Reengineering of Configurations Based on Mathematical Concept Analysis

GREGOR SNELTING
Technische Universität Braunschweig

We apply *mathematical concept analysis* to the problem of reengineering configurations. Concept analysis 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—offers remarkable insight into the structure and properties of possible configurations. The lattice not only displays fine-grained dependencies between configurations, but also visualizes the overall quality of configuration structures according to software engineering principles. In a second step, interferences between configurations can be analyzed in order to restructure or simplify configurations. Interferences showing up in the lattice indicate high coupling and low cohesion between configuration concepts. Source files can then be simplified according to the lattice structure. Finally, we show how governing expressions can be simplified by utilizing an isomorphism theorem of mathematical concept analysis.

Categories and Subject Descriptors: D.2.6 [**Software Engineering**]: Interactive Programming Environments; D.2.7 [**Software Engineering**]: Distribution and Maintenance—*restructuring*; *version control*; 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"

A preliminary version of parts of this article appeared in the Proceedings of the 16th International Conference on Software Engineering, May 1994.
Author's address: Abteilung Softwaretechnologie, Technische Universitat Braunschweig, Gaußstrasse 17, D-38106 Braunschweig, Germany; email: snelting@ips.cs.tu-bs.de.

[Parnas 1994]. However no method for reengineering program families was as yet available, let alone tool support for restructuring.

At the same conference, we presented a first step toward a theory and tools for configuration restructuring [Krone and Snelting 1994]. Based on *mathematical concept analysis* [Wille 1982], we have shown how configuration structures can be inferred from existing source code and how interferences between configurations can be detected.

In this article, we describe in detail how to extract configuration structures from existing source code and how to interpret the obtained structures. Source files must adhere to the paradigm "version selection by #ifdef." The structure of the configuration space is given in the form of a concept lattice, which is computed from the relation between code pieces and their governing expressions. When visually displayed, the lattice offers remarkable insight into properties of configurations and into relationships between configurations.

We then present an algorithm for detecting interferences between configurations. An interference means that two configurations have common code where they should not. Based on the lattice structure and interference analysis, source file simplification can be done by "amputating" parts of the configuration space. Finally we show how an isomorphism theorem from concept analysis makes it possible to simplify governing expressions with respect to inferred properties of the configuration space.

Before we begin to explain our reengineering tool NORA/RECS in detail, we would like to give an overview; this might also serve as an extended abstract for hurried readers. The overview will contain several informal definitions, for which the exact formal definitions are provided later in the text.

1.1 Configuration Reengineering

Software configuration management is the discipline of organizing and controlling the evolution of software systems [Tichy 1988]. A *configuration* of a software system is a collection of elements (software components, code pieces, modules, . . .) which fulfill a particular purpose. Typically, a configuration meets the needs of a particular client or platform. Therefore, configuration management must—among other tasks—be able to identify software components or code pieces which have certain features and to build a complete system from selected components. Several sophisticated configuration management systems have been developed recently—for example, Adele [Estublier and Casallas 1994], Shape [Mahler 1994], and Clearcase [Leblang 1994].

This article, however, is concerned with *reengineering* of configuration structures from existing source code. Therefore we do not make assumptions about the underlying configuration management model. We just assume that we are given a set of software *objects*, as well as a set of features or *attributes*, where each configuration is characterized by a set of attributes. We do not care whether objects are syntactically code pieces,

components, or modules; whether versions of objects are revisions or variants; and what architecture the system in question might have. We do not make any assumptions about the structure of the underlying set of attributes either; on the contrary, discovering such a structure (if it exists) is part of the reengineering process.

The only basic assumption we make is that any configuration consists of objects and is selected by a set of attributes. For the time being, we consider only simple attributes with a dual defined/undefined semantics; later we will see how more complicated attributes can be subsumed by the basic model. Therefore, the process of version selection can abstractly be described by a configuration function as follows:

Definition 1.1.1. Let O be a set of software objects. Let A be a set of attributes. A *configuration function* is a mapping $K : 2^A \rightarrow 2^O$. The set of all possible configurations $K(2^A) \subseteq 2^{2^O}$ is called the *configuration space*, and for any attribute set $V \subseteq A$, the specific object set $P = K(V)$ is called a *configuration* selected by V.

This definition does not say how K is determined, and indeed for the configuration management systems mentioned above, K will take a very specific form. Generally, the approach described in this article works whenever version selection can abstractly be described by a configuration function.

Definition 1.1.2. Let $X, Y \subseteq A$. We say X *implies* Y if $K(X) \subseteq K(Y)$. X and Y are called *interfering*, if $K(X) \cap K(Y) \neq \emptyset$.

The notions of configuration implication and interference will be more fully explained later. We can now specify the task of configuration reengineering as tackled in this article: given a configuration function, determine all configurations (that is, all $K(X)$ for $X \subseteq A$) and all implications or interferences between them. Furthermore, visualize the overall structure of the configuration space.[1]

1.2 Configuration Management by Preprocessing

In this article, we restrict ourselves to a simple and widely used version selection system—namely, the C preprocessor (CPP). This restriction is introduced for pragmatic reasons. A lot of code sticking to "configuration management by preprocessing" is around. For example, both the X-Window system and the GNU software (gcc, g++, flex, bison, rcs, etc.) use CPP for configuration building; thus there is enough raw material for reengineering. Adapting our approach to more modern configuration management systems is possible whenever they can be modeled by configuration functions—this would require development of a new front end.

Configuration management by preprocessing is very simple. Configura-

[1]The latter task is usually called reverse engineering—we do not distinguish between reengineering and reverse engineering of configurations.

tion-dependent source code pieces are enclosed in #if . . . #endif brackets:

```
#if E
. . .
#endif
```

E is a so-called *governing expression*, that is, a boolean expression which may in particular contain atomic formulas like defined(X) or defined(Y) for preprocessor symbols X, Y. The often-used #ifdef X . . . #endif is a short cut for #if defined(X) . . . #endif.[2] CPP also offers #else, #elif E, and #ifndef X constructs. Governing expressions may contain negations, conjunctions, and disjunctions, and #ifdef may be nested.

When starting the compiler, CPP symbols may be defined (e.g., cc -Dultrix prog.c). CPP will evaluate all governing expressions and will include a code piece only if its governing expression evaluates to true. Code pieces governed by a nested #if are selected only if the surrounding code is selected as well. Thus by defining CPP symbols a configuration is determined, and the appropriate code pieces are selected and compiled.

There are two basic methods for using 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
```

If the #ifdef/#endif enclosing a code piece o contains a CPP symbol a, we say either o depends on a, or o is *governed* by a. In the first example, the code piece is governed by SUN, ULTRIX, AIX, whereas in the second example the code piece is governed by HAS_BCOPY and HAS_NFS.

CPP "#ifdef" statements may contain complicated boolean expressions; thus existing programs do not stick to the two basic CPP schemes. 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 configurations (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" and "#define"

[2]The current CPP standard treats X as equivalent to defined (X) & & X != 0.

```
#if  (!defined(SVR4)  ||  !defined(__STDC__))  &&  !defined(sgi)  &&  !de-
fined(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
```

Fig. 1. X-Window tool "xload.c."

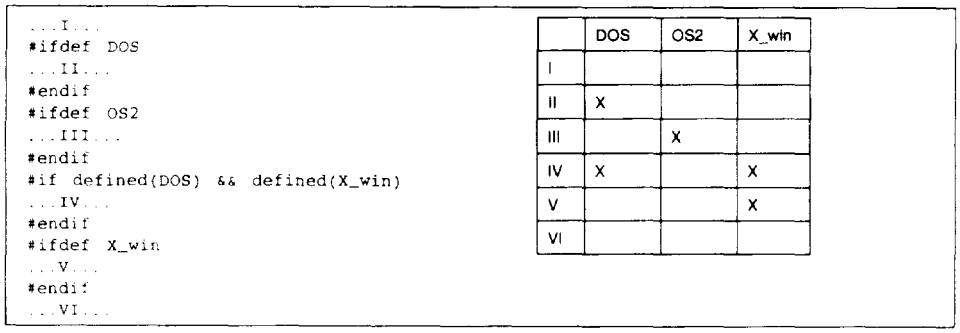
statements are nested, resulting in a rather incomprehensible source text. Even experienced programmers will have difficulties to obtain some insight into the configuration structure, and when a new configuration variant is to be covered, the introduction of errors is very likely.

Nevertheless, CPP adheres to the general configuration scheme described above: the objects are code pieces in a source file; the attributes are the CPP symbols; and the configuration function is just CPP itself. Given a set of defined CPP symbols, CPP will select the corresponding code pieces and—since there is a total order, namely, the textual order defined for code pieces in CPP files—automatically builds the corresponding configuration. As we will see later, even negated CPP symbols, disjunctions, etc. can be interpreted as attributes; thus the notion of a configuration function is powerful enough to describe the full CPP semantics.

Throughout the rest of this article, we stick to the equations *object = code piece (in a CPP source file)* and *attribute = (simple or derived) CPP symbol.* Code pieces are usually identified by line number intervals, but in several examples we will use roman numerals as placeholders for code pieces.

1.3 Configuration Tables

The CPP configuration function $K = K_{CPP}$ can be represented by a two-dimensional boolean array, the so-called configuration table. The table



```
...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
V			X
VI			

Fig. 2. A small code fragment and its configuration table.

$T = T_{K_{CPP}}$ is indexed with objects and attributes, and $T[o, a] = \text{true}$ indicates that the object o depends on attribute a: any $V \subseteq A$ selecting o must contain a.

Definition 1.3.1. Let K be a configuration function. The *configuration table* $T_K : O \times A \to \mathbf{B}$ is defined by $T_K[o, a] = \text{true} \Leftrightarrow \forall V \subseteq A : o \in K(V) \Rightarrow a \in V$.

Figure 2 presents a very small source text and its configuration table. Section 3 will describe how configuration tables are constructed from arbitrary CPP source files.

Conversely, any configuration table T determines a configuration function: for $o \in O$, let $\sigma(o) = \{a \in A \mid T[o, a] = \text{true}\}$. Then $K_T(V) = \{o \in O \mid \sigma(o) \subseteq V\}$. This means that an object is selected by V if all governing attributes are in V. For $K = K_{CPP}$, it is easy to show that $K_{T_K}(V) = K(V)$ (note that this identity captures the CPP semantics).

Thus for every configuration function K a configuration table T_K can be defined, and every configuration table T induces a configuration function K_T. But $K = K_{T_K}$ does not always hold. Therefore, not every configuration function can correctly be represented by a configuration table. Indeed, there are only $2^{|O| \cdot |A|} = 2^{|O| \cdot |A|}$ configuration tables, but there are $(2^{|O|})(2^{|A|})$ configuration functions (many of them pathological). Configuration functions defined by configuration tables enjoy special properties, e.g., monotony: $X \subseteq Y \Rightarrow K(X) \subseteq K(Y)$.

For Adele, Shape, and Clearcase, it is possible to describe their version selection mechanism by configuration tables. Adele, for example, is based on software components which have several attributes of the form *name* = *value*; version selection is done with a boolean expression over such attributes, and a component is selected if the selection expression evaluates to true for the component's attribute values. Many-valued attributes can be described by so-called scaled configuration tables (which have one column for every value of an attribute), and complex selection expressions can be treated with the same techniques as complex governing expressions (see Section 3).

For strange configuration functions K, the notion of a configuration table capturing dependencies of code pieces on attributes might not be adequate.

Nevertheless, a relation between objects and attributes can always be established: $T_K[o, a] \Leftrightarrow \exists V \subseteq A : a \in V \wedge o \in K(V)$. This can be read as "$a$ influences o" and explains why our approach can be used for quite different configuration management models. Using scaled configuration tables, it is even possible to handle structured object or attribute spaces. But note that "influence" is weaker than "dependency"; thus for pathological configuration functions which cannot be coded as configuration tables, the information extracted from the "influences" relation is not as precise as for CPP source files. For the rest of this article, we ignore pathological configuration tables.

1.4 Concept Lattices

We have seen that it is difficult to understand source files like "xload.c." Fortunately, there is a method, called *formal concept analysis*, which allows for the reconstruction of semantic structures from raw data as given in our case. Formal concept analysis can be used whenever a relation between certain objects and attributes is given. The basic idea goes back to G. Birkhoff, who observed in 1940 that a complete lattice can be associated with every binary relation, which offers remarkable insight into the structure of the relation [Birkhoff 1983]. The lattice and its underlying relation can be reconstructed from each other; hence the method is similar in spirit to Fourier analysis.

Later, the universal algebra group at the Technical University of Darmstadt elaborated Birkhoff's basic result. Today, there is not only a sophisticated mathematical theory, but also several extensions of concept analysis which can be applied to more complicated problem domains. Concept analysis has been applied to problems such as classification of finite lattices, analysis of Rembrandt's paintings, or behavior of drug addicts. It can also be used as a knowledge acquisition mechanism, and the structure theory of concept lattices provides for even more powerful analysis methods.[3]

In this introduction we will only explain some basic notions. 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 concepts form a complete lattice; hence the lattice structure imposes a partial order on concepts (more specific versus more general), and for two concepts, supremum and infimum exist.

In our case, objects are code pieces; attributes are CPP symbols; and the object-attribute relation is given by the configuration table. Concepts then correspond to (partial) configurations and are computed by our tool NORA/RECS. In particular,

[3]The characterization of (mental and mathematical) concepts has also been studied by other authors (e.g., see van Mechelen et al. [1993]), but without the lattice-theoretic underpinnings which give formal concept analysis its power.

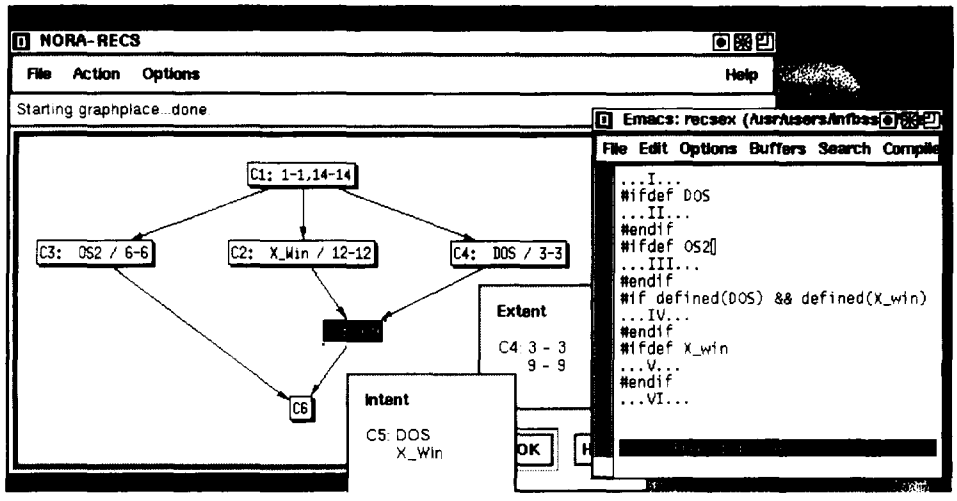


Fig. 3. Concept lattice for Figure 2.

—for each configuration, its *extent* (the code pieces which make up the configuration) and *intent* (the attributes which govern the configuration) are computed;
—all *implications* between configurations are computed, where an implication 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 configurations 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 3. This simple lattice already demonstrates basic principles. The lattice elements are named (C1, C2, . . . , C6) and are labeled with code pieces or CPP symbols (or both). Code pieces are given in form of line number intervals in the source code; the source file is displayed in the emacs window. If a concept labeled with code piece o is below or equal a concept labeled with CPP symbol a, then o depends on a. For example, line 9 is governed by DOS, as C5 (which is labeled with source line interval 9–9) is below C4 (which is labeled DOS). Line 3 is also governed by DOS, as C4 is also labeled with interval 3–3. As C5 $\leq$ C2, line 9 is also governed by X_win.

In fact, a source code piece occurring as label of a concept c is governed by all CPP symbols appearing in concepts *above* c. These are called the *intent* of c. By clicking on c, its intent can be displayed. Indeed, the intent box of C5 shows that line 9 is governed by DOS and X_win (and no other

CPP symbol). Complementarily, a CPP symbol occurring as a label of c governs all code pieces appearing in concepts *below* c. These are called the *extent* of c. The extent box for C4 shows that DOS governs lines 3 and 9 (and no other).

The code piece labels of the top element present the code which is not governed by any CPP symbol (in the example, lines 1 and 14), while the symbol labels of the bottom element present those CPP symbols which do not govern anything (none in the example). The extent of the top element consists of all code pieces, while the intent of the bottom element consists of all CPP symbols in the source code. Code governed by more than one CPP symbol shows up as an infimum in the lattice, while CPP symbols governing more than one code piece show up as a supremum.

CPP symbols which are concerned with different configuration aspects (e.g., window system variants versus operating system variants) are called orthogonal, and code which is governed by orthogonal symbols indicates an *interference*. In the example, line 9 is governed by both DOS and X_win, indicating that window system and operating system aspects are not clearly separated. Adopting traditional terminology, we call this phenomenon *strong coupling* between orthogonal configuration aspects. Note that only a human can decide whether code shared by various configurations reveals a harmful interference or a beneficial reuse. Section 4 will discuss interferences and coupling in depth.

1.5 Source Code Simplification

For very chaotic configuration spaces, restructuring is appropriate, and later we will see how concept analysis can help in restructuring. The first step is perhaps to reduce the size of the program family (called "amputation" by Parnas [1994]). 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. We will see later that amputation can be implemented through *partial evaluation* of CPP files. Under the assumption that certain CPP symbols are (or are not) defined, while for others this information is missing, governing expressions can be simplified, and perhaps some code pieces disappear completely.

Amputation, or source file simplification, can be used if certain configurations are no longer needed. Furthermore, special "problematic" versions can be generated from interferences; this allows for a more detailed inspection of configuration problems on the source code level.

Lattice theory also offers a clever way for simplification of governing expressions. The lattice is generated by the so-called *irreducible elements* alone. These irreducible elements can be used to generate new governing CPP symbols, which lead to simpler governing expressions. The method is much more powerful than just boolean simplification, as it takes lattice-specific information into account.

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*.

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

$$X_1 \subseteq X_2 \Rightarrow \sigma(X_2) \subseteq \sigma(X_1) \quad \text{and} \quad Y_1 \subseteq Y_2 \Rightarrow \tau(Y_2) \subseteq \tau(Y_1);$$

that is, both mappings are *antimonotone*;

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

as well as

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

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; this means that both $\sigma \circ \tau$ and $\tau \circ \sigma$ are *closure operators*. For an index set I and $X_i \subseteq O$, $Y_i \subseteq A$

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

A (formal) *concept* is a pair (X, Y), where $X \subseteq O$, $Y \subseteq A$, $Y = \sigma(X)$, and $X = \tau(Y)$. Hence, a concept is characterized by a set of objects (called its *extent*) and a set of attributes (called its *intent*) such that (1) all objects have all attributes and (2) 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 4, where permutations of lines or columns 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, $B(O, A, P) = (B(O, A, P), \leq)$ is a complete lattice.

	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			

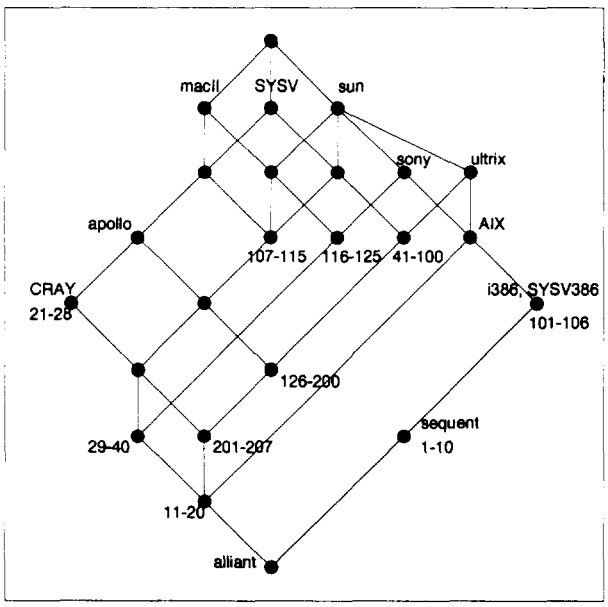


Fig. 4. A configuration table and its concept lattice.

BASIC THEOREM FOR CONCEPT LATTICES. *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*

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

and

$$\bigvee_{i \in I} (X_i, Y_i) = \left(\tau\left(\sigma\left(\bigcup_{i \in I} X_i \right) \right), \bigcap_{i \in I} Y_i \right).$$

This theorem says that in order to compute the infimum (greatest common subconcept) of two concepts, their extents must be intersected and their

intents joined; the latter set of attributes must then be enlarged in order to fit to the object set of the infimum. Analogously, the supremum (smallest common superconcept) of two concepts is computed by intersecting the attributes and joining the objects.

The lattice structure allows for 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, respectively, the concept labeled with attribute a, is

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

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:

$$ext(c) = \bigcup_{c' \le c} \omega(c') \quad \text{and} \quad int(c) = \bigcup_{c' \ge c} \alpha(c').$$

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 $C = (\tau(A), A)$ and $D = (\tau(B), B)$, and $C \le D$, 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 [Duquenne 1987].

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 [Wille 1982]. Thus concepts do not "breed" new concepts; there is no proliferation of virtual information.

The basic theorem was already discovered by Birkhoff [1940]. Later, Wille and Ganter [1993] expanded Birkhoff's result. Wille gave characterizations of all concept lattices, developed their structure theory, and invented scaled contexts, a method for handling nonflat attribute spaces. Ganter, besides other contributions, developed efficient algorithms (see Section 2.3). The interested reader should consult Daveys and Priestley [1990], which contains a chapter on elementary concept analysis. Those who are interested in more advanced theory might wish to consult Wille's lecture notes [Wille and Ganter 1993] or Ganter's advanced introduction [Ganter 1995].

2.2 Interpretation of Concept Lattices

Figure 4 presents a (fictitious) example of a configuration table, where source code pieces are given in form of line number intervals. A cross in the table for code piece o and preprocessor symbol a means that o will only be

included in a configuration if a is defined. For the time being, we assume that only simple governing expressions of the form "if defined(x) && defined(Y) && . . ." are used. This simplification will be dropped in Section 3.

In the corresponding concept lattice, a configuration concept is a subconcept of another concept, if it has a smaller extent (i.e., the configuration has fewer 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. The lattice elements are labeled with code pieces or CPP symbols (or both). If an element labeled with code piece o is below or equal to an element labeled with CPP symbol a (that is, $\gamma(o) \leq \mu(a)$), then o depends on a.

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}). Indeed, Figure 4 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 possible interpretation of a concept lattice: it can be seen as a *hierarchical conceptual 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 analogous 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 4, μ(sony) $\wedge$ μ(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*," or equivalently "*If sun is undefined, defining sony or ultrix has no effect.*" Such knowledge is not easily extracted by hand from a source file like "xload.c"! This example demonstrates the second main possible interpretation of a concept lattice: it represents all implications (or 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 $K(S) = \{o \in O \mid \sigma(\{o\}) \subseteq S\}$. 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.

One can imagine $K(S)$ as a set of downward paths or "strings" starting from the top element; the end of a string is labeled with a selected code piece, and all "nodes" along the string are labeled with CPP symbols in S. The configuration function can more easily be computed from the original configuration table; the lattice representation is intended for other purposes.

2.3 Construction of the Concept Lattice

In order to give the reader an idea of 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),

$$Y = \sigma(X) = \sigma\left(\bigcup_{o \in X} \{o\}\right) = \bigcap_{o \in X} \sigma(\{o\}).$$

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 $o \in O$, 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 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.

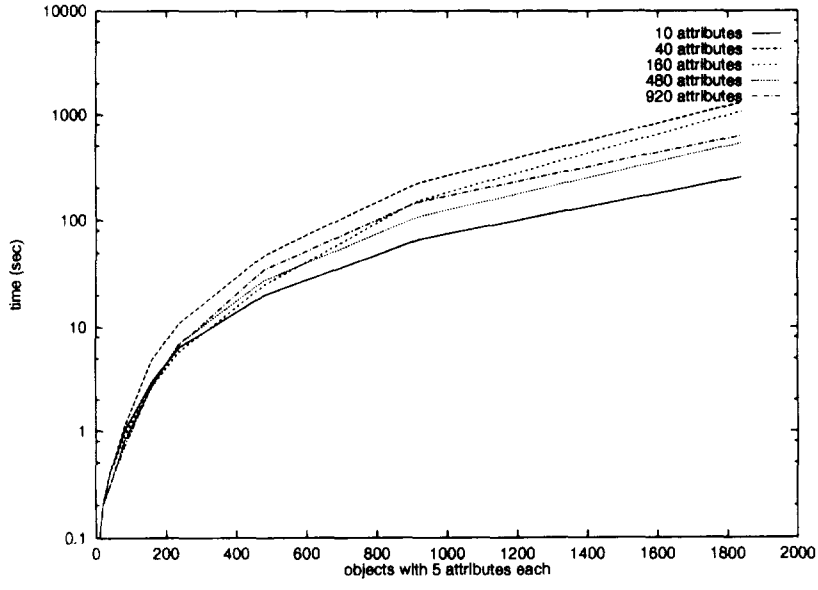
The overall complexity depends on the number of lattice elements. The largest lattices for contexts of size $n \times n$ have 2^n elements; these lattices are isomorphic to finite boolean algebras freely generated by n elements. Thus the worst-case running time of the construction algorithm is exponential 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 reasonably large contexts.

Ganter [1987] has found a more efficient algorithm which avoids tracking the elements and suprema, but is more difficult to understand. This algorithm is used in the Darmstadt implementation of concept analysis. It has recently been reimplemented for NORA. Lindig [1995] presents some empirical data on this implementation (Figure 5), which have been obtained from random contexts. The first graph shows the time for lattice construction as a function of the number of objects; for 1000 objects (code pieces) the analysis needs 100 seconds 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. FROM SOURCE CODE TO CONFIGURATION LATTICES

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 analyzing interferences and sublattices. Source code simplification corresponds to selection of a specific sublattice; upon the restructurer's request, the source file is transformed accordingly. The source language is arbitrary, but the input file must adhere 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 analyzes the latter, and constructs a configuration table according to the rules described below;
(2) *kernel*: the kernel 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 analyzed, and sublattices can be selected;



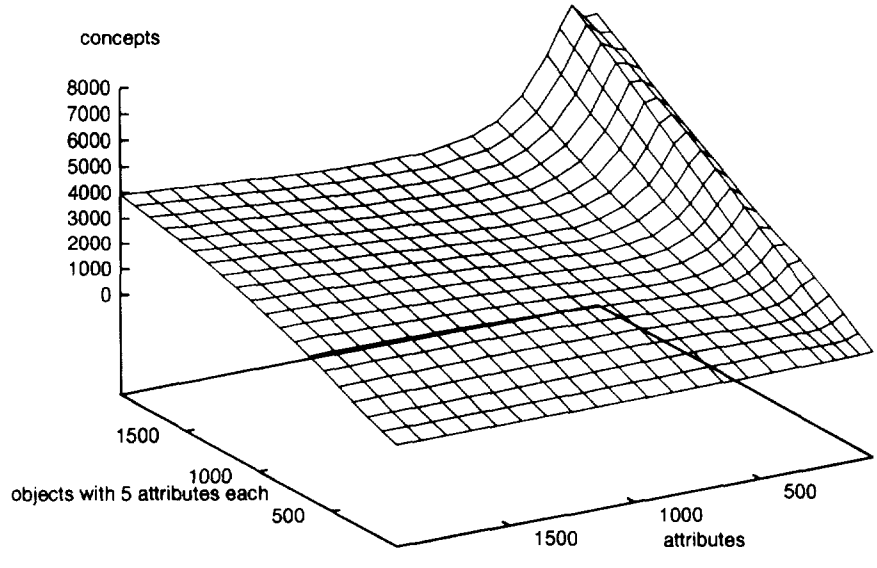


Fig. 5. Time complexity and lattice size for random contexts.

(5) *back end* (optional): the source code is simplified according to certain lattice properties.

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 section describes the first two 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. According to the basic model of a configuration function, configurations only depend on positive attributes. Therefore, complex governing expressions must be transformed into sets of simple attributes. The reader should be aware that the often-used notion of a "feature" of a configuration is not identical to the term "attribute" in our setting. A feature may be a complex property, while an attribute describes a simple, positive fact. For example, a configuration can have the feature that it is governed by CPP symbol X, whether by including or excluding code pieces of the original source file. As we will see, this is described by two attributes: one has the meaning "it is true that X is defined," while the other means "it is true that X is not defined."

We will now describe how to construct configuration tables from source files like "xload.c"; due to complex CPP expressions and nested "#ifdef" statements, this process is not trivial. After construction, every complex governing expression has been "compiled" to a set of simple, positive attributes. 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 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 . . .
```

	. . .	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 . . .
```

	. . .	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
```

	. . .	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 "dichotomized 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 slightly 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 configurations 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.[4] Afterward, all expressions are of the form $(A_1 \vee A_2 \vee \cdot \vee A_i) \wedge (B_1 \vee B_2 \vee \cdot \vee B_j) \wedge \ldots (C_1 \vee C_2 \vee \cdot \vee C_k)$, 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 $A_1 \vee A_2 \vee \ldots A_i$, 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 $A_1 \vee A_2 \vee \ldots \vee A_i$).

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 expressions will show up as concept labels.

NORA/RECS does not provide a fine-grained analysis of arithmetic and relational expressions. But at least it distinguishes between #if A == 42 and #if defined(A), for example. According to the CPP semantics, A must be defined if its value is 42. Therefore, an implication is added in this (and similar) cases: whenever a cross is placed in the column labeled $A == 42$, another one is placed in the column for A as well.

Nested #ifdef, #define, and #undefine Statements. The treatment of nested "#ifdef" 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	...			...

[4]Note that normalizing boolean expressions can have exponential time complexity. But, in practice, even for source files like "x.load.c" the governing expressions are small enough to be tractable.

line 180 ff:

```
#if has_NFS || bad_unlink
        int
un_link(s)
        char const *s;
/*
* Remove S, even if it is unwritable.
* Ignore unlink() ENOENT failures; NFS generates bogus ones.
*/
{
#       if bad_unlink
                int e;
                if (unlink(s) == 0)
                        return 0;
                e = errno;
#               if has_NFS
                        if (e == ENOENT)
                                return 0;
#               endif
                if (chmod(s, S_IWUSR) != 0) {
                errno = e;
                return -1;
                }
#       endif
#       if has_NFS
                return unlink(s)==0 || errno==ENOENT ? 0 : -1;
#       else
                return unlink(s);
#       endif
}
#endif
#if !has_rename
#  if !has_NFS
#       define do_link(s,t) link(s,t)
#  else
        static int
```

	bad_unlink	has_NFS	bad_unlink OR has_NFS	! has_NFS	has_rename	! has_rename	...
...							
181 - 188			X				
190 - 193	X		X				
195 - 196	X	X	X				
198 - 201	X		X				
204 - 204		X	X				
206 - 206				X			
212 - 212				X		X	
214 - 214		X	X			X	
...							

Fig. 6. Source code and configuration table fragments of the RCS stream editor "rcsedit.c."

Nested "#define" and "#undefine" statements can also be treated (for details, see Krone [1993]). Experience has taught us that a "#define" is seldom used for configuration management, but for definition of constants or inline functions instead. Hence the current implementation of NORA/RECS ignores "#define" and "#undefine" statements.

As an example for the construction process, consider the stream editor of the RCS system [Tichy 1985]. This program is 1656 lines long and uses 21 CPP symbols for configuration management. Figure 6 presents a piece of source code, starting at line 180. The resulting configuration table is also presented in Figure 6; the configuration structure will be analyzed in Section 5. Note the typical treatment of disjunctions, negations, and "#else."

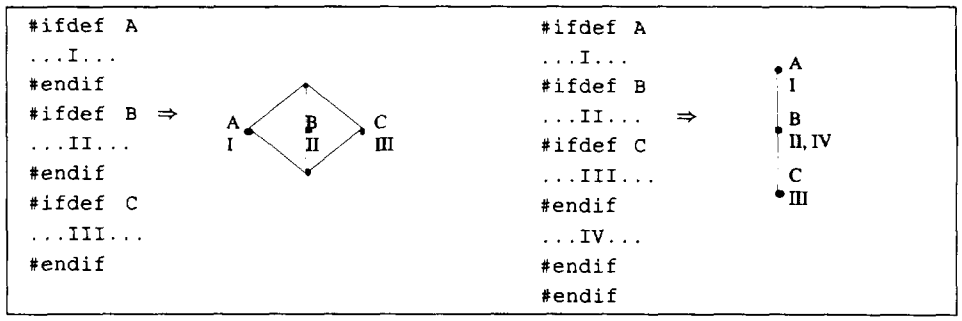


Fig. 7. An antichain and a chain.

3.2 Basic Patterns in Configuration Lattices

We will now explain some characteristic patterns in concept lattices and provide some basic examples. The purpose of this section is to demonstrate how certain source code patterns show up in the lattice. The configuration table is only an intermediate representation and invisible to the user of NORA/RECS.

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 a top and bottom element is called *flat* (left-hand side of Figure 7).

A chain in a lattice is a set of elements $c_1 < c_2 < c_3 < \ldots$ which are mutually comparable. In our case, chains result from nested "#ifdef" statements. Note that, in concept lattices, a chain can be interpreted as a sequence of *implications*. For example, in the right-hand side of Figure 7 any code piece which depends on C (i.e., code piece III) also depends on B, and any code which depends on B (i.e., code pieces II, IV) also depends on A. According to Section 2.1, this is written as $C \Rightarrow B \Rightarrow A$.

Suprema and Infima. A supremum which is not the top element indicates that two code pieces are governed by the same "superordinate" CPP symbol (left-hand side of Figure 8). Simple disjunctions also show up as suprema in the concept lattice (right-hand side of Figure 8). Note that any supremum consists of two chains and an antichain, and the above explanations for chains and antichains still hold.

An infimum which is not the bottom element indicates that a code piece is governed by two different CPP symbols (left-hand side of Figure 9). Infima may indicate interference (see below). An infimum may also result from disjunctions (right-hand side of Figure 9).

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 (Figure 10).

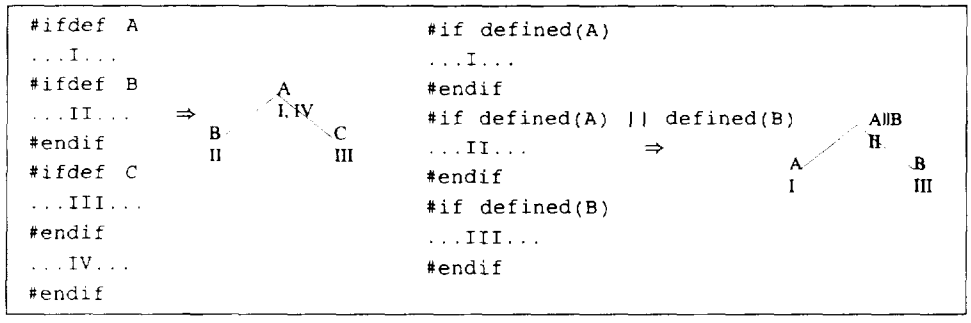


Fig. 8. Two suprema.

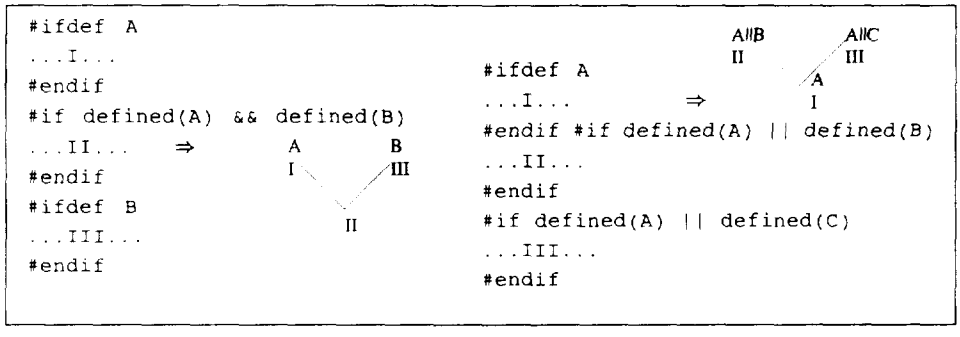


Fig. 9. Two infima.

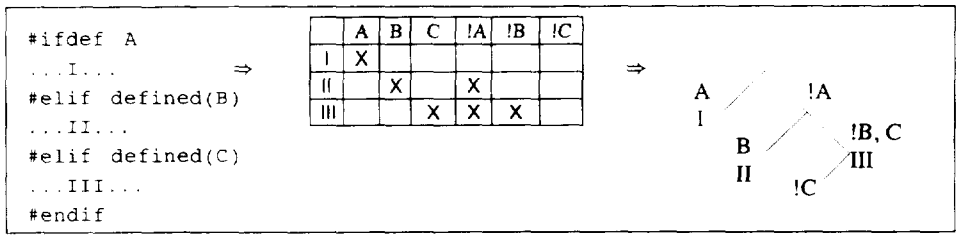


Fig. 10. A cascade.

4. QUALITY OF THE CONFIGURATION SPACE

The concept lattice not only displays fine-grained dependencies between configurations. It also provides good insight into the overall quality of the configuration space, as will be explained in this chapter.

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. In traditional software engineering, *low coupling* and *high cohesion* are considered important criteria for good design, which help to achieve separation of concerns and anticipation of change. Coupling measures the interdependence between

modules, while cohesion means that the elements of a module are related strongly. If there is low coupling between modules and high cohesion within modules, modification of (the implementation of) one module does not influence the behavior of others. We use the notions of coupling and cohesion also for configurations. Coupling between configurations means that they have common code and hence influence each other; the lattice will provide an exact measure for the "strongness" of "configuration coupling." Cohesion within a set of configurations means that all configurations in the set deal with the same configuration aspect.

4.1 Interferences

Coupling arises whenever configurations have common code. As already explained, such code is determined by the infimum operation. This observation gives rise to the following:

Definition 4.1.1. Two preprocessor symbols a and b *interfere* if they have common code, i.e., the intersection of the extent of their configuration concepts is not empty: $ext(\mu(a) \wedge \mu(b)) \neq \emptyset$. Otherwise, a and b are called *mutually exclusive*. Two sets of CPP symbols A and B interfere, if there exist interfering $a \in A$, $b \in B$; otherwise A and B are called mutually exclusive.

Interference is a syntactic phenomenon. Interferences show up whenever CPP symbols have an infimum in the lattice which is not the bottom element, and they can easily be detected automatically.[5] But it is a much more difficult question to decide whether an interference indicates coupling (and therefore should be considered harmful) or whether it just shows that certain combinations of features of the target configuration have specific code.

Definition 4.1.2. 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 versus operating system variants. Two symbol sets A and B are called disjoint, respectively, orthogonal, if this holds for all pairs $a \in A$, $b \in B$.

Disjoint or orthogonal CPP symbols are a semantic phenomenon. Therefore, the definitions of disjoint, respectively, orthogonal, CPP symbols are not completely formal, and such symbols cannot be detected automatically. From the source code alone, it is in general undecideable whether CPP symbols are disjoint or orthogonal. Only the restructurer (if anybody) knows whether CPP symbols can be defined at the same time and whether they deal with independent configuration aspects.

[5]In rare cases, the bottom element may have a nonempty extent and hence induce an interference.

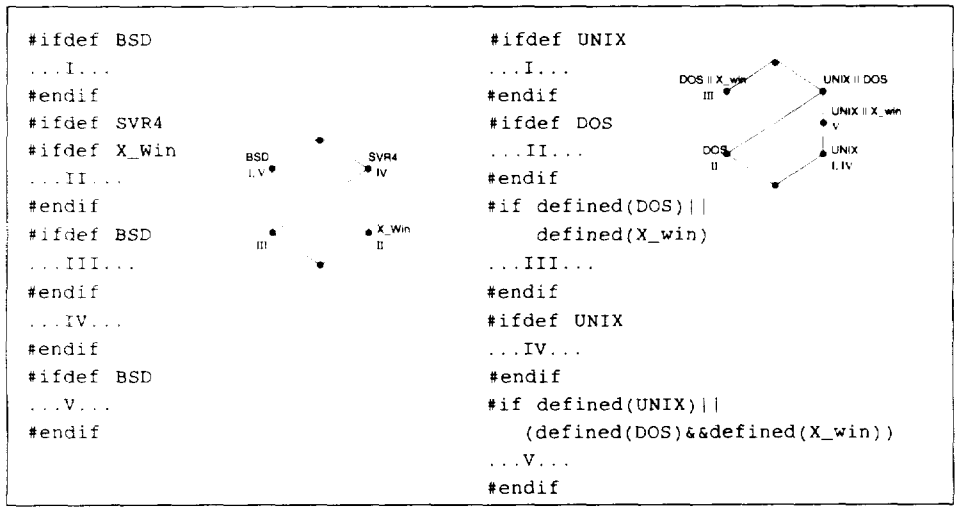


Fig. 11. Two interferences.

Definition 4.1.3. An interference is considered *harmful*, if it is an interference between disjoint or orthogonal CPP symbols (or symbol sets).

In order to decide whether interferences are harmful, the restructurer should contribute some knowledge (or educated guesses) about the intended meaning of CPP symbols. Often, the names of the CPP symbols indicate their meaning. If absolutely nothing is known about the configuration space, and even the names of the CPP symbols do not indicate anything, CPP symbols must not be considered disjoint or orthogonal; thus every interference must be considered potentially harmful.

Interfering disjoint CPP symbols indicate dead code, which can be eliminated. Interfering orthogonal CPP symbols are also very suspicious: from a software engineering viewpoint, such an interference indicates coupling between orthogonal configuration aspects.

Let us consider two examples, which both contain an interference (Figure 11). In the first example, code piece III is governed by both BSD and SVR4, but Berkeley UNIX and System 5 UNIX are to the restructurer's best knowledge incompatible and hence disjoint. Hence code piece III can be deleted. Note that the syntactic knowledge provided by the lattice as well as the semantic knowledge contributed by the restructurer are necessary for this decision. In this example, the names of CPP symbols indicate their meaning. Note that poorly chosen CPP symbol names can easily misguide the restructurer.

The second example in Figure 11 uses "virtual" concept labels DOS || X_win and UNIX || DOS which have been generated by normalizing disjunctions. Therefore, the dependencies of code piece II on these two "CPP symbols" is in fact nonexistent: code piece II is governed by DOS, DOS || X_win, and UNIX || DOS; but according to the absorption law, $DOS \wedge (DOS \vee X\_win) \wedge (UNIX \vee DOS) \equiv DOS$. Dependencies on system-generated basic

disjunctions usually do not show up in the source code directly; they only show that certain combinations of defined CPP symbols select code which might conceal a harmful interference. In the example, the interference merely states that operating system as well as user interface issues show up in both main configurations and that—worse—there is a cross dependency between them.

4.2 Configuration Coupling

We will now discuss how the "configuration-coupling factor" can be extracted from the lattice. If the concept lattice is flat, there are several configurations, but they do not have common code. This means there are no dependencies whatsoever between the possible configurations. Selecting any "feature" of a configuration does not interfere with the selection of any other "feature." Hence there is zero coupling between configurations (left example in Figure 7).

More generally, low coupling is achieved if the lattice is horizontally decomposable. A horizontal decomposition is the inverse operation to a horizontal sum, which will be defined first.

Definition 4.2.1. Let $L_1, L_2, \ldots, L_n$ be lattices. The *horizontal sum* of these lattices is

$$\sum_{i=1}^{n} L_i = \{\top, \bot\} \cup \bigcup_{i=1}^{n} L_i \backslash \{\top_i, \bot_i\}$$

where $\top \geq x$, $\bot \leq x$ for all $x \in \Sigma_{i=1}^{n} L_i$.

Definition 4.2.2. A lattice L is called *horizontally decomposable* if it is the horizontal sum of smaller lattices: $L = \Sigma_{i=1}^{n} L_i$.

A horizontal sum is obtained by removing the top and bottom of the summands and adding new "global" top and bottom elements. Conversely, a horizontal decomposition can be obtained by removing the top and bottom elements of the lattice, determining the strongly connected components of the lattice graph (these are called the *summands*), and adding a new top and bottom element to each summand.

In case a lattice is not horizontally decomposable, it might be that it is decomposable after a small number of interferences have been removed. This gives rise to the following definition.

Definition 4.2.3. A lattice L is called *k-decomposable* if

(1) after removal of $\top$ and $\bot$, the lattice graph is k-connected in the standard graph-theoretic sense;[6]

[6]A graph has connectivity k if the deletion of any $k - 1$ vertices fails to disconnect the graph [Aho et al. 1983].

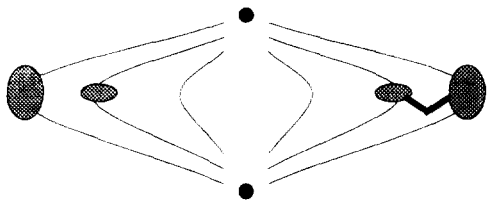


Fig. 12. Horizontal lattice decomposition and a simple interference.

(2) the k vertices $x_1, x_2, \ldots, x_k$ which disconnect the graph are infima: $x_i = a_i \wedge b_i$;
(3) after removal of $x_1, x_2, \ldots, x_k$, the remaining disconnected subgraphs $L_1, L_2, \ldots, L_n$ are—after reattachment of $\top$ and $\bot$ to each L_i—sublattices of L.

If there is a simple interference between two horizontal summands, the lattice is 1-decomposable (Figure 12). If there are k simple interferences between summands, the lattice is k-decomposable. But there can be even worse interferences between two sublattices.

Definition 4.2.4. An *interference of connectivity* k in lattice L consists of k simple interferences, where L is k-connected, and removal of the interferences splits the lattice into only two sublattices.

Such k-interferences are more difficult to detect and to treat than simple interferences (see Section 6).

Ideally, the lattice is horizontally decomposable (i.e., 0-decomposable), where the CPP symbols in each sublattice deal with the same configuration aspect, e.g., operating system variants, while the CPP symbols in different summands are orthogonal or disjoint. This guarantees high cohesion within one sublattice and low coupling between orthogonal configuration concepts. Paths which are glued together in their top or bottom sections are acceptable, but cross arcs between orthogonal sublattices always indicate harmful interference between configurations.

The notion of k-connectivity can be used to give an exact definition of configuration coupling. This definition requires the notion of orthogonal configuration aspects, which—as discussed above—may require human judgment. Therefore the coupling factor cannot be computed automatically, but this does not imply that the notion of "configuration coupling factor" is useless or imprecise.

Definition 4.2.5. Two orthogonal configuration aspects have *coupling factor* k, if

(1) the configuration lattice L is k-connected with horizontal summands $L_1, L_2, \ldots, L_n$;
(2) for each L_i, the CPP symbols in L_i deal with the same configuration aspect;
(3) the CPP symbols in two different summands L_i, L_j are orthogonal.

Flat lattices and horizontal sums are 0-decomposable and hence interference free; they are the optimal structures for configuration management with respect to the configuration-coupling factor. Thus 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.

5. THREE CASE STUDIES

We applied NORA/RECS to several UNIX programs. NORA/RECS uses the well-known Sugiyama algorithm [Sugiyama 1981] for graph layout. There are special layout algorithms for concept lattices which are much more clever (e.g., Wille [1989a; 1989b] and Skorsky [1992]); we plan to integrate these in a future version.

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). Indeed, layouts in Figures 13, 14, and 16 are manually improved.

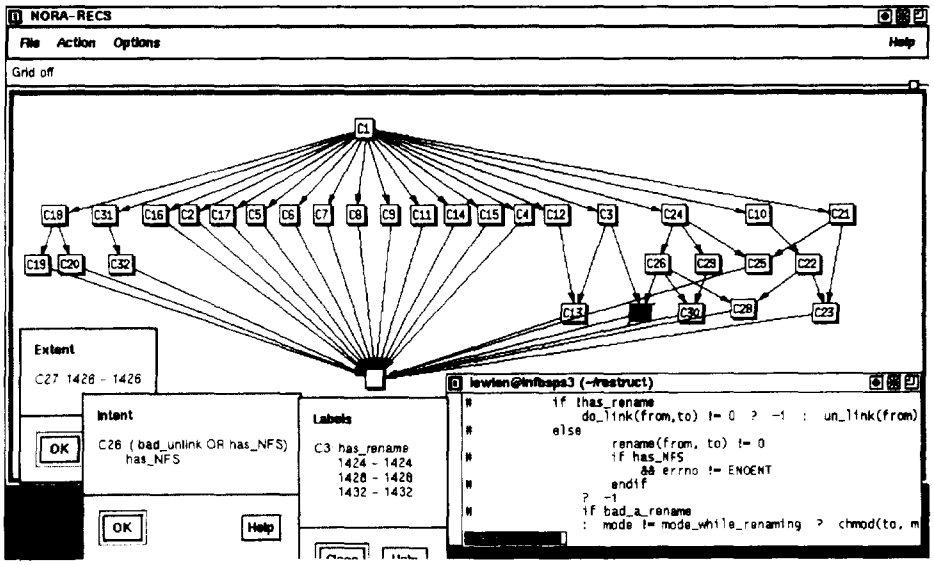
5.1 Example 1: The RCS Stream Editor

Our first example is the stream editor from the RCS system "rcsedit.c." This 1656-line program uses 21 preprocessor symbols for configuration management. Part of the source code and the configuration table of "rcsedit.c" have been presented in Figure 6. Construction of the concept lattice took 0.1 seconds on a Sparcstation 2.

The main window in Figure 13 displays the concept lattice, which has 33 concepts.[7] There are some additional pop-up windows displaying the extent of C27, the intent of C26, and the labels of C3. A piece of source code, namely, lines 1421–1430, is displayed in a separate window. All concept labels are displayed below the picture.

First of all, the labels of top element C1 consist of all code pieces not governed by anything; in "rcsedit" these are quite numerous. There are no unused CPP symbols, as the bottom element C33 has no labels (this, by the way, is the reason why the bottom element has no name). In the left part of the lattice, there are a lot of simple variants, which correspond to specific UNIX features like "has_set_uid" or "has_readlink." For example, C17 shows that "has_readlink" governs lines 1051–1094, 1123, and 1144–1148 (and no other) and that these lines are not governed by any other CPP symbol. Hence for most of the source code, there is no coupling between

[7]The corresponding figure in Krone and Snelting [1994] was produced with the old version of NORA/RECS which did *not* ignore "#define" statements. As explained in Section 3.1, the new version ignores "#define" and "#undefine" statements. Hence, the old lattice in Krone and Snelting [1994] had some additional "virtual" concepts and interferences. The central interference is however still present, and the new lattice is a suborder of the old one. The same remark applies to Figure 13.




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

Fig. 13. Configuration structure of the RCS stream editor.

configurations. The concepts below C24/C10/C21 (concerning networking) form a grid-like cluster. For example, C26 and C29 display the lines solely governed by "has_NFS," respectively, "bad_unlink," and C30 says that lines

195–196 are governed by both (the reader should compare these statements to Figure 6).

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 "labels" box); C26 is labeled "has_NFS"; and C27 is labeled 1426–1426. Thus, line 1426 is governed by both "has_NFS" and "has_rename"—this agrees with the source code in the text window. "has_rename" has to do with the file system, while "has_NFS" controls networking. As these should be orthogonal, the interference is considered harmful. C30 presents a similar interference. It seems that C13 is an interference as well, 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 NFS bug, "rename( )" can in rare cases destroy the RCS file! This problem has been rediscovered by concept analysis, just by analyzing the configuration structure. The example demonstrates that NORA/RECS can track down bugs, even bugs which the programmers would like to keep covered: the comment reads *"An even rarer NFS bug can occur when clients retry requests. . . . This not only wrongly deletes B's lock; it removes the RCS file! . . . Since this problem afflicts scads of UNIX programs, but is so rare that nobody seems to be worried about it, we won't worry either."*

5.2 Example 2: The TC Shell

Our next example is a popular shell, the "tcsh" developed at Berkeley. We have analyzed one of its modules, namely, "sh.exec.c." This program is 959 lines long and uses 24 different preprocessor symbols for configuration management. Construction of the concept lattice took 0.05 seconds on a Sparcstation 2; the lattice has 21 elements. In the concept lattice (Figure 14), singleton attribute or object labels are displayed in the diagram; the others can be looked up by clicking at a concept.

The almost flat lattice shows that there are several features of possible configurations, which however can be selected independently. It seems that there is an interference between concepts C14 and C15. 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. Therefore the configuration structure is perfect according to the criteria described in Section 4.

5.3 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. Construction of the concept lattice took 0.3 seconds on a Sparcstation 2. The resulting lattice has 148 concepts and is shown in Figure 15. It looks pretty chaotic—the program obviously suffers from configuration hacking.

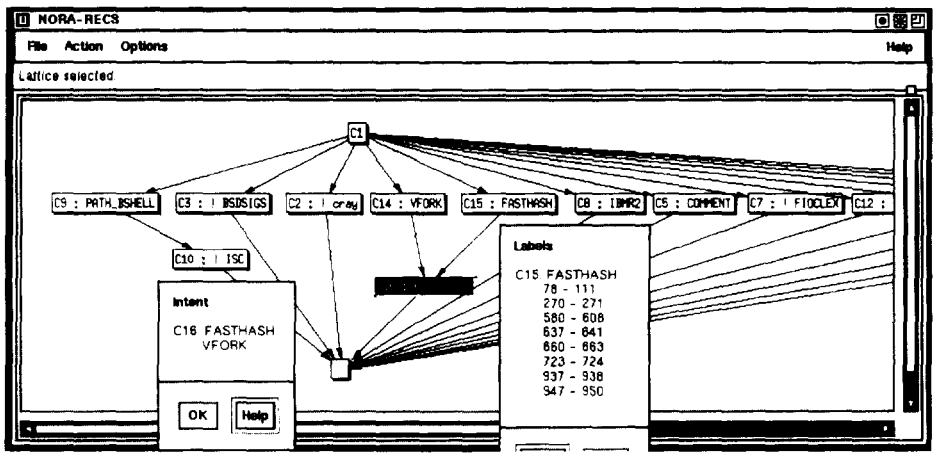


Fig. 14. Configuration structure of tcshell module "sh.exec.c."

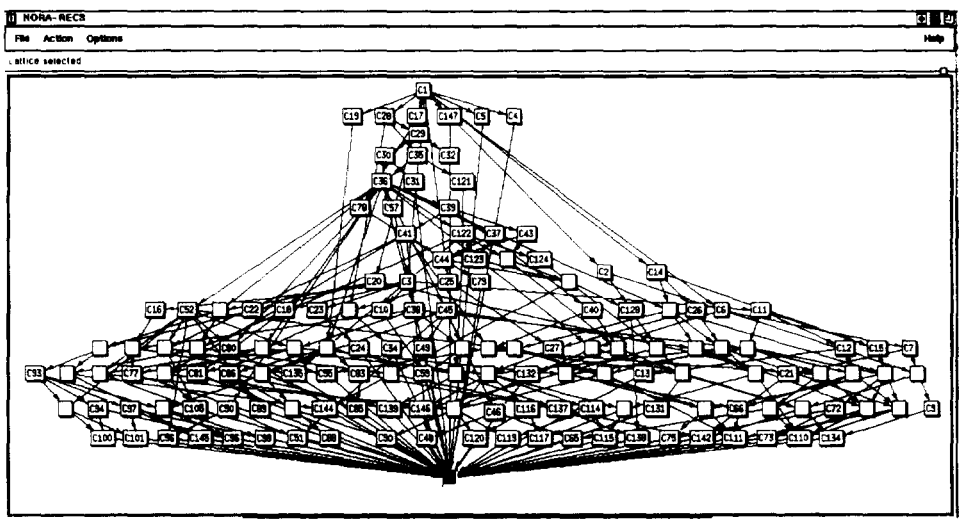
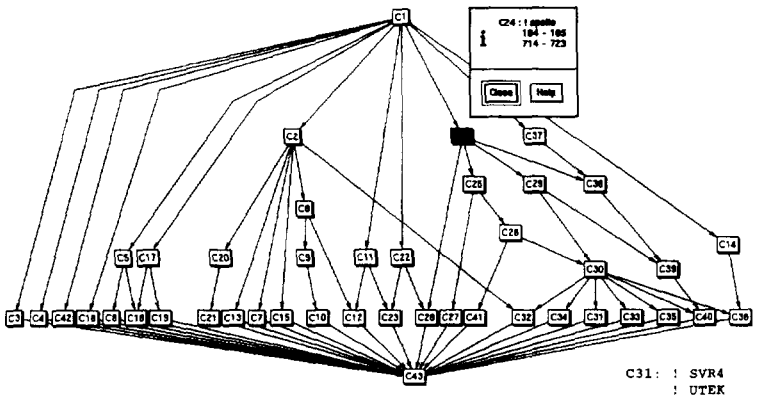


Fig. 15. Configuration structure of "xload.c."

NORA/RECS offers a parameter which controls the maximal nesting depth of #ifdef statements taken into account. For a small value of this parameter, many concepts and dependencies will disappear, but the overall taxonomy of the configuration space is usually still visible.[8] For "xload," the lattice displaying only the top four "#ifdef" nesting levels is given in Figure 16. 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

[8]In fact, the resulting lattice is a sublattice of the original one. The technique is reasonable, as many programs use top-level "#ifdef" statements for important configuration alternatives, whereas lower-level "#ifdef" statements are used to control minor configuration details.



```
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
```

Fig. 16. Top-level configuration structure of "xload.c."

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 a set of nine 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 ≤ C2), and a "not macII, not apollo" sublattice (concepts ≤ 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 for "#ifdef"—namely those which are used for definition of constants or inline functions.

6. LATTICE ANALYSIS AND INTERFERENCE DETECTION

Once the lattice has been constructed and laid out, 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 simplifications can be triggered (see below).

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

6.1 Automatic Interference Detection

As explained above, interferences indicate coupling between configurations. It is therefore of practical importance to detect interferences automatically. In small lattices, interferences are easy to spot manually, but for big lattices (e.g., Figure 15), tool support for lattice analysis is essential.

The interference analysis algorithm incorporated in NORA/RECS tries to horizontally decompose the lattice in such a way that connectivity is minimal. 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 analyzed recursively.

The algorithm for detecting interferences of minimal connectivity implements the definitions from Section 4.2. It proceeds 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 [Aho et al. 1983]. 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* [Aho et al. 1983] of the remaining graph. A bridge between two biconnected components which leads to a $\wedge$-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. 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.

As discussed above, 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 extent and intent of configurations.

6.2 Determining Sublattice Intersections

NORA/RECS offers another approach to lattice analysis, namely, explicit selection of sublattices through their maximal elements.

The restructurer may click at a number of concept nodes $\{c_1, \ldots, c_n\}$, and thereby select the downward suborder $\downarrow \{c_1, \ldots, c_n\} = \{x \mid \exists i : x \leq c_i\}$ (the dual operation of selecting upward suborders is also supported).[9] The downward suborder is then highlighted (or colored) on the screen. It provides interesting information, as the code piece labels in $\downarrow \{c_1, \ldots, c_n\}$ are those which depend on the intent of one of the c_i. For example, selecting the downward suborder $\downarrow$ {apollo, AIX} in Figure 4 displays all code pieces in the "apollo" or "AIX" configurations.

Several downward sublattices may be selected this way, with each highlighted in a different color.[10] 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.

6.3 Example: Analyzing "rcsedit.c"

The configuration lattice of the RCS stream editor was given in Figure 13. After initial horizontal decomposition of the lattice, NORA/RECS immedi-

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

[10]Unfortunately, the current NORA version only supports black and white.

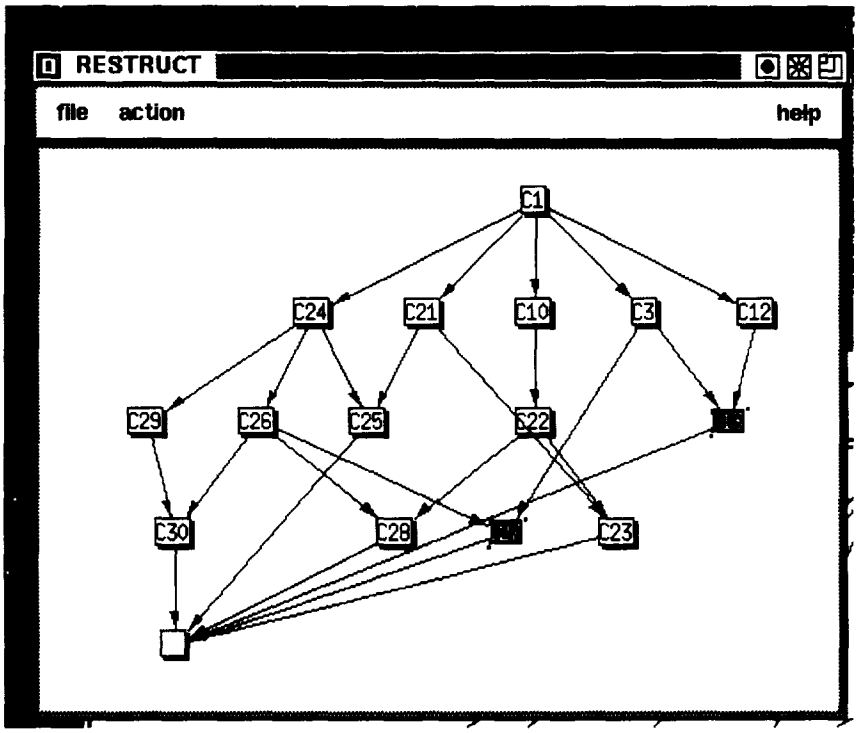


Fig. 17. Automatic interference detection in a sublattice.

ately detects two interferences with connectivity 1, namely, C27 and C13 (Figure 17). Interestingly, C30 (which was criticized as a harmful interference in Section 5.1) is not a top-level interference, as it is part of a grid-like suborder (in fact, it is an interference of connectivity 1 in the sublattice C24/C26/C29/C30; according to the above algorithm, it will only be detected after the $k = 2$-interference C25/C28 has been removed). On the other hand, in Section 5.1 it was argued that C13 is not a harmful interference. This example shows again that each proposed interference must be inspected manually and that interferences can be hidden in sublattices and will only be detected after recursive application of the algorithm.

Should we schedule "rcsedit.c" for restructuring? Lattice analysis shows that restructuring does not make much sense here, because the original program already had a reasonable configuration structure. As the NFS bug which causes the problem obviously cannot be fixed, there is little hope that the interferences can be removed.

7. CODE SIMPLIFICATION

According to Parnas [1994], 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 or she might conclude that certain configurations

```
#if bad_unlink
        int
un_link(s)
        char const *s;
/*
* Remove S, even if it is unwritable.
* Ignore unlink() ENOENT failures; NFS generates bogus ones.
*/
{
#       if bad_unlink
                int e;
                if (unlink(s) == 0)
                        return 0;
                e = errno;
                if (chmod(s, S_IWUSR) != 0) {
                errno = e;
                return -1;
                }
#       endif
                return unlink(s);
}
#endif
#if !has_rename
#       define do_link(s,t) link(s,t)
 ...
```

Fig. 18. Partial evaluation of "rcsedit.c" under context expression !defined(has_NFS).

(i.e., certain preprocessor settings) need no longer be supported. Hence the corresponding code is irrelevant and should be discarded (this is called "amputation" by Parnas). 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.

In this section, we describe how amputation can be implemented via partial evaluation of CPP files. The process is driven by the lattice, which provides excellent insight into the possibilities and effects of an amputation.

7.1 Partial Evaluation of CPP Files Using NORA/ICE

Simplification 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 [Zeller 1995; Zeller and Snelting 1995]. NORA/ICE offers—among other features—partial evaluation of preprocessor files. It allows one 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 behavior is included as the "limit case," namely, that *all* preprocessor symbols have a known value. NORA/ICE allows for arbitrary complex context expressions, including those which introduce new symbols. Simplification is not just constant folding, but is based on *feature unification* [Smolka 1992].

Partial evaluation will considerably reduce the size of the source file, if several preprocessor symbols are known to be always defined or undefined. Figure 18 gives an example of partial evaluation: the source code of

"rcsedit.c" (line 180ff; see Figure 6) is simplified under the assumption that "has_NFS" is always undefined. The code piece shrinks from 35 lines to 24 lines (the inner #if becomes redundant; thus it could shrink to 22 lines, but NORA/ICE does not detect this).

7.2 The Art of Amputation

Generation of Source Code from Sublattices. Once a sublattice has been determined—either by horizontal decomposition or by explicit selection—a simplified source file corresponding to this sublattice can be created. This works with every sublattice, but the restructurer 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.

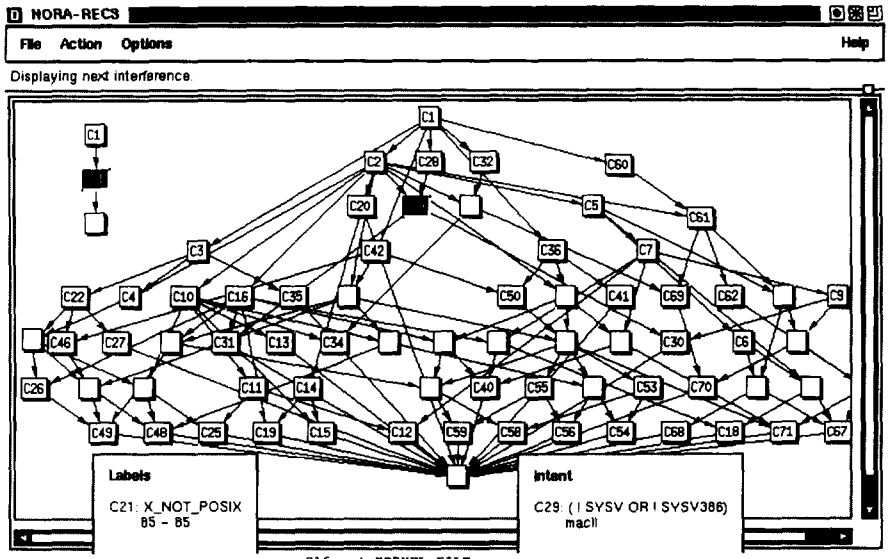
Source file simplification is done by partial evaluation of CPP files. Let $C = \{c_1, c_2, \ldots, c_k\}$ be the concepts *not* in the sublattice and $X = \{x_1, \ldots, x_n\} = \cup_{i=1}^{k} \alpha(c_i)$ all their attribute labels. Then the source text is fed to NORA/ICE, together with the context expression $[\neg defined(x_1), \ldots, \neg defined(x_n)]$. This removes all code pieces not in the configuration subspace and simplifies the governing expressions for the remaining code pieces.[11] 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 $C = \{c_1, \ldots, c_k\}$ 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 $X = \cup_{i=1}^{k} \alpha(c^i)$ 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 $[defined(a) \mid a \in W \setminus X] \sqcap [\neg defined(y) \mid y \in A \setminus (W \cup X)]$. This creates a source text which contains exactly the configurations containing the problematic code pieces. Such a specialized source file is useful for an analysis of interferences on the source code level.

7.3 Example: Simplifying "xload.c"

Let us now apply amputation to "xload.c." Figure 16 shows that there are several 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, and sgi.[12] Feeding the source code with context expression "[!defined(apollo) && !defined(X_NOT _POSIX) && . . . && !defined(sgi)]" to NORA/ICE, a reduced source file with 540 source lines results. The corresponding concept lattice has 75 concepts, which is still a lot. We therefore decide that we consider only configurations where KVM_ROUTINES are not available (whatever they may do). This results in a further reduction of the source file; it is now 490 lines long, and

[11] Actually, it would be enough to undefine only the maximal elements of C.
[12] This is not to be understood as a recommendation to the authors of "x.load.c!"




```
C1  : 1 - 35
      39 - 39
      43 - 43
      47 - 48
      57 - 58
      69 - 70
      74 - 77
      83 - 83
      87 - 91
      482 - 490
C2  : ! SYSV OR ! SYSV386
      179 - 179
      183 - 183
      224 - 225
      254 - 258
      271 - 271
      288 - 293
      341 - 350
      356 - 356
      479 - 480
C3  : SVR4 OR sun
C4  : 358 - 362
C5  : ! STDC__ OR ! SVR4
C6  : ! MOTOROLA OR ! SYSV
C7  : ! SVR4
C8  :
C9  : ! sun
C10 : ! KERNEL_LOAD_VARIABLE
      227 - 228
      234 - 236
      240 - 240
      252 - 252
C11 : ! USG
C12 : 267 - 267
C13 : USG
      261 - 262
C14 : hpux
C15 : hp9000s800
      231 - 231
C16 : ! KERNEL_FILE
      185 - 187
      191 - 191
      195 - 198
      210 - 213
C17 :
C18 : ! SYSV
      220 - 220
C19 : 189 - 189
C20 : ! KMEM_FILE
      181 - 181
C21 : X_NOT_POSIX
      85 - 85
C22 : SVR4
C23 :
C24 :
C25 : 265 - 265
C26 : 215 - 215
C27 : ! FSCALE
      80 - 80
C28 : macII
      60 - 67
C29 : 273 - 280
      295 - 314
      352 - 352
C30 : 365 - 368
C31 : 193 - 193
C32 : AIXV3
      50 - 55
C33 :
C34 : 238 - 238
C35 : sun
      45 - 45
C36 : SYSV
C37 :
C38 :
C39 :
C40 : 218 - 218
C41 : SYSV386
      93 - 177
C42 : MOTOROLA
C43 :
C44 :
C45 :
C46 : 72 - 72
C47 :
C48 : 249 - 249
C49 : 207 - 207
C50 : 41 - 41
C51 :
C52 :
C53 : m88k
C54 : 246 - 246
C55 : m68k
C56 : 243 - 243
C57 :
C58 : 204 - 204
C59 : 201 - 201
C60 : ! macII
      37 - 37
C61 : 282 - 286
      319 - 319
      325 - 339
      354 - 354
C62 : ! AIXV3
      323 - 323
C63 :
C64 :
C65 :
C66 :
C67 : 474 - 474
C68 : ! MOTOROLA
      317 - 317
C69 : 321 - 321
C70 : 371 - 400
C71 : 403 - 472
C72 :
```

Fig. 19. Simplified xload.c with an interferene of connectivity 1.

the corresponding lattice has 72 concepts. The symmetric case that we assume "defined(KVM_ROUTINES)" leads to a small source file with 16 configuration concepts and is not further investigated.

The 72-concept lattice is then subject to horizontal decomposition and interference analysis (Figure 19). The lattice is horizontally decomposable (see the C21 chain), which shows that X_NOT_POSIX (label of C21) does

not interfere with other configurations. There are three top-level interferences of connectivity $k = 1$, namely, C29, C33, and C61. Removal of each of these would isolate C28, C32, respectively, C60 (macII, AIXV3, ! macII), but leave the rest of the lattice unchanged and is therefore not further investigated. Similarly, $k = 2$, respectively, $k = 3$, interferences do not produce a "clean" lattice decomposition either.

We therefore decide to base further amputation on sublattice selection. Obviously, many concepts are below C2, which is labeled !SYSV || !SYSV386. We therefore produce a simplified source file which handles all configurations except "System V"; this is based on the sublattice $\downarrow \{C2\}$. According to Section 7.3, NORA/RECS collects the concept labels of the maximal elements outside $\downarrow \{C2\}$; these are C3, C28, C32, C60, and C21 with labels SVR4 || sun, macII, AIXV3, !macII, and X_NOT_POSIX. NORA/RECS then feeds the simplified source file to NORA/ICE, together with context expression $|\neg defined(SV\ R4),\ \neg defined(sun),\ \neg defined(macII),\ \neg defined(AIXV3),\ defined(macII),\ \neg defined(X\_NOT\_POSIX)|$. 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).

8. SIMPLIFICATION BASED ON CONCEPT LATTICE ISOMORPHISMS

In this final section, we present an algorithm which simplifies governing expressions by utilizing an isomorphism theorem from concept analysis. In particular, disjunctions in governing expressions are eliminated. This is of practical value, as disjunctions are very difficult to understand.

Definition 8.1. A lattice element $c \in L$ is called *join-irreducible*, if for all $a, b \in L$, $a < c$, and $b < c$ it implies $a \vee b < c$. c is called *meet-irreducible*, if for all $a, b \in L$, $a > c$, and $b > c$ it implies $a \wedge b > c$. The join-irreducible elements of L are written $J(L)$, and the meet-irreducible elements are written $M(L)$.

The join-irreducible elements are those with only one outgoing downward edge, and the meet-irreducible elements are those with only one outgoing upward edge. It is not surprising that the lattice is generated by the irreducible elements alone. The following isomorphism theorem resembles a famous result by Birkhoff for distributive lattices.

THEOREM [WILLE 1982]. *Let* $\underline{L} = (B(O, A, P), \leq)$ *be a concept lattice. Let* $X \supseteq J(L)$ *and* $Y \supseteq M(L)$ *be supersets of the join- (respectively, meet-irreducible) elements of L. Then* $\underline{L} \cong \underline{B}(X, Y, \leq)$. *In particular,* $\underline{L} \cong \underline{B}(J(L), M(L), \leq)$ *and* $\underline{L} \cong \underline{B}(L, L, \leq)$.

The latter isomorphism (which was already mentioned in Section 2.1) means that a concept lattice reproduces itself. The first isomorphism means that a concept lattice is generated by its join- (respectively, meet-irreducible) elements alone, if these are taken as objects (respectively, attributes). $(J(L), M(L), \leq)$ is called the *reduced context*; usually it is considerably smaller than the original one. In general, any supersets of the irreducible

elements generate the lattice, if used as objects (respectively, attributes); the corresponding contexts are called partially reduced.

The isomorphism theorem can be used to generate a restructuring proposal as follows. According to the theorem, the concept lattice corresponding to the formal context $\underline{C} = (\gamma(O), M(L), \leq)$ is isomorphic to the original lattice L—the meet-irreducible concepts are enough to "span" the configuration space. Hence we can generate a new configuration table. This restructured table has the same rows as the original one (the O in $(\gamma(O), M(L), \leq)$. For each meet-irreducible concept, we choose one new preprocessor symbol (if a meet-irreducible concept already has an attribute label, this can be used instead of generating a new one); these are used as column labels. The configuration table is then created according to the rule: for $o \in O$, $a \in M(L)$

$$T[o, a] = \begin{cases} \text{true} & \gamma(o) \leq \mu(a) \\ \text{false} & \text{otherwise.} \end{cases}$$

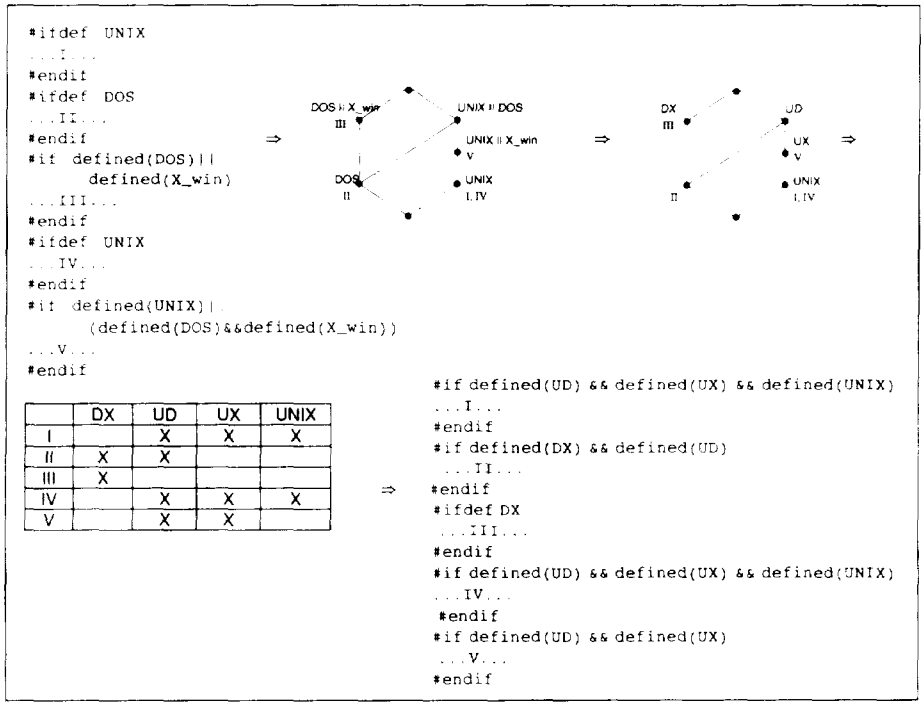
This table generates a concept lattice which is isomorphic to the old one. The new table is usually "leaner" than the original one, as it has fewer columns.

From the new table, the new governing expressions are generated as follows. Let $\sigma_T(o) = \{a_1, \ldots, a_n\}$ be the new CPP symbols governing o. Then the new governing expression for o is #if defined(a1) && ... defined(an). Hence the new governing expression contains only conjunctions, which is a tremendous simplification.

Example 8.1. In order to visualize the above notions, consider the lattice in Figure 4. The meet-irreducible elements are those labeled maxII, SYSV, sun, sony, ultrix, i386, sequent, apollo, or CRAY. AIX however is the infimum of sony and ultrix, hence not irreducible. This means that any code piece governed by AIX is also governed by sony and ultrix, indicating that AIX is redundant. Hence NORA/RECS proposes to remove all dependencies on AIX and replace them by defined(sony) && defined(ultrix). All configurations are kept intact: as sony and ultrix are probably disjoint,[13] they could not be defined simultaneously; hence defined(sony) && defined(ultrix) can safely be used to replace AIX.

Example 8.2. Consider the second example in Figure 11 (repeated in Figure 20). The meet-irreducible elements are those labeled DOS || X_win, UNIX || DOS, UNIX || X_win, and UNIX; these labels have been generated by normalizing complex governing expressions. This allows us to rename the $\wedge$-irreducible elements and reconstruct a simpler configuration table from the isomorphic lattice. We introduce three new preprocessor symbols, named DX, UD, and UX, which stand for DOS || X_win, UNIX || DOS, UNIX || X_win, respectively. According to the above theorem, the configuration table in Figure 20 produces an isomorphic concept lattice. From the table, we

[13]Remember that Figure 4 is a fictitious example.



	DX	UD	UX	UNIX
I		X	X	X
II	X	X		
III	X			
IV		X	X	X
V		X	X	

Fig. 20. Elimination of disjunctions from source code.

obtain the restructured source code, which is much easier to understand, as it no longer contains any disjunctions. A human restructurer would have a hard time transforming the governing expressions in a similar manner!

But note that the configuration consisting of code pieces I, III, IV, and V can no longer be selected. In the old source file, it could have been selected by defining both UNIX and DOS. As these are disjoint preprocessor symbols, the configuration need not be preserved. A computation of the configuration functions of both source files shows that all other configurations can still be selected in the simplified source code.

Thus by removing nonirreducible symbols, configurations can be lost. By computing the configuration functions NORA/RECS can track down all configurations which can no longer be selected and thus present them to the user. If the user insists on a particular configuration, he or she may again add attribute labels (CPP symbols) from the original lattice; the restructured table will be extended accordingly. The generated lattice still remains the same, as the theorem allows for arbitrary supersets of the meet-irreducible elements. In the example, adding DOS again as a "guard" to II allows us to select I, III, IV, and V; the source code is still considerably simpler.

In general, all governing expressions are transformed into minimal conjunctive normal form; furthermore, the lattice suggests the introduction

of new CPP symbols. But note that the structure of the lattice remains the same; hence interferences are not really resolved.

Example 8.3. Once more, we apply the method also to "xload.c." In the simplified lattice of Figure 19, the meet-irreducible concepts are C2, C3, C5, C6, C7, C9, C10, C11, C13, C14, C16, C20, C21, C22, C27, C28, C32, C35, C36, C41, C42, C53, C55, C60, C62, and C68. All of the other (meet-reducible) concepts except C15, C18, and C32 do not have attribute labels; hence the potential for saving CPP symbols is very low. Indeed, most of the governing expressions cannot be simplified.

This example shows once more that "xload" is a hard "nut to crack": its lattice cannot be decomposed, and its governing expressions cannot be simplified. Future work must show whether even more powerful methods (such as subdirect decomposition of the lattice) can reveal some structure or whether "xload" must be blamed for irrepairable configuration hacking.

9. CONCLUSION

We applied mathematical concept analysis in order to explore the configuration space of existing software. NORA/RECS not only displays all dependencies between configurations (respectively, their features), but also allows for simplifying governing expressions. Our preliminary experience can be summarized as follows:

(1) Keeping track of all possible variants of a software system is important and hard.
(2) Analyzing relationships among such variants is important as a prelude to restructuring the code to enhance maintainability.
(3) The set of possible variants of a software system can be characterized by the features of each variant.
(4) Concept lattices provide a theory of such variant spaces.
(5) Tools based on this theory are potentially useful for analyzing and minimizing variant spaces, because they can express interesting, relevant characteristics of various subsets of the variants, especially inclusion relationships among them.
(6) Applying these tools in several case studies has produced evidence that the analysis is useful.
(7) Efforts to use the theory as the basis for code restructuring has only begun.

Thus, NORA/RECS is a useful tool for analysis of configurations; as a tool for restructuring, it is still in its infancy. Future research must show whether modularization can be achieved automatically. We will investigate whether the theory of concept analysis can be utilized for restructuring; *subdirect decomposition* of concept lattices [Wille 1983] looks particularly promising.

As certain relations between "objects" and "attributes" occur all the time in software engineering, concept analysis is potentially useful in other

kinds of maintenance activities. 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. Note that in contrast to other reengineering approaches, concept analysis is deterministic and always allows one to reconstruct the raw data from the lattice. For example, the approach of Schwanke [1991] is based on a similarity function which contains several free parameters and therefore requires tuning. Concept analysis does not use any heuristics and is more transparent.

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.[14] 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 [Zeller 1995; Zeller and Snelting 1995];
—NORA/HAMMR (highly adaptive multimethod retrieval) offers software component retrieval based on deductive and lattice-theoretic techniques [Fischer et al. 1995a; 1995b; Lindig 1995];
—NORA/HOML (higher-order module language) is a calculus for designing reference architectures, which is based on $\lambda\nu$-calculus with dependent types [Grosch 1995].

NORA/RECS can be obtained through the WWW page of the software technology group in Brunswick: http://www.cs.tu-bs.de/softech/.

ACKNOWLEDGMENTS

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

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

REFERENCES

AHO, A., HOPCROFT, J., AND ULLMAN, J. 1983. *Data Structures and Algorithms,* 2nd ed. Addison-Wesley, Reading, Mass.

BIRKHOFF, G. 1940. *Lattice Theory,* 1st ed. American Mathematical Society, Providence, R.I.

DAVEY, B. A. AND PRIESTLEY, H. A. 1990. *Introduction to Lattices and Order,* 2nd ed. Cambridge University Press, Cambridge, England.

DUQUENNE, V. 1987. Contextual implications between attributes and some representation properties for finite lattices. In *Beiträge zur Begriffsanalyse,* B. Ganter, R. Willie, and K. Wolff, Eds. B. I. Wissenschaftsverlag, Berlin, 213–240.

ESTUBLIER, J. AND CASALLAS, R. 1994. The Adele configuration manager. In *Trends in Software.* Vol. 2, *Configuration Management,* W. F. Tichy, Ed. John Wiley and Sons, Chichester, England, 99–133.

FISCHER, B., KIEVERNAGEL, M., AND SNELTING, G. 1995a. Deduction-based software component retrieval. In *Working Notes of the IJCAI-95 Workshop on Formal Approaches to Reuse of Proofs, Plans, and Programs* (Montreal, Canada).

FISCHER, B., KIEVERMAGEL, M., AND STRUCKMANN, W. 1995b. VCR: A VDM-based software component retrieval tool. In *Proceedings of the ICSE-17 Workshop on Formal Approaches to Software Engineering* (Seattle, Wash.).

GANTER, B. 1987. Algorithmen zur formalen Begriffsanalyse. In *Beiträge zur Begriffsanalyse,* B. Ganter, R. Willie, and K. Wolff, Eds. B.I. Wissenschaftsverlag, Berlin, 241–254.

GANTER, B. 1995. Formal concept analysis—A subjective introduction. Mathematik-Bericht, Technische Universitaet Dresden, Germany.

GANTER, B., WILLE, R., AND WOLFF, K., Eds. 1987. *Beiträge zur Begriffsanalyse.* B.I. Wissenschaftsverlag, Berlin.

GROSCH, F.-J. 1995. No type stamps and no structure stamps—A fully applicative higher-order module language. Informatik-Bericht, TU Braunschweig. May. Submitted for publication.

KRONE, M. 1993. Reverse engineering von konfigurationsstrukturen. Master's thesis, Technical Univ. of Braunschweig, Germany. Sept. In German.

KRONE, M. AND SNELTING, G. 1994. On the inference of configuration structures from source code. In *ICSE-16 Proceedings of the 16th International Conference on Software Engineering.* IEEE Computer Society Press, Los Alamitos, Calif., 49–57.

LEBLANG, D. B. 1994. The CM challenge: Configuration management that works. In *Trends in Software.* Vol. 2, *Configuration Management,* W. F. Tichy, Ed. John Wiley and Sons, Chichester, England, 1–37.

LINDIG, C. 1995. Concept-based component retrieval. In *Working Notes of the IJCAI-95 Workshop: Formal Approaches to the Reuse of Plans, Proofs, and Programs,* J. Kohler, F. Giunchiglia, C. Green, and C. Walther, Eds. 21–25.

MAHLER, A. 1994. Variants: Keeping things together and telling them apart. In *Trends in Software.* Vol. 2, *Configuration Management,* W. F. Tichy, Ed. John Wiley and Sons, Chichester, England, 39–69.

PARNAS, D. 1994. Software aging. In *ICSE-16 Proceedings of the 16th International Conference on Software Engineering.* IEEE Computer Society Press, Los Alamitos, Calif., 279–290.

SCHWANKE, R. W. 1991. An intelligent tool for reengineering software modularity. In *Proceedings of the 13th International Conference on Software Engineering.* IEEE Computer Society Press, Los Alamitos, Calif., 83–92.

SKORSKY, M. 1992. Endliche Verbäude—Diagramme und Eigenschaften. Ph.D. thesis, Dept. of Mathematics, Technical Univ. of Darmstadt, Germany.

SMOLKA, G. 1992. Feature-constrained logics for unification grammars. *J. Logic Program. 12,* 51–87.

SUGIYAMA, K., SHORJIRO, T., AND TODA, M. 1981. Methods for visual understanding of hierarchical system structures. *IEEE Trans. Syst. Man Cybernet. 11,* 2, 109–125.

TICHY, W. 1985. RCS—A system for version control. *Softw. Pract. Exper. 15,* 7 (July), 637–654.

TICHY, W. 1988. Tools for software configuration management. In *Proceedings of the 1st International Workshop on Software Configuration Management*, J. Winkler, Ed. Teubner Verlag, Grassau, 1–20.

TICHY, W. F., Ed. 1994. *Trends in Software*. Vol. 2, *Configuration Management*. John Wiley and Sons, Chichester, England.

VAN MECHELEN, I., HAMPTON, J., MICHALSKI, R. S., AND THEUNS, P., Eds. 1993. *Categories and Concepts*. Academic Press, London.

WILLE, R. 1982. Restructuring lattice theory: An approach based on hierarchies of concepts. In *Ordered Sets*, I. Rival, Ed. Reidel, 445–470.

WILLE, R. 1983. Subdirect decomposition of concept lattices. *Algebra Universalis 17*, 275–287.

WILLE, R. 1989a. Geometric representation of concept lattices. In *Conceptual and Numerical Analysis of Data*, O. Opitz, Ed. Springer-Verlag, Berlin, 239–255.

WILLE, R. 1989b. Lattices in data analysis: How to draw them with a computer. In *Classification and Related Methods of Data Analysis*, H. H. Bock, Ed. North-Holland, Amsterdam, 33–58.

WILLE, R. AND GANTER, B. 1993. Mathematische theorie der formalen begriffsanalyse. Lecture notes, Dept. of Mathematics, Technical Univ. of Darmstadt, Germany.

ZELLER, A. 1995. A unified version model for configuration management. In *Proceedings of the SIGSOFT 3rd Symposium on the Foundations of Software Engineering*. ACM, New York, 151–160.

ZELLER, A. AND SNELTING, G. 1995. Handling version sets through feature logic. In *Proceedings of the 5th European Software Engineering Conference*, W. Schafer, Ed. Lecture Notes in Computer Science, vol. 989. Springer-Verlag, Berlin, 191–204.

Received January 1995; revised August 1995; accepted February 1996