Reengineering of Configurations Based on Mathematical Concept Analysis

Gregor Snelting

Informatik-Bericht Nr. 95–02
Januar 1995

Technische Universität Braunschweig
Gaußstraße 17
D-38106 Braunschweig
West Germany

Reengineering of Configurations Based on Mathematical Concept Analysis

Gregor Snelting
Abteilung Softwaretechnologie
Technische Universität Braunschweig
Gaußstraße 17, D-38106 Braunschweig

Abstract

We apply *mathematical concept analysis* to the problem of reengineering configurations. Concept analysis has been developed by German mathematicians over the last years; it will reconstruct a taxonomy of concepts from a relation between objects and attributes. We use concept analysis to infer configuration structures from existing source code. Our tool NORA/RECS will accept source code, where configuration-specific code pieces 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.

In a second step, interferences between configuration threads can be analysed in order to restructure configurations. Interferences indicate high coupling and low cohesion between configuration concepts. They can be resolved by automatically decomposing the source code into "modules", where each module deals with a cohesive subset of the configuration space.

The paper presents experimental results on various programs. In particular, we consider some well-known Unix programs which suffer from configuration hacking.

Categories and Subject Descriptors: D.2.7 [Software Engineering] Distribution and Maintenance – restructuring, version control, D.2.6 [Software Engineering] Interactive programming environments, D.2.9 [Software Engineering] Software configuration management
General Terms: Design, Management, Theory
Additional Key Words and Phrases: concept analysis, concept lattices

1 Introduction

In his invited talk at the 16th International Conference on Software Engineering, David Parnas said "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" [Pa94]. Unfortunately, no method for understanding program families was as yet available, let alone tool support for restructuring.

At the same conference, we presented a first step towards a theory and tools for configuration restructuring [KS94]. Based on *mathematical concept analysis* [Wi82], we have shown how configuration structures can be infered from existing source code, and how interferences between configuration threads can be detected.

A preliminary version of parts of this paper appeared in the proceedings of the 16th International Conference on Software Engineering [KS94]

In this article, we continue our work and move on from inference of configuration structures to restructuring of configurations. We describe in detail how to extract configuration structures from existing source code, and how to interpret the obtained structures. We then present an algorithm for detecting interferences between configuration threads. An interference means that two configurations have common code, where they should not. A source file can be made interference free by dividing it into several modules, where each module covers a cohesive subset of the configuration space.

Before we begin to explain our restructuring tool NORA/RECS in detail, we would like to give an overview; this might also serve as an extended abstract for hurried readers.

1.1 Configuration management by preprocessing

In this paper, we assume that version selection and system building are done with the C preprocessor (CPP). Although much more sophisticated configuration management systems have been developed recently, a lot of code sticking to "configuration management by preprocessing" is around. Indeed, configuration management by preprocessing has been called an industry standard. Nevertheless, the methods explained in this paper can easily be adapted to more modern configuration management techniques.

Configuration management by preprocessing is very simple: 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. There are two basic methods how to use preprocessor symbols. The first method introduces a CPP symbol for every target configuration (e.g. AIX, SUN4, ULTRIX); this symbol must be defined if compiling for a specific target. Code common to several target configurations is enclosed in a disjunction of CPP symbols:

```
#if defined(SUN) || defined(ULTRIX) || defined(AIX)
...
#endif
```

The second method uses one CPP symbol for each feature of the target configuration (e.g. HAS_NFS, BSD, HAS_BCOPY); code requiring certain features is enclosed in a conjunction of CPP symbols:

```
#if defined(HAS_BCOPY) && defined(HAS_NFS)
...
#endif
```

Unfortunately, many programs mix both styles of preprocessor use. 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). Code pieces not only depend on simple preprocessor symbols, but on arbitrary boolean combinations of such symbols. Furthermore, "#ifdef"s and "#define"s are nested, resulting in a rather incomprehensible source text. 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.

1.2 Concept lattices

Fortunately, there is a method, called *formal concept analysis*, 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*, which is computed from a relation between *objects* and

```
#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 “xload.c”

```
...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	X		
III		X	
IV	X		X
VI			X
VI			

Figure 2: A small code fragment and its configuration table

attributes – in our case, from the relation between code pieces and governing preprocessor symbols (the so-called *configuration table*). Figure 2 presents a small code fragment and its configuration table.

A concept is a pair, consisting of a set of objects and a set of attributes such that all objects have all attributes and all attributes fit to all objects. 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, supremum and infimum exist.

In our case, concepts correspond to (partial) configuration threads. In particular,

- for each configuration, its *extent* (the code pieces which make up the configuration thread) and *intent* (the attributes which govern the configuration thread) are computed;
- all *dependencies* between configurations are computed, where a dependency is of the form "Any code piece valid for the sun configuration is valid for the ultrix and sony configuration as well";
- by computing a lattice of configuration concepts, a *taxonomy* of configurations is determined;
- *interferences* between configurations are displayed, where an interference between configuration threads means that they have common code where they should not;
- the *overall quality* of the configuration structure can be judged according to software engineering principles.

The concept lattice for the code in figure 2 is presented in figure 4. A concept lattice which arises from a more complex configuration table is presented in figure 3.[2] It reveals simple facts e.g. that source lines 21–28, 29–40, 201–207, and 11–20 depend on CRAY, apollo, macII, and SYSV being defined. But it also displays less obvious information, e.g. that lines 11–20 depend on all CPP symbols except sequent, alliant, i386, and SYSV386; that apollo and ultrix configurations have lines 126–200, 201–207, 11–20 in common; and that source lines depending on sony or ultrix depend on sun as well. Such knowledge is not easily extracted by hand from a source file like "xload.c"!

The concept lattice also allows to judge the overall configuration structure. According to software engineering principles, configuration-specific code should be collected in one module and hidden from the rest of the program. For example, code dealing with operating system dependencies should be collected in one module, and code dealing with window system details should be collected in another module. If several operating systems and window systems have to be supported, this can be done by maintaining several variants of the corresponding modules.

[2] Figure 3 and figure 6 are isomorphic copies of an instructive example presented in [Wi90]

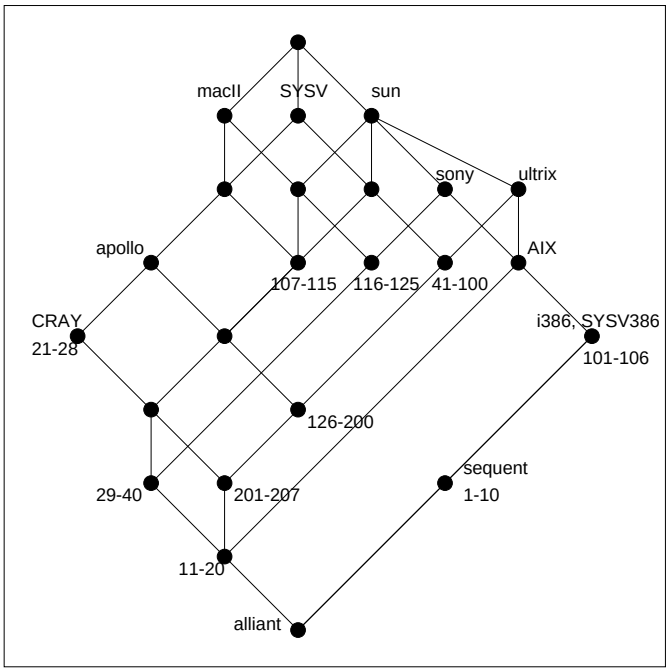


Figure 3: A concept lattice

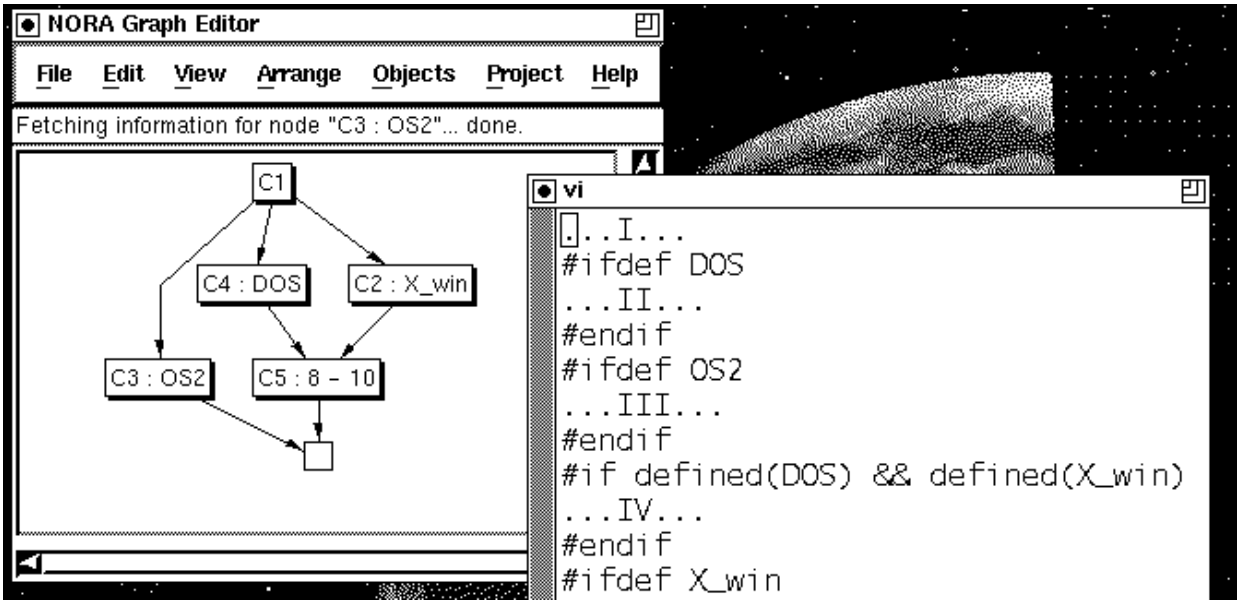


Figure 4: An elementary interference displayed by NORA/RECS

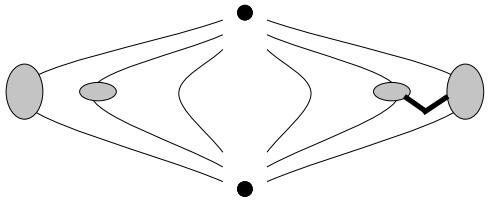


Figure 5: Horizontal lattice decomposition and an interference

Unfortunately, existing software does not stick to this principle. We will see later that “xload.c” (figure 1) is an extreme example of configuration hacking; in this introduction, we consider a very simple example, namely the code fragments in figure 2. Here, code piece IV can be part of the DOS as well as X-window configuration, preventing the code from modularization. And indeed, NORA/RECS displays this interference: lines 8–10 are governed by both DOS and X_win (figure 4).

Ideally, the lattice is composed of disjoint sublattices (figure 5), where the CPP symbols in each sublattice deal with the same topic, e.g. operating system variants. This guarantees high cohesion and low coupling between orthogonal configuration concepts.

1.3 Decomposing source files

For very chaotic configuration spaces, restructuring is appropriate. The first step is perhaps to reduce the size of the program family (called “amputation” in [Pa94]); in a second step a restructuring of the source code in order to obtain more cohesive modules is appropriate.

The concept lattice provides very good insight into the possibility and effect of an amputation. Furthermore, it allows for a restructuring of source code which is based on lattice decomposition. Our restructuring method – which should be considered a first step in configuration restructuring – does not produce modules in the traditional sense of the word; it merely offers a clever way to cut up a source file such that high cohesion and low coupling between configuration subspaces are achieved.

If the concept lattice can be decomposed horizontally, the governing preprocessor symbols can be divided into, say, n independent classes. If we know that preprocessor symbols in two different classes can never be defined at the same time (as is the case for e.g. DOS/OS2 and X_win), the code can be decomposed into n "modules". Each module covers a specific configuration aspect, for example operating system variants or user interface variants. It is the restructurer's task to guarantee that symbols in disjoint classes cannot be defined simultaneously, otherwise some possible configurations are lost. This requires knowledge about the semantics of the CPP symbols; usually, the symbols' names indicate their meaning.

The source code is then decomposed in accordance with a horizontal lattice decomposition. In case there is an interference (indicated in the right part of figure 5), an additional module containing the "problematic" configuration can be created. For the example in figure 2, restructuring produces three modules: an operating system dependent module, a window system dependent module, and the problematic module; the latter corresponds to a specific combination of operating system and window system variants.

```
/* OS variants */   /* WS variants */  /* problematic variant */
 ...I...              ...I...            #if defined(DOS) && defined(X_win)
 #ifdef DOS           #ifdef X_win       ...I...
 ...II...             ...V...            ...II...
 #endif               #endif             ...IV...
 #ifdef OS2           ...VI...           ...V...
 ...III...                               ...VI...
 #endif                                  #endif
 ...VI...
```

This example shows that a price must be paid for better cohesion, namely that redundant copies of code pieces must be introduced. Code pieces I and VI occur in all modules, in order to guarantee identical configuration threads after restructuring. A similar phenomenon occurs in the restructuring of source code: due to the Boehm-Jacopini theorem, any code can be made GOTO-free, but at the price that some code pieces must be duplicated. Future research must show whether a better modularization can be achieved.

2 Basic notions of concept analysis

2.1 The concept lattice

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. In our case, the objects are source code pieces, the attributes are governing preprocessor symbols, and the relation is called a *configuration table*. Figure 6 presents a (fictious) example of a configuration table, where source code pieces are given in form of line number intervalls. A cross in the table for code piece o and preprocessor symbol a means that o will only be included in a configuration thread if a is defined; we say "o is governed by a" or "o depends on a". Hence configuration threads are characterized by their governing preprocessor symbols.

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 $X, X_1, X_2 \subseteq O,\ Y, Y_1, Y_2 \subseteq A$

$$X_1 \subseteq X_2 \Longrightarrow \sigma(X_2) \subseteq \sigma(X_1) \qquad \text{and} \qquad Y_1 \subseteq Y_2 \Longrightarrow \tau(Y_2) \subseteq \tau(Y_1)$$

that is, both mappings are *antimonotone*;

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

as well as

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

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 $X_i \subseteq O, Y_i \subseteq A$

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

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 6, 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.

Basic Theorem for Concept Lattices [Wi82]: 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

	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 6: A configuration table

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 *labeling* of the concepts: a concept is labeled with an object, if it is the smallest concept in the lattice subsuming that object; a concept is labeled with an attribute, if it is the largest concept subsuming that attribute. The concept labeled with object o resp. the concept labeled with attribute a is

$$\gamma(o) = \big(\tau(\sigma(\{o\})), \sigma(\{o\})\big) \qquad \text{and} \qquad \mu(a) = \big(\tau(\{a\}), \sigma(\tau(\{a\}))\big)$$

The attribute labels of a concept c are written $\alpha(c)$, and the object labels of c are written $\omega(c)$. Utilizing this labeling, 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:

$$ext(c) = \bigcup_{c' \leq c} \omega(c') \quad \text{and} \quad int(c) = \bigcup_{c' \geq c} \alpha(c')$$

For any 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". If A and B are intents of concepts $C = \big(\tau(A), A\big)$ and $D = \big(\tau(B), B\big)$, and $C \leq D$, then $A \Rightarrow B$ obviously holds. For the set of all implications, a minimal and complete basis can be constructed, which means that any implication can be deduced from the basis, but that this property is lost if any basis implication is removed [Du87].

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 [Wi82]. Thus concepts do not "breed" new concepts; there is no proliferation of virtual information.

There is much more to say about the theory of concept lattices, but for the purposes of this paper, the basic theorem suffices. The interested reader should consult [DP90], which contains a chapter on concept analysis.

2.2 Interpretation of concept lattices

Figure 3 shows the concept lattice which has been derived from the table in figure 6. In the lattice, a configuration concept is a subconcept of another concept, if it has a smaller extent (i.e. the configuration thread has less code pieces), or equivalently, a larger intent (i.e. more governing symbols). Hence, going down in the lattice, we obtain more precise information about smaller object sets.

As an example, consider the concept labeled CRAY, which is in fact the concept μ(CRAY) = ({11-20, 21-28, 29-40, 201-207}, {CRAY, apollo, macII, SYSV}). And indeed, figure 6 reveals that this concept is a rectangle in the configuration table. It reveals a simple fact about the configuration space, namely that lines 11-20, 21-28, 29-40, 201-207 are exactly those which are governed by CRAY, apollo, macII, SYSV – and vice versa. The concept labeled apollo stands for μ(apollo) = ({11-20, 21-28, 29-40, 126-200, 201-207}, {apollo, macII, SYSV}), which again is a rectangle in the configuration table, higher but leaner than the first one: μ(CRAY)$\leq$ μ(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. The original table can always be reconstructed from the lattice, e.g. the column for i386 has entries for all objects below concept μ(i386), namely 1–10,101–106 whereas the row labeled 41–100 has entries for all attributes above, namely sun, SYSV, and ultrix. Hence, a context table (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.

The infimum of two concepts (C, V) and (D, W) says which preprocessor symbols govern the intersection of the extents: $(C, V) \wedge (D, W) = (C \cap D, \sigma(\tau(V \cup W)))$. Since $X \subseteq \sigma(\tau(X))$, this symbol set can be larger than just the "intuitive" $V \cup W$. In figure 3, μ(sony)$\wedge\mu$(ultrix) = ({1–10, 11–20, 29–40, 116–125, 101–106}, {sun, sony}) $\wedge$ ({1–10, 11–20, 41–100, 101–106, 126–200, 201–207}, {sun, ultrix}) = ({1–10, 11–20, 101–106}, {sun, sony, ultrix, AIX}) = μ(AIX).

The supremum says which code pieces are governed by the intersection of the intents: $(C, V) \vee (D, W) = (\tau(\sigma(C \cup D)), V \cap W)$; this can be more code than just $C \cup D$.

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 labeled 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 preprocessor symbols 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 "xload.c"! This example demonstrates the second main possibility to interpret a concept lattice: it represents all implications (that is, *dependencies*) between sets of attributes.

How can we use the lattice to determine which code pieces will actually be included in a configuration, if a certain set S of preprocessor symbols is defined? A code piece o is included if all governing symbols are defined. The governing symbols of o are $\sigma(\{o\}) = int(\gamma(o)) = \bigcup_{c \geq \gamma(o)} \alpha(c)$; these are just all attribute labels above $\gamma(o)$. Hence the code pieces included are given by the *configuration function* $K : \ 2^A \to 2^O$, where $K(S) = \{o \in O \mid \sigma(\{o\}) \subseteq S\}$. For an arbitrary set S, a direct geometric interpretation of this formula is difficult, which reflects the fact that a random choice of defined CPP symbols leads to strange configurations. At least all code pieces in $K(S)$ must be above $\bigwedge_{s \in S} \mu(s)$, as any code piece further down in the lattice must depend on preprocessor symbols not in S. For a singleton $S = \{a\}$, this means that $K(S) = \omega(\mu(a))$ or $K(S) = \emptyset$: the code pieces selected if just a is defined are either just the code piece labels of the concept labeled a, if there are no attribute labels above a; otherwise, defining a alone selects nothing. In general, "reasonable" choices of S select sublattices in the concept lattice.

2.3 Construction of the concept lattice

In order to give the reader an idea how a concept lattice is constructed from a formal context, we describe a simple construction algorithm. The concept lattice can be constructed either top-down or bottom-up; we describe the bottom-up version. The algorithm utilizes the fact that for a concept (X, Y),

$$Y = \sigma(X) = \sigma\Big(\bigcup_{o \in X} \{o\}\Big) = \bigcap_{o \in X} \sigma(\{o\})$$

The smallest element is $(\tau(\sigma(\emptyset)), \sigma(\emptyset))$. Hence one can start by first computing all the $\sigma(\{o\})$, which constitute the atoms of the lattice. For any given $o \in O$, this can be done by a simple loop over A with time complexity $\mathbf{O}(|A|)$. The other elements are then obtained as suprema of already computed ones. Due to the basic theorem this can be done by intersecting the attribute sets of any two elements already constructed, which can again be done in time $\mathbf{O}(|A|)$. The extent of a lattice element is obtained by applying τ, which needs two nested loops and has time complexity $\mathbf{O}(|O| \cdot |A|)$.

A hash table is used to store the lattice elements and check whether a newly constructed element is already in the lattice. Furthermore, one has to keep track of all pairs of elements to be considered for supremum computation. This is done with a FIFO-queue: initially, the queue contains all pairs of atomic elements; the

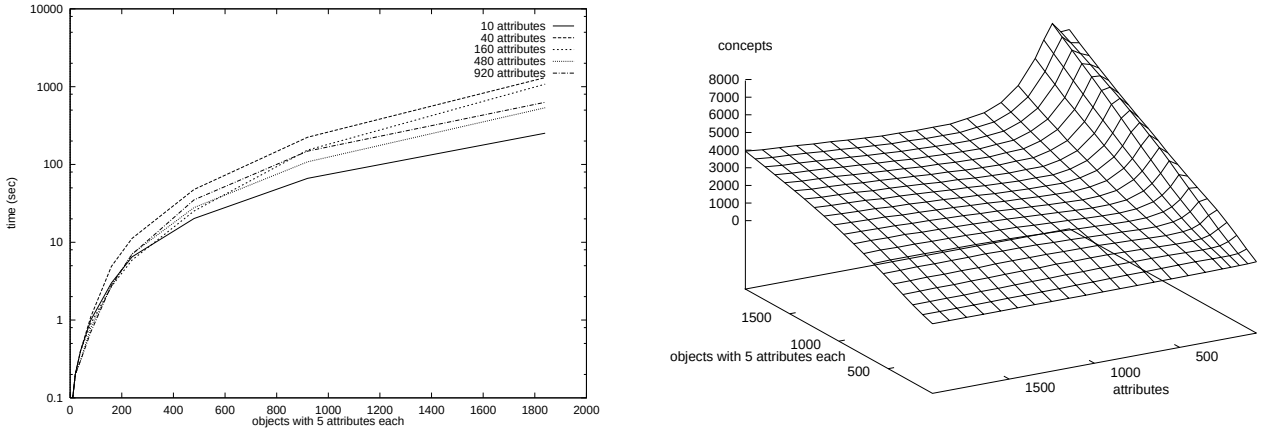


Figure 7: Time complexity and lattice size for random contexts

next supremum to be determined is given by the first element of the queue, and pairs of concepts $[c_1, c_2]$, where at least one is newly generated and not $c_1 \leq c_2$ or $c_2 \leq c_1$, are appended to the end of the queue.

The overall complexity depends on the number of lattice elements. The largest lattices for contexts of size $n \times n$ have 2^n elements; these lattices are isomorphic to finite boolean algebras freely generated by n elements. Thus the worst case running time of the construction algorithm is exponential in n. In practice, however, the concept lattice has typically $\mathbf{O}(n^2)$ or even $\mathbf{O}(n)$ elements rather than $\mathbf{O}(2^n)$, resulting in a typical running time of $\mathbf{O}(n^3)$; this makes the method feasible for reasonable large contexts.

Ganter has found a more efficient algorithm which avoids tracking the elements and suprema, but is more difficult to understand [Ga87]. This algorithm is used in the Darmstadt implementation of concept analysis. It has recently been reimplemented for NORA. Lindig [Li94] presents some empirical data on this implementation (figure 7), which have been obtained from random contexts. The first picture shows the time for lattice construction as a function of the number of objects; for 1000 objects (code pieces) the analysis needs 100 sec on a SUN ELC. The second picture shows the number of concepts as a function of object and attribute cardinality; it shows that this number can indeed grow exponentially. Fortunately, for UNIX source files, we found the number of configuration concepts to be much lower.

3 Inference of Configuration Structures from Source Code

The tool NORA/RECS for restructuring of configurations accepts source code as input and produces a graphical display of the concept lattice as intermediate representation. Reengineering is then done by analysing interferences and sublattices. Restructuring corresponds to a specific lattice decomposition; upon the restructurer's request, the source file is transformed accordingly. The source language is arbitrary, but the input file must stick to the conventions of the C preprocessor. NORA/RECS consists of the following phases:

1. front end: the front end separates code pieces and preprocessor statements, syntactically analyses the latter, and constructs a configuration 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 configuration table and computes the corresponding concept lattice.
3. visualization: this accepts a description of the concept lattice and produces a graphical display.
4. interaction: the lattice can be inspected, interferences can be selected, and modularizations can be triggered.
5. back end: by partial evaluation of preprocessor files, the source code is decomposed.

As usual, NORA/RECS is invoked as a UNIX command with the source file name as a parameter; additional options which control some display parameters may be added. This chapter describes the first three phases.

3.1 Construction of the configuration table

A configuration table describes how code pieces depend on preprocessor symbols. Configuration tables are used as input to formal concept analysis. We will now describe how to construct configuration tables from source files like "xload.c"; as there may be complex preprocessor expressions and nested "#ifdef"s, this process is not trivial. In the following semiformal construction rules, A, B, C denote preprocessor symbols, and p-p, n-n, q-q denote code pieces.

Basic rule

As already mentioned, an entry in the configuration table for a code piece o and a preprocessor symbol a means that o depends on (or is governed by) a; this means that o will only be included in a configuration thread if a is defined. Hence the basic rule for code pieces governed by single preprocessor symbols is:

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

$\Longrightarrow$

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

Conjunctions of preprocessor symbols

If a code piece is governed by a conjunction of preprocessor symbols, it depends on all of the symbols:

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

$\Longrightarrow$

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

Negated preprocessor symbols

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
```

$\Longrightarrow$

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

A similar rule applies to

```
#ifdef ... #elif ... #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 preprocessor symbols

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	...

In order to see that this rule is correct, imagine we introduce a new CPP symbol `AorB` which is always defined whenever `A` or `B` is defined. `A || B` is replaced by `AorB`, and any code piece dependent on `A` or `B` is in addition made dependent on `AorB`. This transformation of the source file keeps all configuration threads intact. The transformed source code would – according to the conjunction rule – produce exactly the configuration table which is given in the disjunction rule.

Complex governing expressions

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 $(A_1 \vee A_2 \vee \cdot \vee A_i) \wedge (B_1 \vee B_2 \vee \cdot \vee B_j) \wedge ...(C_1 \vee C_2 \vee \cdot \vee C_k)$, 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: for each negated symbol, as well as for each simple disjunction of the form $A_1 \vee A_2 \vee ...A_i$, an additional column is introduced. Additional crosses are then placed according to the disjunction rule (whenever a row contains an entry for A_ν, it must contain an entry for $A_1 \vee A_2 \vee ...A_i$).

Arithmetic expressions

In rare cases, one can find CPP expressions like "`#if version>50`" – that is, arithmetic CPP expressions which are used for configuration management. Our approach however assumes a binary "defined / undefined" semantics for CPP expressions. Therefore, arithmetic and relational expressions are treated as follows: for every arithmetic or relational expression, a new column in the configuration table is introduced. This column is labeled with the complete arithmetic expression, and an entry in the configuration table is made. Thus arithmetic expresions will show up as concept labels. NORA/RECS does not provide a fine-grained analysis of arithmetic and relational expressions, and therefore does not add dependencies as is done in the disjunction rule.

Nested "#ifdef"s, "#define"s and "#undefine"s

The treatment of nested "#ifdef"s is obvious: for any line preceding an "#ifdef", the governing symbols have already been determined. These are extended by an entry for the symbol(s) in the new "#ifdef". Example:

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

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

Nested #defines and #undefines can also be treated (for details, see [Kr93]). However experience has taught us that a “#define” is usually not used for configuration management, but for definition of constants or inline functions. Hence the current implementation of NORA/RECS ignores “#define”s and “#undefine”s.

3.2 Elementary patterns in concept lattices

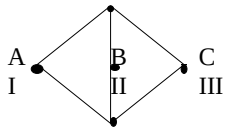
We will now explain some characteristic patterns in concept lattices, and provide some basic examples.

Chains and antichains

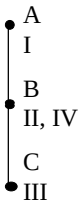
An antichain in a lattice is a set of uncomparable elements. In our case, an antichain in the concept lattice comes from code pieces which are governed by different, independent preprocessor symbols. A lattice which consists of only one antichain plus top and bottom element is called flat. Flat lattices are the optimal structures for configuration management, as there is no dependence whatsoever between different CPP symbols (left hand side of picture).

A chain in a lattice is a set of elements $c_1 < c_2 < c_3 <$ ... which are mutually comparable. In our case, chains result from nested “#ifdef”s. Note that in concept lattices, a chain can be interpreted as a sequence of implications: If code piece o depends on symbol a, it also depends on b; if o depends on b, it also depends on c etc. (right hand side of picture).

```
#ifdef A
...I...
#endif
#ifdef B
...II...
#endif
#ifdef C
...III...
#endif
```



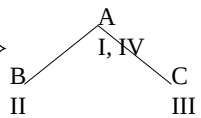

```
#ifdef A
...I...
#ifdef B
...II...
#ifdef C
...III...
#endif
...IV...
#endif
#endif
```



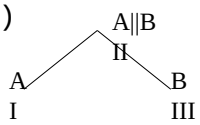
Suprema

A supremum which is not the top element indicates that two code pieces are governed by the same ”superordinate” CPP symbol. Simple disjunctions also show up as suprema in the concept lattice. Note that any supremum consists of two chains and an antichain, and the above explanations for chains and antichains still hold.

```
#ifdef A
...I...
#ifdef B
...II...
#endif
#ifdef C
...III...
#endif
...IV...
#endif
```




```
#if defined(A)
...I...
#endif
#if defined(A) || defined(B)
...II...
#endif
#if defined(B)
...III...
#endif
```

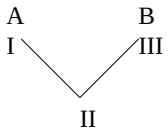


Infima

An infimum which is not the bottom element indicates that a code piece is governed by two different CPP symbols. Infima may indicate interference (see below). An infimum may also result from disjunctions, as in the right part of the picture.

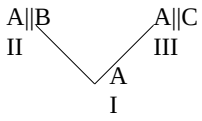
```
#ifdef A
...I...
#endif
#if defined(A) && defined(B)
...II...
#endif
#ifdef B
...III...
#endif
```

⇒




```
#ifdef A
...I...
#endif
#if defined(A) || defined(B)
...II...
#endif
#if defined(A) || defined(C)
...III...
#endif
```

⇒



Cascades

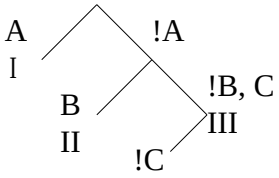
Nested "#ifdef# ... #elif" structures produce so-called cascades in the concept lattice. A cascade resembles the flow diagram of a nested if-then-else statement.

```
#ifdef A
...I...
#elif defined(B)
...II...
#elif defined(C)
...III...
#endif
```

⇒

	A	B	C	!A	!B	!C
I	X					
II		X		X		
III			X	X	X	

⇒



3.3 Interferences

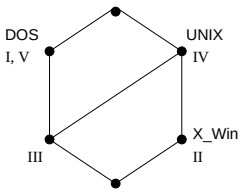
For our purpose, the most important property of a concept lattice is its ability to determine the code which depends on two or more preprocessor symbols; as already explained, such code is determined by the infimum operation.

Definition. Let a and b be two preprocessor symbols. We say a and b *interfere*, if $ext(\mu(a) \wedge \mu(b)) \neq \emptyset$. Otherwise, a and b are called *mutually exclusive*.

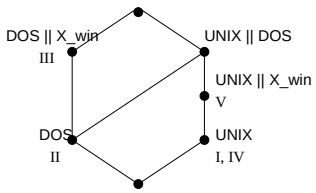
This definition means that the two CPP symbols have an infimum which is not the bottom element (only the bottom element can have empty extent, as two concepts with equal extent and different intent are not possible; in rare cases, the bottom element may have a non-empty extent and hence induce an interference).

Let us consider two examples, where the second example is slightly more complicated:

```
#ifdef DOS
...I...
#endif
#ifdef UNIX
#ifdef X_win
...II...
#endif
```




```
#ifdef UNIX
 ...I...
 #endif
 #ifdef DOS
 ...II...
 #endif
 #if defined(DOS)||
```



```
#ifdef DOS
...III...
#endif
...IV...
#endif
#ifdef DOS
...V...
#endif
```

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

Both lattices contain an interference. In the first example, code piece III is governed by both DOS and UNIX. In the second example, 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. This example uses concept labels which have been introduced by normalizing disjunctions, hence the interference does not really state that code piece II depends on DOS || X_win as well as UNIX || DOS; it merely states that operating system as well as user interface issues show up in both main configuration threads, and that —worse— there is a cross dependency between them.

Definition. Two CPP symbols a and b are called *disjoint*, if they cannot be defined at the same time. They are called *orthogonal*, if they deal with different and independent aspects of the configuration space, e.g. user interface variants vs. operating system variants.

These definitions are not completely formal. For two preprocessor symbols, only a human can decide whether they are disjoint or orthogonal, as this depends on their semantics. Usually, the symbols' names indicate their meaning, but nevertheless the restructurer must contribute some knowledge.

Interfering disjoint CPP symbols indicate dead code, which can be eliminated. This phenomenon can be found in the first example: there is code which depends on both DOS and UNIX, but DOS and UNIX are to the restructurer's best knowledge incompatible and hence disjoint.

Interfering orthogonal CPP symbols are also very suspicious: from a software engineering viewpoint, an interference indicates coupling between independent configuration aspects. 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. This principle is violated in the second example, as there is a cross dependency between operating system and window system. Cross dependencies prevent the lattice from being decomposed into independent sublattices, and this shows there is low cohesion and strong coupling between configuration threads.

In general, *low coupling* of configuration threads is achieved when orthogonal or disjoint preprocessor symbols appear in disjoint sublattices, as indicated in figure 5. 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 neither orthogonal nor disjoint, the corresponding sublattice is a grid:  . Missing arcs indicate that certain combinations of CPP symbols have not been taken into consideration, which is at least suspicious. 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. Indeed, we will see later that analysis of interferences in the RCS system reveals a subtle UNIX bug.

3.4 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 data reduction techniques have been implemented.

Sublattices: ignoring deeply nested "#ifdef"s

First, the restructurer may specify a maximal nesting depth for nested "ifdef"s. All "#ifdef"s and "#define"s which are more deeply nested are ignored. This results in a reduced concept lattice which displays only the overall structure of possible configurations, ignoring fine-grained details. The reduced lattice is a *sublattice* of the full concept lattice: ignoring deeply nested "#ifdef"s means that some concepts, dependencies and interferences will disappear; on the top level however, the dependencies are the same as in the full lattice.

In fact, ignoring deeply nested "#ifdef"s just means that some rows in the configuration table disappear, whereas others obtain new object labels (as code pieces merge and grow larger). It is a well-known theorem of concept analysis that deleting some rows or columns in a context table will result in a sublattice of the original lattice [WG93]. Therefore, an embedding of the reduced lattice into the full lattice can always be found.

Lattice congruences: merging similar rows

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 configuration table can be merged into one row if they "do not differ too much". The restructurer 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. In fact, if the rows for code pieces $\{o_1, o_2, ..., o_k\}$ are merged, all concepts c, where $\bigwedge_{1 \leq i \leq k} \gamma(o_i) \leq c \leq \bigvee_{1 \leq i \leq k} \gamma(o_i)$ are merged (in particular all the $\gamma(o_i)$ are merged).

3.5 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 [Wi89a, Wi89b, Sk92]. Some of the techniques used are to embed lattices into n–dimensional 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 implemented a simpler approach, based on the Sugiyama algorithm [STT81]. This well-known layout algorithm for arbitrary directed graphs uses the topological ordering of nodes in order to determine their vertical position. The nodes are grouped in layers, where each layer contains nodes with identical distance (i.e. number of edges) to the top element. The algorithm then tries to minimize crossings by choosing appropriate horizontal positions for the nodes.

If there is enough screen space, NORA/RECS will display the concept labels inside the concept's box; otherwise it can be looked up by clicking at the concept box. The concepts and their labels are also written to a file.

As the layout results are not always satisfactory, the user may finally change the graph layout manually (but the system will maintain integrity of the concept lattice). We plan to integrate the more sophisticated display algorithms in a future version.

4 Experimental results

We applied NORA/RECS to several UNIX programs. The lattice layouts in figures 8, 9, and 11 are manually improved.

4.1 Example 1: the TC shell

Our first example is a popular shell, the “tcsh” 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 for configuration management. In the concept lattice (figure 8)[3], singleton attribute or object labels are displayed in the diagram, the others can be looked up in a separate window by clicking on a concept (e.g. C19). It turns out that the configuration structure is perfect according to the criteria described above, as the lattice is almost flat. It seems that there is an interference between concepts C17, C20 – C21, and C18. But a look at the source code reveals that both VFORK and FASTHASH, as well as the attribute labels of C18, have to do with the hash function used, hence there are no dependencies between orthogonal configuration concepts.

4.2 Example 2: the RCS stream editor

Our second example is the stream editor from the RCS system “rcsedit.c”. This 1656–line program uses 21 preprocessor symbols for configuration management. The concept lattice has 33 concepts and is is shown in figure 5. In the left part of the lattice, there are a lot of simple variants, which correspond to specific features like “has_set_uid” or “has_readlink”. The concepts below C24/C10/C21 (concerning networking) form a grid-like cluster.

But there is an interference manifest in C27, which is the infimum of C3 and C26. C3 is labeled has_rename (as can be seen in the info box), C26 is labeled has_NFS, and C27 is labeled 1425 – 1427. Thus, lines 1425 – 1427 are governed by both has_NFS and has_rename. It seems that C13 is a similar interference, but as C12 is labeled ”bad_a_rename”, both C12 and C3 have to do with the file system.

Thus, 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

[3] The corresponding figure in [KS94] was produced with the old version of NORA/RECS which did *not* ignore “#define”s (and also had some bugs). As explained in section 3.1., the new version ignores “#define”s and “#undefine”s. Hence the old lattice in [KS94] had one more concept, namely BITS_PER_BYTE, which is in fact a constant and not a configuration attribute. The same remark applies to figure 9, which had some additional “virtual” concepts and interferences in [KS94]. The central interference is however still present, and the new lattice is a sublattice of the old one.

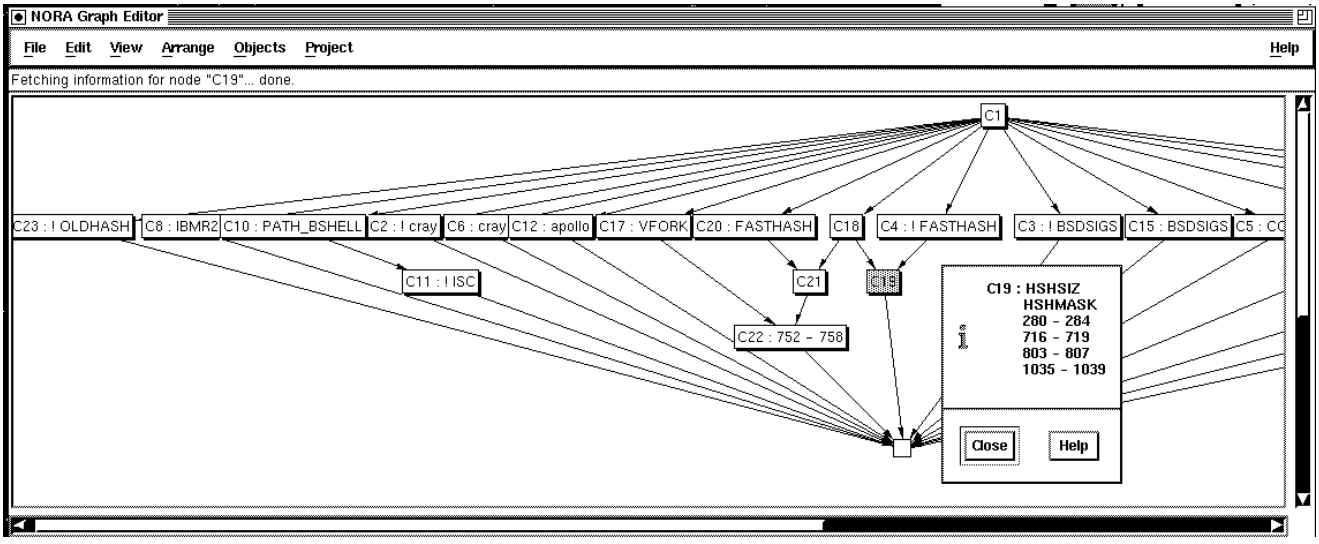


Figure 8: Configuration structure of tcshell module “sh.exec.c”

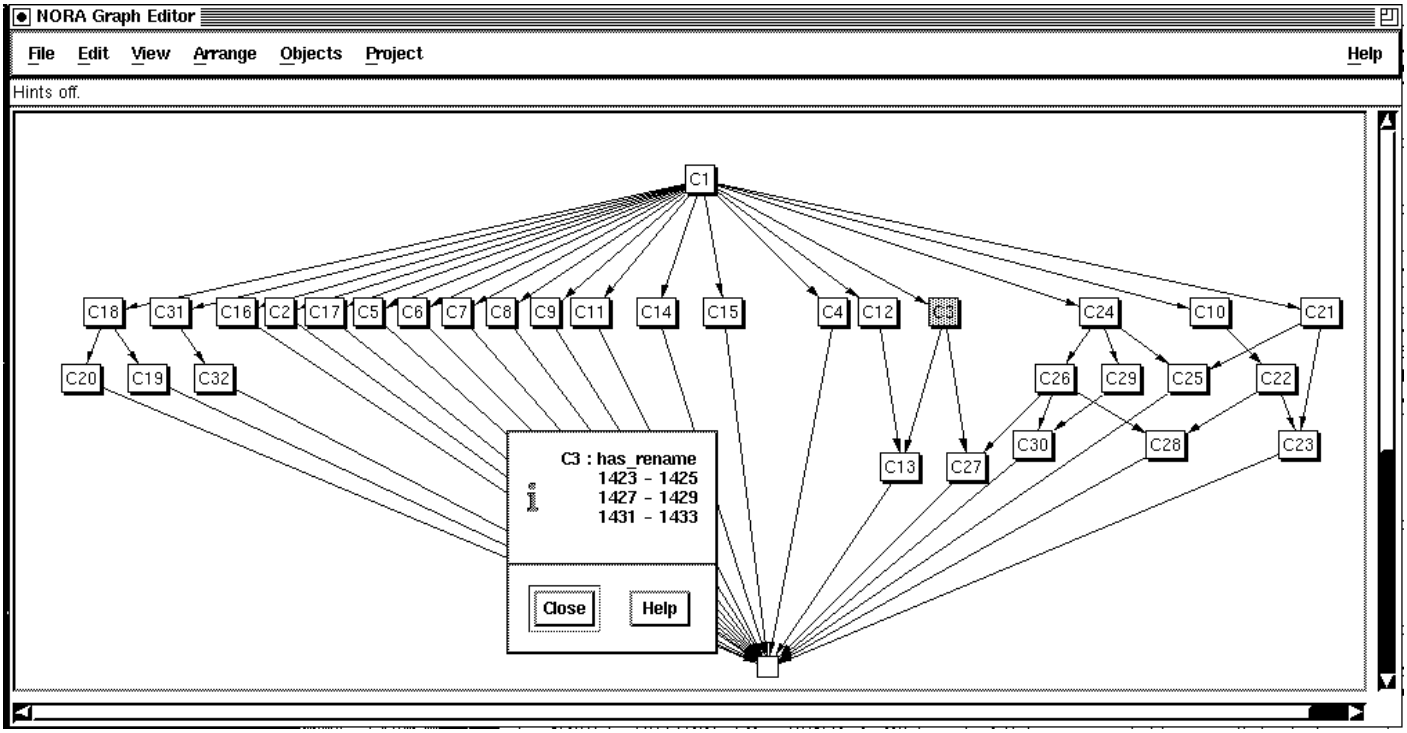


Figure 9: Configuration structure of the RCS stream editor

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 NORA/RECS can 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”[4].

4.3 Example 3: “xload.c”

Let us now come back to our introductory example, “xload.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 and is shown in figure 10. It looks pretty chaotic – the program obviously suffers from configuration hacking[5]. We therefore used data reduction to display only the top 4 “#ifdef” nesting levels (figure 11).

Even on the top level, there are interferences, namely C28 and C32, and the central role of C30 does not inspire confidence (C28 is the infimum of C22 and C24, whereas C32 is the infimum of C2 and C30. C22 is SYSV, and C24 is !apollo. C2 is SVR4 || UTEK || alliant || hex || sequent || sgi || sun. C30 is is a set of 9 code pieces governed by the sundries SYSV386, !LOADSTUB and !KVM_ROUTINES). Overall, the lattice consists of several “standalone” configurations (C3, C4, C42, C26, C5, C6, C17, C18, C19), a “SVR4 || UTEK || alliant || hex || sequent || sgi || sun“ sublattice (concepts $\leq$ C2), and a “not macII, not apollo” sublattice (concepts $\leq$ C24/C37). The top element C1 shows that several code pieces are configuration independent (not governed by any CPP symbol), while the bottom element C43 shows that several CPP symbols are defined, but not used in “#ifdef”s – namely those which are used for definition of constants or inline functions.

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

[5] if #defines irrelevant for configuration managment are ignored, the lattice shrinks to 103 concepts – but still looks chaotic

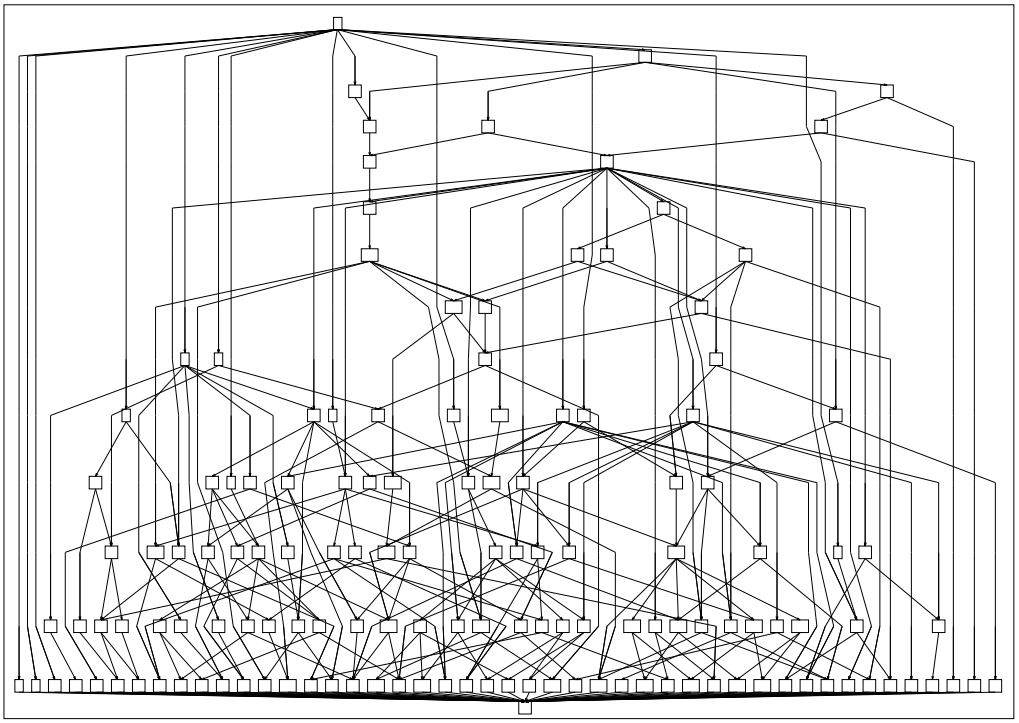


Figure 10: Configuration structure of “xload.c”

5 Lattice analysis and interference detection

Once the lattice has been constructed and layouted, the restructurer may inspect it in an interactive manner. NORA/RECS offers the following functions:

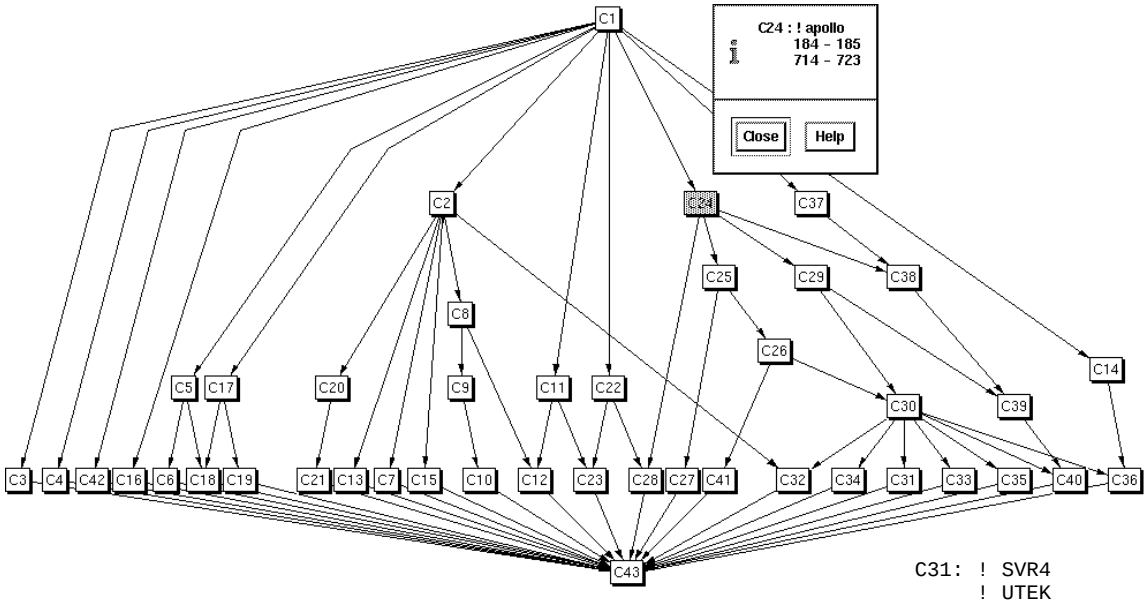
- for every concept c, its labels $\alpha(c)$ and $\omega(c)$ can be displayed by a simple mouse click;
- the source code pieces given in $\omega(c)$ can be displayed;
- lattices can be horizontally decomposed (if possible);
- interferences of minimal connectivity can be computed and displayed (see below);
- sublattices can be selected by clicking on their top or bottom elements;
- the intersection of sublattices can be displayed, which contains interferences (see below);
- lattice decompositions and corresponding source file restructuring can be triggered (see below);

After analysis, the restructurer can execute the restructuring algorithms described below. He may also manually restructure the source code, and repeat the analysis. Hence, NORA/RECS supports an interactive and incremental way of analysing configuration structures.

It is planned to include further analysis algorithms in a future version, in particular the minimal implication basis [Du87], subdirect decomposition [Wi83] and tensor decomposition [Wi85].

5.1 Automatic interference detection

As explained above, interferences indicate coupling between configuration threads. A restructuring of the configuration space should of course try to minimize coupling. Restructuring corresponds to decomposition of the lattice (see below). Therefore, coupling is minimal, if the lattice is decomposed in such a way that the number of connecting edges between sublattices is minimal. The number of edges between sublattices is a measure of the “badness” of the interference and is called the *connectivity* of the interference.




```
C1: 0 - 35
    37 - 39
    45 - 47
    49 - 51
    57 - 59
    61 - 63
    70 - 72
    74 - 76
    85 - 87
    89 - 91
    93 - 95
    97 - 99
    101 - 103
    109 - 111
    115 - 117
    119 - 125
    723 - 724
C2: SVR4 OR UTEK OR alliant
    OR hcx OR sequent OR sgi OR sun
C3: apollo
    125 - 184
C4: X_NOT_POSIX
     117 - 119
C5: (OSMAJORVERSION = 4)
    sony
    103 - 104
    108 - 109
C6: ! mips
    106 - 108
C7: sgi
    99 - 101
C8: SVR4 OR UTEK OR alliant
C9: SVR4
    111 - 112
    114 - 115
C10: ! FSCALE
     112 - 114
C11: MOTOROLA OR UTEK OR alliant
C12: 91 - 93
C13: hcx
     87 - 89
C14: macII
     76 - 85
C15: sequent
     72 - 74
     95 - 97
C16: AIXV3 OR CRAY
     63 - 70
C17: mips OR umips
C18: mips
     104 - 106
C19: ultrix OR umips
     59 - 61
C20: sun
     51 - 53
     56 - 57
C21: i386
     53 - 56
C22: SYSV
C23: MOTOROLA
     47 - 49
C24: ! apollo
     184 - 185
     714 - 723
C25: ! SYSV
     ! SYSV386
     271 - 272
     713 - 714
C26: ! KVM_ROUTINES
     316 - 317
     712 - 713
C27: KVM_ROUTINES
     272 - 316
C28: SYSV386
     185 - 271
C29: ! LOADSTUB
C30: 332 - 334
     336 - 338
     398 - 401
     449 - 455
     466 - 468
     483 - 490
     547 - 558
     562 - 564
     709 - 712
C31: ! SVR4
     ! UTEK
     ! alliant
     ! hcx
     ! sequent
     ! sgi
     ! sun
     570 - 709
C32: 564 - 570
C33: ! KERNEL_LOAD_VARIABLE
     401 - 449
     455 - 466
C34: ! KERNEL_FILE
     338 - 398
C35: ! KMEM_FILE
     334 - 336
C36: X_AVENRUN
     fxtod
     468 - 477
     490 - 511
     558 - 560
C37: ! macII
     39 - 40
     44 - 45
C38: 40 - 41
     43 - 44
C39: 41 - 43
C40: 477 - 483
     511 - 547
     560 - 562
C41: LOADSTUB
     317 - 332
C42: att
     35 - 37
C43: word
     ! word
     ! n_type
     n_type
     FSCALE
     KERNEL_FILE
     KMEM_FILE
     VAR_NAME
     PROC_NAME
     BUF_NAME
     DECAY
```

Figure 11: Top level configuration structure of “xload.c”

The interference analysis algorithm incorporated in NORA/RECS tries to horizontally decompose the lattice in such a way that connectivity is minimal. This will guarantee that the restructured source file has highest cohesion and minimal coupling. Lattice decomposition is done in a top-down fashion: top–level interferences between big sublattices are detected first, whereas minor interferences will be detected very late. This helps for restructuring, as interferences between big sublattices are more likely to indicate errors or bad configuration structure. The sublattices can later be analysed recursively.

The algorithm for detecting interferences of minimal connectivity works as follows.

1. Try a horizontal decomposition of the lattice. This is done by removing the top and bottom elements and their outgoing edges, and then determine the connected components of the (undirected) concept graph by a standard algorithm. If successful, there are no top level interferences (connectivity $k = 0$). Reattach top and bottom element to each sublattice, and apply the remaining steps recursively to the sublattices.
2. If a sublattice cannot be decomposed horizontally, it may contain interferences. First, simple interferences of connectivity $k = 1$ are investigated. These are detected by removing the top and bottom elements, and then computing the *biconnected components* of the remaining graph. A bridge between two biconnected components which leads to an $\wedge$–reducible concept node (that is, of the form $c = a \wedge b$) points to an interference. The node is highlighted.
3. Often, there is more than one interference between sublattices. For example, in figure 9, the two edges C6/C13 and C15/C27 connect two sublattices. Thus we compute the k–connected components of the lattice graph (without top and bottom), where k is minimal. A simple method to determine k–connected sublattices is to consider all sets of k $\wedge$–reducible concept nodes and test whether their removal will break the graph into unconnected subgraphs.

Only the restructurer can decide whether the interference must be resolved, or whether it should be ignored. This decision must be based on the semantics of the involved CPP symbols. If the semantics is not documented, the lattice structure provides insight into the meaning of a certain preprocessor symbol.

5.2 Determining sublattice intersections

The above algorithm proposes a lattice decomposition (and hence a code restructuring, see below) which has minimal coupling. This sounds reasonable from a software engineering viewpoint, but as the analysis does not take the semantics of CPP symbols into account, the proposed decomposition may be against the restructurer's intuition. NORA/RECS therefore offers a complementary approach, where sublattices are not determined automatically, but selected by the restructurer.

The restructurer may click at a number of concept nodes $\{c_1, ..c_n\}$, and thereby select the downward suborder $\downarrow \{c_1, ..., c_n\} = \{x \mid \exists i : x \leq c_i\}$ (the dual operation of selecting upward suborders is also supported)[6]. The downward suborder is then highlighted (or coloured) on the screen. It provides interesting information, as the code piece labels in $\downarrow \{c_1, ..., c_n\}$ are those which depend on the intent of one of the c_i. For example, selecting the downward suborder $\downarrow \{\mathsf{SYSV}, \mathsf{SVR4}\}$ displays all code pieces in the "System V" or "System V Release 4" versions.

Several downward sublattices may be selected this way, and each highlighted in a different colour[7]. The intersection of such sublattices also contains interferences; in fact, the maximal elements in the intersection of two sublattices are interferences. These are, however, not necessarily of minimal connectivity. But they are choosen by the restructurer, which may be more appropriate.

---

[6] if $n = 1$, $\downarrow \{c_1, ..., c_n\}$ is a sublattice, but otherwise not all subsets of elements have a supremum.

[7] unfortunately, the current NORA version only supports black and white

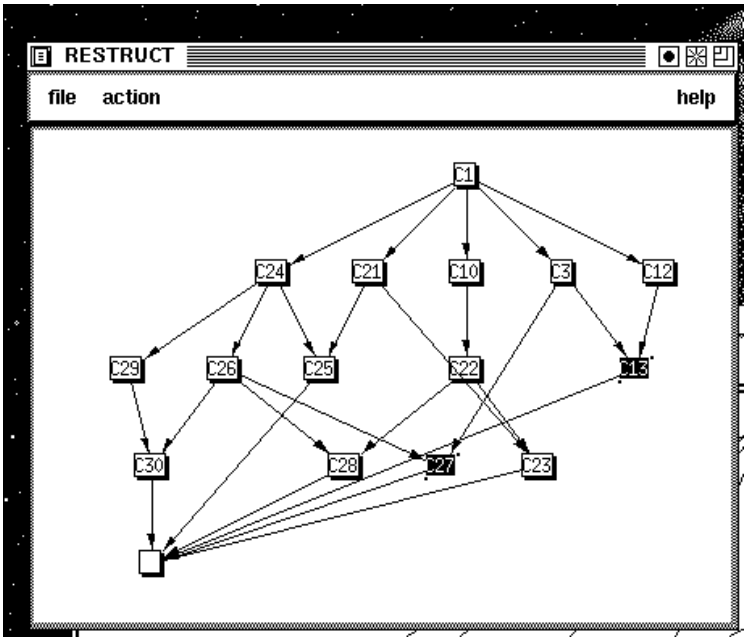


Figure 12: Automatic interference detection in a sublattice

5.3 Example: analysing “rcsedit.c”

The configuration lattice of the RCS stream editor was given in figure 9. After initial horizontal decomposition of the lattice, NORA/RECS immediately detects two interferences with connectivity 1, namely C27 and C13 (figure 12). We already argued that C13 is not an interference, so it seems obvious to decompose the source code “along the C27 interference”. But the restructurer is wise to take a look at the other concepts before, in order to achieve high cohesion. For example, C4 is labeled “!bad_a_rename” and hence belongs to the whole complex of networking. The other concepts in the left part of the lattice all deal with specific UNIX features; all concept labels are orthogonal (but not disjoint).

It seems that restructuring does not make much sense here, because the original program already had a reasonable configuration structure. As the NFS bug which leads to the interference obviously cannot be fixed, there is little hope that the interference can be removed.

6 Code decomposition

In this section, we describe how a source file can be decomposed into several source files or “modules”, where each module covers a cohesive subset of the configuration space. The decomposition process is driven by the structure of the concept lattice and implemented via partial evaluation of preprocessor files. Note that the resulting “modules” are not traditional modules in the software engineering sense. They are merely subsets of the original set of code pieces, where the lattice-based decomposition guarantees high cohesion and low coupling.

We assume that the code pieces itself need not be changed. We do not consider the case where code pieces itself are minced and remerged, as this involves program understanding and program transformation. If individual code pieces must be broken for restructuring, the method described in this paper cannot be used.

Modules (henceforth written without apostrophes) will in general contain redundant copies of source code pieces, and it is well known that code duplication is problematic from a maintenance viewpoint. Unfortunately, code duplication can in general only be avoided if all code pieces for each of the modules are textual neighbours in the original source file. Therefore, the technique described in this section should be considered as a first step

to configuration restructuring. Future research must show whether it is possible to automatically restructure configurations such that code duplication can be avoided and more "traditional" modules be created.

6.1 Partial evaluation of CPP files using NORA/ICE

Restructuring of configurations relies on partial evaluation of CPP files; a technique which will be sketched in this section. It is implemented in NORA/ICE, a tool for incremental configuration management based on feature logic [ZS94]. NORA/ICE – among other features – offers partial evaluation of preprocessor files. It allows to simplify preprocessor files with respect to the information that certain (combinations of) governing symbols will (or will not) be defined. NORA/ICE will simplify governing expressions and delete code pieces or preprocessor statements with respect to a given "context" expression, which is assumed to be true. The ordinary preprocessor behaviour is included as the "limit case", namely that *all* preprocessor symbols have a known value. NORA/ICE allows arbitrary complex context expressions, including those which introduce new symbols. Simplification is not just constant folding, but is based on *feature unification*.

Partial evaluation will considerably reduce the size of the source file, if several preprocessor symbols are known to be always defined or undefined. Figure 12 gives a simple example of partial evaluation, where preprocessor symbol A is assumed to be always undefined.

6.2 Preliminary simplification of the configuration space

According to Parnas, the first step in restructuring must be to reduce the size of the program family. If the restructurer has some knowledge about the old configuration structure and the meaning of the old preprocessor symbols, he might conclude that certain configurations (i.e. certain preprocessor settings) must no longer be supported. Hence the corresponding code is irrelevant and should be discarded (this is called "amputation" in [Pa94]). In particular, if certain preprocessor symbols are no longer needed, code depending on them will never be included in any restructured configuration and can be deleted. Such a simplification of the source code is appropriate before more complicated restructuring takes place.

6.3 Source code decomposition

Generation of modules from sublattices

Once a sublattice has been determined – either by horizontal decomposition or by explicit selection –, a module corresponding to this sublattice can be created. This works with every sublattice, but of course the restructurer

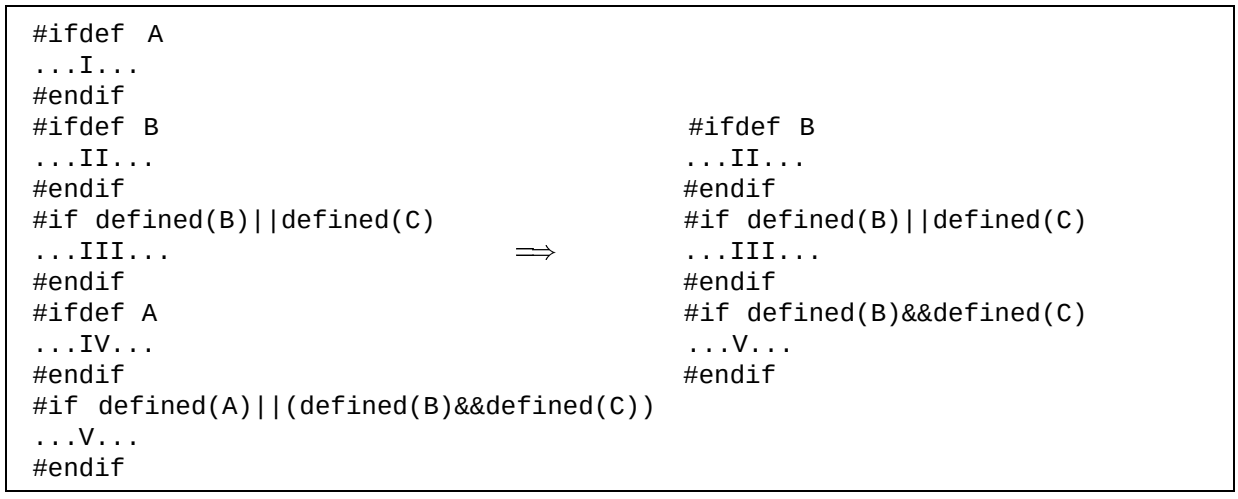


Figure 13: Partial evaluation of a preprocessor file under context expression $\neg defined(A)$

should try to achieve high cohesion and low coupling. Also, preprocessor symbols in the sublattice should be orthogonal or disjoint from the rest of the lattice.

Module generation is done by partial evaluation of CPP files. Let $C = \{c_1, c_2, ..., c_k\}$ be the concepts *not* in the sublattice and $X = \{x_1, ..., x_n\} = \bigcup_{i=1}^{k} \alpha(c_i)$ all their attribute labels. Then the source text is fed to NORA/ICE, together with the context expression $[\neg defined(x_1), ..., \neg defined(x_n)]$. This removes all code pieces not in the configuration subspace, and simplifies the governing expressions for the remaining code pieces.[8] The resulting source code contains only preprocessor symbols which appear in the sublattice.

Generation of problematic variants from interferences

Once an interference has been determined, a special "problematic" variant can be generated. Let $C = \{c_1, ..., c_k\}$ be the concept nodes which constitute an interference of connectivity k. C need not necessarily be an antichain; C can be a suborder or even a sublattice. Let $X = \bigcup_{i=1}^{k} \alpha(c^i)$ be all attribute symbols of C, and let W be the attribute symbols in $\uparrow C$. Then the source text is fed to NORA/ICE, together with the context expression $[\neg defined(y) \mid y \in A \setminus (W \cup X)]$. This creates a source text which subsumes exactly the configurations containing the problematic code pieces.

6.4 Example: decomposing "xload.c"

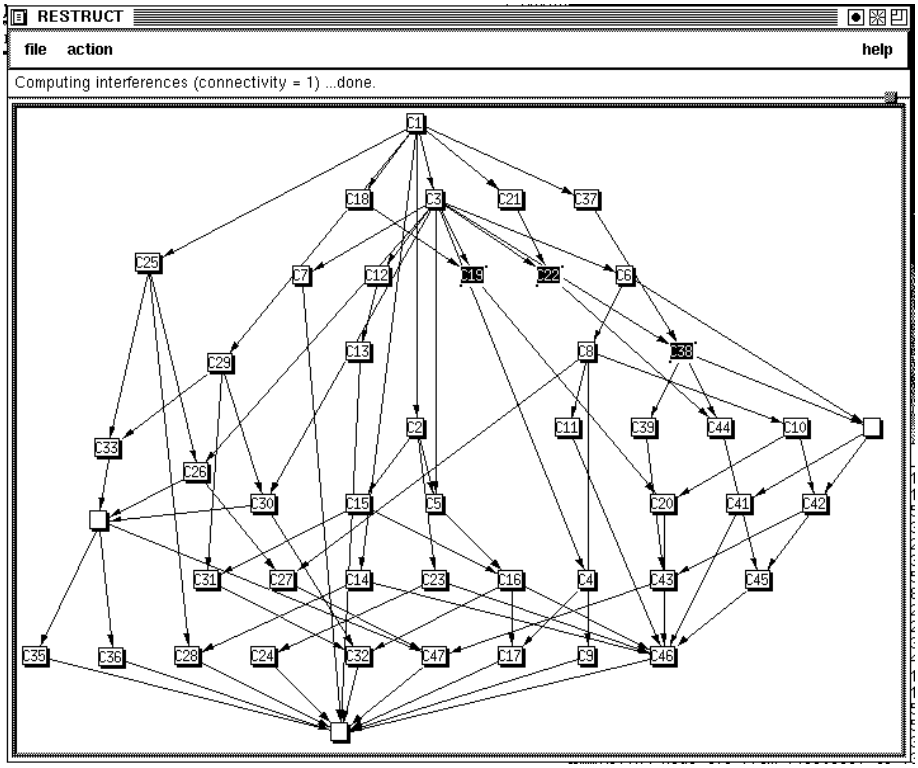
Let us now try to restructure "xload.c". We begin with the "amputation" step. Figure 11 shows that there are several "standalone" configurations for rather uncommon machines. We decide to discard the following configurations: apollo, X_NON_POSIX, sony, CRAY, mips, umips, att, LOADSTUB, alliant, sequent, UTEK, hcx, sgi.[9] Feeding the source code with context expression "[!defined(apollo) && !defined(X_NOT_POSIX) &&...&& !defined(sgi)]" to NORA/ICE, a reduced source file with 547 source lines results. The corresponding concept lattice has 68 concepts, which is still a lot. A look at the lattice reveals that there are a lot of interferences concerned with a CPP symbol KVM_ROUTINES. We therefore decide that we consider only configurations where these routines are available (whatever they may do). This results in a further reduction of the source file; it is now 501 lines long and the corresponding lattice has 48 concepts. The symmetric case that we assume "!defined(KVM_ROUTINES)" leads to a small source file with 14 configuration concepts, and is not further investigated.

The 48–concept lattice is then subject to interference analysis and code decomposition. There are three top-level interferences of connectivity $k = 1$, namely C19, C22 and C38 (figure 14). Removal of each of these would isolate C18, C21 resp. C37 (macII, AIXV3, ! macII) and is not further investigated. Furthermore, there is one interference of connectivity $k = 2$, namely C28/C46. Removal of these concept nodes would isolate C14, but leave the rest of the lattice unchanged and is therefore not further investigated. There are als two interferences of connectivity $k = 3$, namely C33/C26/C46 and C33/C26/C28, but these do not produce a "clean" lattice decomposition either.

We therefore decide to base modularization on sublattice selection. Obviously, most concepts are below C3, the "not System V" concept. We therefore produce a module which handles all configurations except "System V"; this is based on the sublattice $\downarrow \{C3\}$. According to section 6.3, NORA/RECS collects the concept labels of the maximal elements outside $\downarrow \{C3\}$; these are C18, C21, C37, C25, C29 with labels macII, AIXV3, !macII, SYSV, MOTOROLA. NORA/RECS then feeds the simplified source file to NORA/ICE, together with context expression $[\neg defined(macII), \neg defined(AIXV3), defined(macII), \neg defined(SYSV), \neg defined(MOTOROLA)]$.

[8] Actually, it would be enough to undefine only the maximal elements of C.

[9] This is **not** to be understood as a recommendation to the authors of "xload.c"!




```
C1:  0 - 36
     38 - 40
     42 - 44
     50 - 53
     60 - 63
     72 - 75
     77 - 82
     86 - 93
     180 - 181
     202 - 203
     205 - 206
     208 - 209
     213 - 214
     224 - 225
     228 - 229
     231 - 232
     252 - 253
     255 - 256
     258 - 259
     263 - 264
     418 - 419
     493 - 494
     497 - 506
C2:  SVR4 OR sun
C3:  ! SYSV OR ! SYSV386
     181 - 191
     193 - 195
     197 - 202
     214 - 220
     223 - 224
     232 - 240
     244 - 248
     250 - 252
     264 - 273
     282 - 285
     300 - 307
     353 - 364
     368 - 370
     417 - 418
     494 - 497
C4: ! USG
     276 - 277
     281 - 282
C5:  370 - 376
C6:  ! STDC__ OR ! SVR4
C7:  USG
     273 - 276
C8:  ! SVR4
     222 - 223
C9:  279 - 281
C10: ! sun
     376 - 377
     416 - 417
C11: ! SYSV
     229 - 231
C12: hpux
     191 - 193
     240 - 241
     243 - 244
C13: hp9000s800
     241 - 243
C14: SYSV386
C15: SVR4
     82 - 86
C16: 220 - 222
C17: 277 - 279
C18: macII
     63 - 72
C19: 195 - 197
     285 - 294
     307 - 328
     364 - 366
C20: 377 - 382
C21: AIXV3
     53 - 60
C22: 248 - 250
C23: sun
     44 - 46
     49 - 50
C24: i386
     46 - 49
C25: SYSV
C26: ! SYSV386
C27: 225 - 228
C28: 93 - 180
C29: MOTOROLA
C30: 209 - 210
     212 - 213
     259 - 260
     262 - 263
C31: 75 - 77
C32: 210 - 212
     260 - 262
C33: 40 - 42
C34:
C35: m88k
     206 - 208
     256 - 258
C36: m68k
     203 - 205
     253 - 255
C37: ! macII
     36 - 38
C38: 294 - 300
     328 - 329
     331 - 333
     337 - 353
     366 - 368
C39: ! AIXV3
     335 - 337
C40:
C41: ! MOTOROLA
     329 - 331
C42: 382 - 383
     415 - 416
C43: 414 - 415
C44: 333 - 335
C45: 383 - 414
C46: 490 - 493
C47: 419 - 490
C48:
```

Figure 14: Simplified xload.c with three interference of connectivity 1

The resulting source file consists of 197 lines and can be installed on all non-System V platforms (but not CRAY etc., as these configurations have been amputated before).

Figure 14 also shows that even in the simplified source file, lines 490–493 (extent of C46) occur in a lot of configurations. These lines are governed by the easy-to-understand expression

```
!(defined(SVR4) || defined(sun)) && !defined(macII) &&
!defined(AIXV3) && !(defined(SYSV) && defined(MOTOROLA)) &&
!(defined(SYSV) && defined(SYSV386)) && !(defined(SYSV) &&
!defined(macII) && !defined(AIXV3) && defined(MOTOROLA) &&
!(defined(SYSV) && defined(SYSV386)))
```

In fact, C46 is an interference of at least connectivity 7. We therefore decide to create a "problematic" variant, which subsumes all configurations including lines 490–493. According to section 6.3, this is done by collecting all attribute labels not in $\uparrow \{C46\}$ and feeding their "undefinedness" to NORA/ICE. This results in a source text of 289 lines; its concept lattice has still 19 concepts.

But the (human) restructurer can do more: from the lattice, he can conclude that lines 490–493 occur in all configurations except USG, !USG, hpux, SYSV, MOTOROLA (and their subconfigurations). Therefore, he can replace the governing expression by another one, which might be simpler. This manual operation produces a different (and simpler) lattice, which still covers the same configuration space. NORA/RECS cannot deduce lattice-specific equivalences between governing expressions, and hence cannot transform governing expressions automatically.

It should be noted that the above example has sort of an *ad libitum* character. A salesman would perhaps make different choices in the initial simplification step, and a Unix guru would perhaps find a more clever modularization. But the example clearly demonstrates the principle: source code is simplified and decomposed according to an analysis of dependencies between configurations. Furthermore, a human restructurer can obtain insight from the lattice which helps to manually restructure a source file.

7 Conclusion

NORA/RECS is a valuable tool for *analysis* of configuration spaces, but as a tool for *restructuring*, the approach is still in its infancy. Although source code decompositions which guarantee high cohesion and low coupling are nice, maybe it is not what restructurers really want. We therefore plan to use more of the theory of concept lattices, in order to obtain more powerful restructuring algorithms. In particular, we want to investigate the following topics:

- The minimal implication basis, mentioned in chapter 2, could serve as a help for restructuring, as it already generates the whole lattice.
- Concept lattices are isomorphic to the lattice generated by their join- and meet-irreducible elements. This can be used to simplify governing expressions according to lattice-specific equivalences.
- The congruences and subdirectly irreducible sublattices of a concept lattice might offer a better basis for restructuring than horizontal lattice decomposition.

For all of these options, efficient algorithms exist [WG93]. Work is already under way to utilize implications, irreducible elements and subdirect decomposition for restructuring.

In this paper, no correctness proofs have been given. But every restructuring algorithm must face some correctness criterion, in order to be sure that configuration threads are kept intact. We have developed a formal notion of correct restructuring of configurations and have proven our restructuring algorithm correct. For reasons of readability, the proofs are not included in the present article.

The methods presented in this paper can easily be adapted to more modern configuration management systems, although this would be a mere technical task. From a scientific viewpoint, it is more interesting to explore other applications of concept analysis in software engineering. Besides configuration restructuring, we consider the following applications of concept analysis:

- *Analysis of software architectures.* A software architecture is defined by relations between components, and hence can be subject to concept analysis. This might also help for automatic modularization of old code.
- *Software component retrieval.* Imagine a library where components are indexed by keywords. The relation between components and keywords can be subject to concept analysis. The resulting lattice allows for incremental narrowing of the set of still possible components, and gives users feedback about the still applicable keywords.

NORA/RECS is part of the inference-based software development environment NORA[10]. NORA aims at utilizing inference technology in software tools and – besides NORA/RECS – covers the following topics:

- NORA/ICE (incremental configuration engine) offers configuration management based on feature logic [ZS94];
- NORA/HAMMR (highly adaptive multi-method retrieval) offers software component retrieval based on deductive and lattice-theoretic techniques [FKS94, Li94];
- NORA/HOML (higher-order module language) is a calculus for designing reference architectures, which is based on $\lambda\nu$–calculus with dependent types [Gr94].

More general descriptions of NORA can be found in [SGS91, GS93, SFGKZ94].

NORA/RECS can be obtained via anonymous ftp: `ftp.ips.cs.tu-bs.de` (134.169.32.1).

Acknowledgements. Several persons contributed to NORA/RECS. Maren Krone detected how to treat disjunctions and implemented the frontend. Anke Lewien implemented interactive restructuring. Christian Lindig implemented the graph layouter and the concept lattice algorithm, and conducted several experiments. Martin Skorsky from the Darmstadt algebra group provided a lot of helpful comments. Andreas Zeller implemented the NORA graph editor and developed NORA/ICE. Franz-Josef Grosch provided valuable comments on a preliminary version of this article.

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

8 References

[DP90] Davey, B.A., Priestley, H.A.: Introduction to Lattices and Order. Cambridge University Press 1990.

[Du87] Duquenne, V.: Contextual implications between attributes and some representation properties for finite lattices. In [GWW87], pp. 213 – 240.

[FKS94] Fischer, B., Kievernagel, M., Struckmann, W.: VCR: A VDM-based software component retrieval tool. Informatik-Bericht 94–08, TU Braunschweig. Submitted for publication.

[Ga87] Ganter, B.: Algorithmen zur formalen Begriffsanalyse. In [GWW87], pp. 241 – 254.

[GWW87] Ganter, B., Wille, R., Wolff, K. (ed.): Beiträge zur Begriffsanalyse. B.I. Wissenschaftsverlag 1987.

[Gr94] Grosch, F.-J.: Ein semantikbasierter Architekturentwurfskalkül. Thesis proposal, TU Braunschweig 1994 (in German).

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

[GS93] Grosch, F.-J., Snelting, G.: Polymorphic Components for Monomorphic Languages. Proc. Second International Workshop on Software Reusability. IEEE 1993, pp. 47 – 55.

[KS94] M. Krone, G. Snelting: On the Inference of Configuration Structures from Source Code. Proc. 16th International Conference on Software Engineering, IEEE 1994, pp. 49-58.

[Kr93] Krone, M.: Reverse Engineering of Configuration Structures. Master's thesis, TU Braunschweig, Institut für Programmiersprachen, 1993 (in German).

[Li94] Lindig, C.: Interaktives, rückgekoppeltes Softwarekomponentenretrieval. Informatik-Bericht 94–07, TU Braunschweig.

[Pa94] Parnas, D.: Software aging. Proc. 16th International Conference on Software Engineering, IEEE 1994, pp. 279–290.

[Sk92] Skorsky, M.: Endliche Verbände – Diagramme und Eigenschaften. PhD thesis, Technical University of Darmstadt, Dept. of Mathematics, 1992.

[SGS91] 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.

[SFGKZ94] Snelting, G., Fischer, B., Grosch, F.-J., Kievernagel, M., Zeller, A.: Die inferenzbasierte Softwareentwicklungsumgebung NORA. Informatik – Forschung und Entwicklung 9(3). pp. 116 – 131, September 1994 (in German).

[STT81] 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.

[Wi82] Wille, R.: Restructuring Lattice Theory: An Approach Based on Hierarchies of Concepts. In: I. Rival (ed.) Ordered Sets. Reidel 1982, pp. 445 – 470.

[Wi83] Wille, R.: Subdirect decomposition of concept lattices. Algebra Universalis 17 (1983), pp. 275 – 287.

[Wi85] Wille, R.: Tensorial decomposition of concept lattices. Order 2 (1985), pp. 81 – 95.

[Wi89a] Wille, R.: Lattices in data analysis: how to draw them with a computer. In H.H. Bock (ed.): Classification and related methods of data analysis. North Holland 1989, pp. 33 – 58.

[Wi89b] Wille, R.: Geometric Representation of Concept Lattices. In: O. Opitz (ed.): Conceptual and Numerical Analysis of Data. Springer 1989, pp. 239 – 255.

[Wi92] Wille, R.: Concept Lattices and Conceptual Knowledge Systems. Computers & Mathematics with Applications 23 (1992), pp. 493 – 515.

[Wi87] Wille, R.: Bedeutung von Begriffsverbänden. In [GWW87], pp. 161 – 212.

[WG93] Wille, R. and Ganter, B.: Mathematische Theorie der formalen Begriffsanalyse. Lecture notes, Technical University of Darmstadt, Dept. of Mathematics, 1993 (in German).

[ZS94] Zeller, A., Snelting, G.: Handling version sets through feature logic. Informatik-Bericht 94–04, TU Braunschweig. Submitted for publication.