labelled with an object, if it is the smallest concept in the lattice subsuming that object; a concept is labelled with an attribute, if it is the largest concept subsuming that attribute. In fact, a concept c labelled with an object o is of the form $c = \big(\tau(\sigma(\{o\})), \sigma(\{o\})\big)$, and if c is labelled with attribute a, it is of the form $c = \big(\tau(\{a\}), \sigma(\tau(\{a\}))\big)$. Utilizing this labelling, the extent of c can be obtained by collecting all objects which appear as labels on concepts *below* c, and the intent of c is obtained by collecting all attributes which appear *above* c.

For two attribute sets A and B we say "A implies B" (written $A \Rightarrow B$) if $\tau(A) \subseteq \tau(B)$ (or equivalently, if $B \subseteq \sigma(\tau(A))$). This can be read as "any object having all attributes in A also has all attributes in B". Now if A and B constitute two concepts $C = \big(\tau(A), A\big)$ and $D = \big(\tau(B), B\big)$, and $C \leq D$, then $A \Rightarrow B$ obviously holds.

The concept lattice can be considered as a graph, that is, a relation. What happens if we again apply concept analysis to this derived relation? It turns out that the concept lattice reproduces itself [1]! Thus concepts do not "breed" new concepts; there is no proliferation of virtual information.

There is much more to say about concept lattices, but for the purposes of this paper, the basic theorem suffices. The interested reader should consult [1], which contains a chapter on concept analysis. We conclude this section with the remark that there are several algorithms which actually compute a concept lattice (see [3]); the typical time complexity is $O(n^3)$, where $n = max(\mid O \mid, \mid A \mid)$.

2.2 Interpretation of context lattices

Let us apply the basic theorem to the context table of figure 3 and its concept lattice, given in figure 2. In order to get a feeling what kind of insight can be obtained from such a lattice, we first remember that a subconcept of a concept has a smaller object set, but (note the symmetry) a larger attribute set. That is, if we go down in the lattice, we get more precise information about smaller object sets.

The above-mentioned labelling allows a concise characterisation of concepts. For example, the concept labelled CRAY is in fact the concept ({11-20, 21-28, 29-40, 201-207},{CRAY, apollo, macII, SYSV}). And indeed, figure 3 reveals that this concept is a rectangle in the context table. Hence, we have inferred that the lines 11-20, 21-28, 29-40, and 201-207 characterize the CRAY, apollo, macII, and SYSV configurations (and vice versa). The concept labelled apollo stands for ({11-20, 21-28, 29-40, 126-200, 201-207},{apollo, macII, SYSV}), which again is a rectangle in the context table, higher but leaner than the first one: CRAY$\leq$ apollo. Thus, the CRAY configuration comprises lines 11-20, 21-28, 29-40, 201-207 (and no other), but these lines appear in the apollo configuration as well.

This example already demonstrates one possibility to interpret a concept lattice: it can be seen as a *hierarchical conceptional clustering* of objects. Objects are grouped into sets and the lattice structure imposes a taxonomy on these object sets.

If we want to know what an apollo and an ultrix configuration have in common, we look at the infimum in the lattice, which is labelled 126–200; going down we see that lines 126–200, 201–207 and 11–20 appear in both configurations. On the other hand, if we want to see which attributes govern both lines 126–200 and 101–106, we look at the supremum of the corresponding concepts, which is ultrix; going up, we see that the sun and the ultrix configurations (and no other) will include both code pieces.

Upward arcs in the lattice diagram can be interpreted as *implications*: "*If a code piece appears in the sony or ultrix configuration, it will appear in the sun configuration as well*". Such knowledge is not easily extracted by hand from a source file like "x_load.c"! This example demonstrates the second main possibility to interpret a concept lattice: it represents all implications (that is, *dependencies*) between sets of attributes.

The original context can always be reconstructed from the lattice, e.g. the column for i386 has entries for all objects below concept i386, namely 1–10,101–106 whereas the row labelled 41–100 has entries for all attributes above, namely sun, SYSV, and ultrix. Hence, a context (i.e. relation) and its concept lattice are analogeous to a function and its Fourier transform (which also can be reconstructed from each other): concept analysis is similar in spirit to spectral analysis of continuous signals.

3 The Reverse Engineering Tool

We have developed a tool which implements the approach described in the previous sections. This tool accepts source code as input and produces a graphical display of the concept lattice as output. The source language is arbitrary, but the input file must stick to the conventions of the C preprocessor. Our tool consists of the following phases:

1. front end: the front end separates code pieces and preprocessor statements, syntactically analyses the latter, and constructs the context table according to the rules described below.
2. kernel: the kernel is is a software package developed by P. Burmeister in Darmstadt; it reads a context table and computes the corresponding concept lattice.
3. back end: the back end accepts a description of the concept lattice and produces a graphical display.

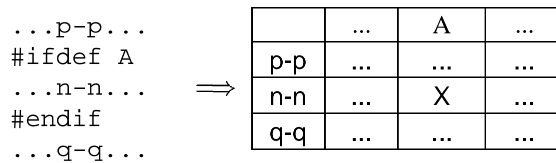
As usual, our tool is invoked as a UNIX command with the source file name as a parameter; additional options which control some display parameters may be added.

3.1 Construction of the context table

In our application of concept analysis, code pieces are not only governed by simple preprocessor symbols, but also by complex expressions, e.g.
`#if defined(A)||defined(B)&&defined(C)`
We will now describe the treatment of such expressions.

After syntax analysis, the context table is constructed according to the following semiformal rules (A, B, C denote preprocessor symbols, p-p, n-n, q-q denote code pieces).

- The basic rule for code pieces governed by single preprocessor symbols is:



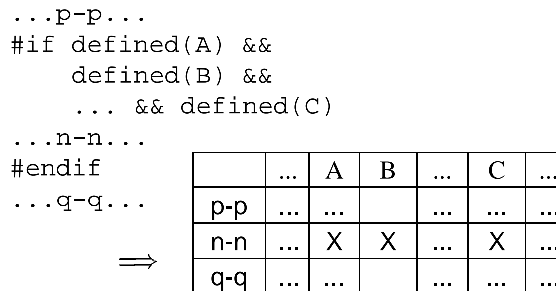
```
...p-p...
#ifdef A
...n-n...
#endif
...q-q...
```

⟹

	...	A	...
p-p	...	...	...
n-n	...	X	...
q-q	...	...	...

Of course, `#if defined(A)` has the same meaning as `#ifdef A`.

- If a code piece is governed by a conjunction of preprocessor symbols, the rule is:



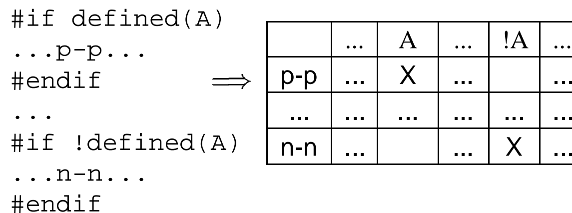
```
...p-p...
#if defined(A) &&
    defined(B) &&
    ... && defined(C)
...n-n...
#endif
...q-q...
```

⟹

	...	A	B	...	C	...
p-p	...	...		...	...	...
n-n	...	X	X	...	X	...
q-q	...	...		...	...	...

This is correct, since a set of columns in a formal context is itself a conjunction of single columns.

- If a symbol occurs in negated form, this negated symbol needs a column of its own, since a basic formal context can express only positive statements. The rule thus is:



```
#if defined(A)
...p-p...
#endif
...
#if !defined(A)
...n-n...
#endif
```

⟹

	...	A	...	!A	...
p-p	...	X	...		...
...	...	...	...	...	...
n-n	...		...	X	...

A similar rule applies to
`#ifdef ... #else ... #endif`
In the theory of concept lattices, the resulting table is called the "dichotomised context". Prolog programmers have known the same trick (explicit rules for negated predicates) for a long time.

- Disjunctions of symbols are a little bit more complicated. The basic idea is as follows: In order to handle `#if defined(A) || defined(B)`, we introduce a separate column for $A \vee B$. As both A and B imply $A \vee B$, we must therefore place a cross in the $A \vee B$ column whenever we place a cross in the column for A or B. The basic rule for disjunctions hence is:

```
#if defined(A)
...p-p...
#endif
#if defined(A) || defined(B)
...n-n...
#endif
#if defined(B)
...q-q...
#endif
```

⟹

	...	A	B	...	A‖B	...
p-p	...	X	...	...	X	...
n-n	...		...	...	X	...
q-q	...	...	X	...	X	...

Simple disjunctions show up as suprema in the concept lattice. In case there are complex conditions arbitrarily built up from conjunctions, disjunctions and negations, these are first transformed into conjunctive normal form by applying the distributive and de Morgan laws. Afterwards, all expressions are of the form $(A_1 \vee A_2 \vee ...A_i) \wedge (B_1 \vee B_2 \vee ...B_j) \wedge ...(C_1 \vee C_2 \vee ...C_k)$, where all A_μ, B_ν, C_ρ are either simple symbols or negated symbols. Expressions in conjunctive normal form can then be treated by the above rules.

- Nested #ifdefs, #defines, and #undefs are treated as implications. For example, in

```
#ifdef A
...p-p...
#define B
#ifdef B
...n-n...
#endif
#endif
...q-q...
```

⟹

	...	A	B	...
p-p	...	X		...
n-n	...	X	X	...
q-q	...			...

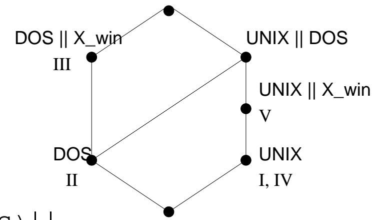
we must add a cross in the “B”-column whenever we place a cross in the “A”-column; a similar mechanism is used for “#undef”s.

It should be noted that transforming an expression into an equivalent one (e.g. $A \vee A \wedge B \equiv A$) does not change the concept lattice. In particular, it is not necessary to use a *minimal* conjunctive normal form; any conjunctive normal form will do. Intuitively, the reason is that a concept is a *maximal* filled rectangle in a table.

3.2 A small example

Consider the source text and its corresponding concept lattice:

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



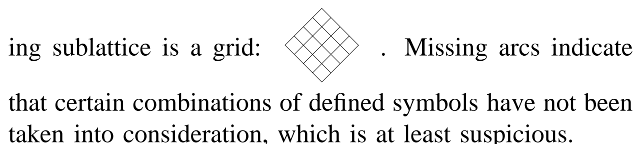
The lattice shows that code pieces I and IV are governed by UNIX, code piece II is governed by DOS, UNIX || X_win implies UNIX || DOS (which means that any code piece valid for X-windows is also valid for UNIX or DOS) etc. Such dependencies are not easy to see in more complicated sources, but nevertheless the reader might ask: so what? After all, we mentioned the analogy to spectral analysis, and using spectral analysis, astronomers have shown that the universe is expanding!

Although we cannot offer such spectacular insights, the lattice clearly shows that *the configuration structure is faulty*. Two important software engineering principles are *separation of concerns* and *anticipation of change*. For example, operating system issues should be separated from user interface issues, and it should be easy to incorporate another window system into a future version. The lattice shows that OS as well as UI issues show up in both main configuration threads, and that —worse— there is a cross dependency between them. Cross dependencies prevent the lattice from being decomposed into independent sublattices, and this shows there is low coherence and strong coupling between configuration threads. Hence, concept analysis not only provides a detailed account of all dependencies, but can serve as a quality assurance tool in order to check for good design of the configuration structure, or to limit entropy increase as a software system evolves.

In general, *low coupling* of configuration threads is achieved when “semantically different” preprocessor sym-

bols appear in disjoint sublattices: .

Paths which are glued together in their top or bottom sections are acceptable, but cross arcs between sublattices always indicate interference between orthogonal configuration threads.

High cohesion is achieved, if, for a subset of preprocessor symbols in the same semantic (sub)domain, the corresponding sublattice is a grid:  . Missing arcs indicate that certain combinations of defined symbols have not been taken into consideration, which is at least suspicious.

Unfortunately, only a human can decide whether preprocessor symbols are "semantical neighbours". Usually, the names of the preprocessor symbols indicate their meaning. This helps to interpret the lattice, but nevertheless certain experience is needed.

3.3 Data Reduction

Often one would like to obtain a quick overview of the configuration structure and explore the full details later. For such purposes, two simple data reduction techniques have been implemented.

First, the user may specify a maximal nesting depth for nested "ifdef"s. All #ifdefs and #defines which are more deeply nested are ignored. This results in a concept lattice which displays only the overall structure of possible configurations, ignoring fine-grained details.

The second technique is based on the observation that certain code pieces are often governed by almost identical preprocessor settings. The corresponding rows in the context table can be merged into one row if they "do not differ too much". The user may specify a threshold value t, and if a set of rows can be identified where all rows do pairwise differ in less than t positions, these rows are replaced by a new row which has crosses in a column if *all* original rows had. Such a "multirow" thus describes a set of code pieces such that all code pieces have at least all attributes which are marked (but some may have more). This gives us a conservative approximation (we loose some dependencies, but we never introduce false ones). In the concept lattice, the technique has the effect that several concepts are merged into one concept: row merging induces a lattice congruence and hence is compatible with supremum and infimum.

3.4 Graphical Display

It is a non-trivial task to display the concept lattice in such a way that interesting properties show up immediately. In fact, a number of sophisticated algorithms has been devised for that purpose [8,14]. Some of the techniques used are to embed lattices into grids, or to present the lattice as a (sub)direct product of smaller lattices. Such techniques allow to detect e.g. the automorphisms of the lattice, or to check whether the lattice is distributive.

Some of these algorithms have been implemented, but were not available to us. Thus, we use a simpler approach, based on the Sugiyama algorithm [11]. This well-known layout algorithm for arbitrary directed graphs uses the topological ordering of nodes in order to determine their vertical position. As the results are not always completely satisfactory, the user may finally change the graph layout manually (but the system will maintain integrity of the concept lattice).

4 Experimental Results

We applied our tool to several UNIX programs. The reader should keep in mind that the graph displays below have been produced by a program which has been developed for a different purpose, hence the layout is not optimal for concept analysis. We plan to integrate the more sophisticated display algorithms sketched above in the public–domain ftp version.

Our first example is a popular shell, the "tcshell" developed at Berkeley. We have analysed one of its modules, namely "sh.exec.c". This program is 959 lines long and uses 24 different preprocessor symbols. In the concept lattice (figure 4), singleton attribute or object labels are displayed in the diagram, the others can be looked up in a separate window through the concept name "Cnn". It turns out that the configuration structure is perfect according to the criteria described above. It seems that there is an interference between the path including C15 and the concepts C15 – C18, C20. But a look at the source code reveals that both VFORK and FASTHASH have to do with the hash function used, hence there are no dependencies between orthogonal configuration concepts.

Our second example is the stream editor from the RCS system "rcsedit.c" [12]. This 1656–line program uses 21 preprocessor symbols. The concept lattice is shown in figure 5, together with 25 lines of source code (beginning with line 179) and the labelling of the concepts. The concepts below C6 (which concern different file access variants) as well as those below C8 (C8 is labelled "large_memory") have a simple structure, and the concepts below C9/C10 (concerning networking) form a grid-like cluster. But there is an interference manifest in C27, which is the infimum of

C38 and C15. C38 is labelled has_NFS, C15 is labelled has_rename, and C27 is labelled 1425 – 1427. Thus, lines 1425 – 1427 are governed by both has_NFS and has_rename. A similar interference shows up in C37.

Hence, although the overall structure is quite good, we suspect that networking issues and file access variants are not clearly separated in "rcsedit.c". And indeed: a comment in the source code explains that due to an NFS bug, "rename()" can in rare cases destroy the RCS file! This problem has been re-discovered by concept analysis, just by analysing the configuration structure. The example demonstrates that our tool can indeed track down bugs, even bugs which the programmers would like to keep covered: the last sentence of the comment reads "Since this problem afflicts scads of Unix programs, but is so rare that nobody seems to be worried about it, we won't worry either"[2].

Let us finally come back to our introductory example, "x_load.c" (see figure 1). This program is 724 lines long and uses 43 preprocessor symbols for configuration management. The resulting concept lattice has 141 concepts

[2] The problem is in fact a little bit more complicated; the interested reader should look at the source code himself

Figure 4: Configuration structure of tcshell module "sh.exec.c"

Figure 5: Configuration structure of the RCS stream editor

and is shown in figure 6. It looks pretty chaotic, and we therefore used data reduction to display only the top 4 #ifdef nesting levels (figure 7). Even on the top level, there are interferences (C19/C24), and the central role of C33 does not inspire confidence (C19 is the infimum of C2 and C11; C2 is SVR4 || UTEK || alliant || hex || sequent || sgi || sun, C11 is !apollo. C33 is is a set of 9 code pieces governed by the sundries SYSV386, !LOADSTUB and !KVM_ROUTINES). It seems that this program suffers from configuration hacking.

5 Conclusion

We described a tool for extracting configuration structures from existing source code. Our point of departure was "configuration management by preprocessing", but the method can easily be adapted to more modern configuration management techniques (e.g. shape [7]). It turned out that mathematical concept analysis is a powerful tool for gaining insight into configuration structures, just as Fourier analysis is for ordinary functions.

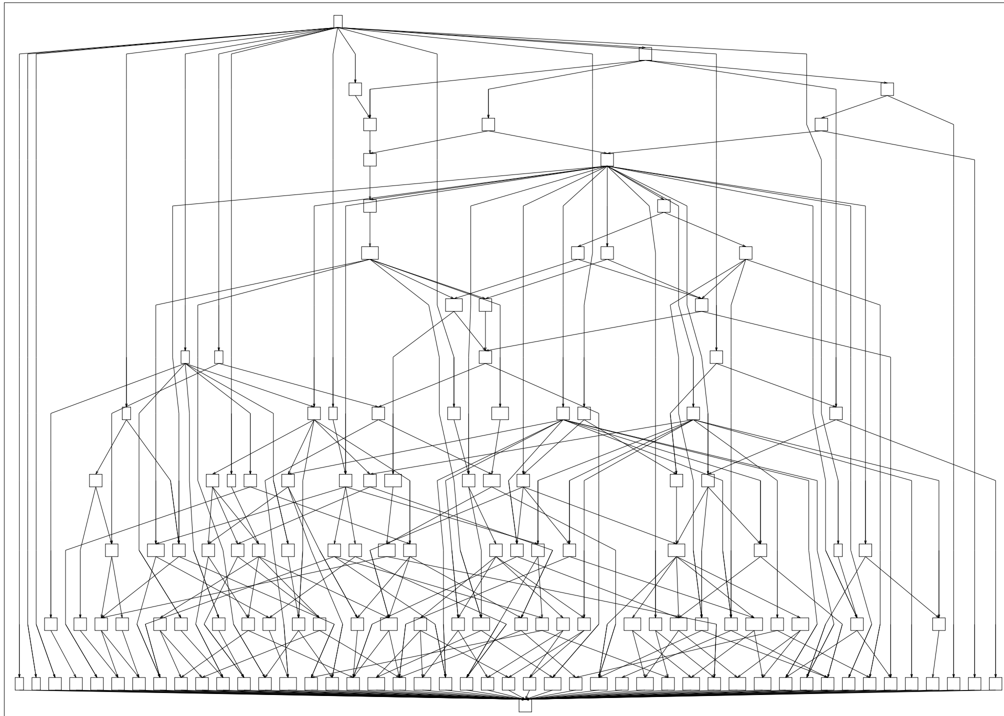


Figure 6: Configuration structure of "x_load.c"

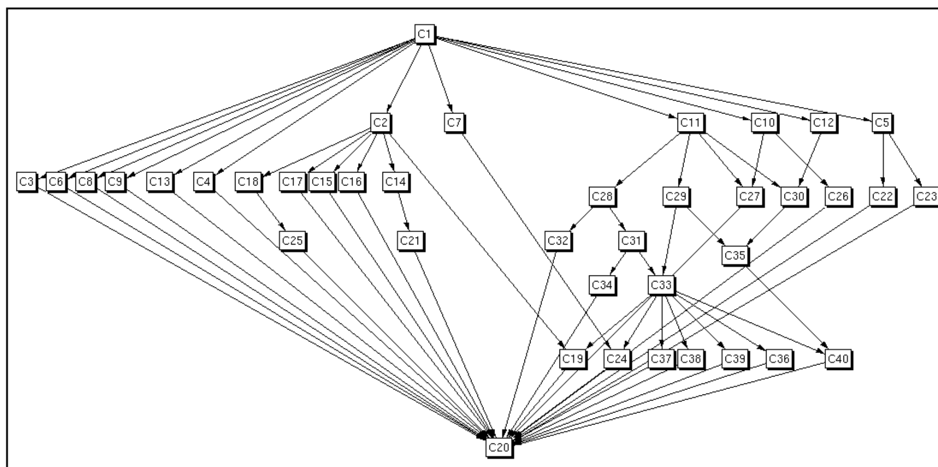


Figure 7: Top level configuration structure of "x_load.c"

It might very well be that concept analysis has other applications in reverse engineering; this should be investigated. There is an extension of the theory called *conceptual knowledge systems* [15] which allow to infer relationships between *user-defined* concepts (in our tool, concepts are generated automatically). We will investigate the usefulness of this extension to our problem. Another possible application is *restructuring* of configurations: by analysing and decomposing the concept lattice, hints for improving the configuration structure may be obtained.

Our tool is part of the inference-based software development environment NORA[3]. NORA aims at utilizing unification theory and inference technology in software tools; concepts and preliminary results can be found in [9,5,10].

The tool described in this paper can be obtained via anonymous ftp: `ftp.ips.cs.tu-bs.de` (134.169.32.1).

Acknowledgements. Andreas Zeller and Christian Lindig have been a great help with the graph display program. Martin Skorsky from the Darmstadt algebra group contributed several helpful comments. Peter Burmeister kindly made available his CONIMP program for concept analysis.

NORA is funded by the Deutsche Forschungsgemeinschaft, grants Sn11/1–2 and Sn11/2–1.

6 References

[1] Davey, B.A., Priestley, H.A.: Introduction to Lattices and Order. Cambridge University Press 1990.

[2] Feiler, P. (ed.): Proc. of the 3rd International Workshop on Software Configuration Management. ACM 1991.

[3] Ganter, B.: Algorithmen zur formalen Begriffsanalyse. In [4], pp. 241 – 254.

[4] Ganter, B., Wille, R., Wolff, K. (ed.): Beiträge zur Begriffsanalyse. B.I. Wissenschaftsverlag 1987.

[5] Grosch, F.-J., Snelting, G.: Polymorphic Components for Monomorphic Languages. Proc. Second International Workshop on Software Reusability. IEEE 1993, pp. 47 – 55.

[6] Krone, M.: Reverse Engineering of Configuration Structures. Master's thesis, TU Braunschweig, Institut für Programmiersprachen, 1993 (in German).

[7] Mahler, A. und Lampen, A.: An Integrated Toolset for Engineering Software Configurations. Proc. ACM Symposium on Practical Software Development Environments, SIGSOFT Notices 13, 5 (November 1988), pp. 191 – 200.

[8] Skorsky, M.: Endliche Verbände – Diagramme und Eigenschaften. PhD thesis, Technical University of Darmstadt, Dept. of Mathematics, 1992.

[9] Snelting, G., Grosch, F.-J., Schroeder, U.: Inference-Based Support for Programming in the Large. Proc. 3rd European Software Engineering Conference, Milano 1991. LNCS 550, pp. 396 – 408.

[10] Snelting, G., Zeller, A.: Inferenzbasierte Werkzeuge in NORA. Proc. Softwaretechnik '93, pp. 25 – 32, GI 1993 (in German).

[11] Sugiyama, K., Tagawa, S., Toda, M.: Methods for Visual Understanding of Hierarchical System Structures. IEEE Transaction on Systems, Man and Cybernetics 11, 2 (1981), pp. 109 – 125.

[12] Tichy, W. F.: RCS - A System for Version Control. Software Practice and Experience 15(7), pp. 637 – 654, Juli 1985.

[13] Wille, R.: Restructuring Lattice Theory: An Approach Based on Hierarchies of Concepts. In: I. Rival (ed.) Ordered Sets. Reidel 1982, pp. 445 – 470.

[14] Wille, R.: Geometric Representation of Concept Lattices. In: O. Opitz (ed.): Conceptual and Numerical Analysis of Data. Springer 1989, pp. 239 – 255.

[15] Wille, R.: Concept Lattices and Conceptual Knowledge Systems. Computers & Mathematics with Applications 23 (1992), pp. 493 – 515.

[3] NORA is a drama by the Norwegian writer H. IBSEN. Hence, NORA is no real acronym.