Algorithms for Concept Lattice Decomposition and their Application

P. Funk, A. Lewien, G. Snelting
TU Braunschweig
Abteilung Softwaretechnologie

Abstract

We present algorithms for horizontal decomposition, subdirect decomposition, and subtensorial decomposition of concept lattices. The implementations of these algorithms are described, and their complexity is investigated. We then apply the decomposition algorithms to reengineering problems in software engineering, and present several examples. It turns out that concept lattice decomposition is useful not only for understanding old software, but also for restructuring it.

1 Introduction

Analysing old software has become an important topic in software technology, as there are millions of lines of legacy code which lack proper documentation; due to ongoing modifications, software entropy has increased steadily. If nothing is done, such software will die of old age - and the knowledge embodied in the software is inevitably lost. As a first step in "software geriatry", one must understand the structure of old software and reconstruct abstract concepts from the source code (called "software reengineering"). In a second step, one might try to transform the source code such that the structure of the system is improved and obeys modern software engineering principles.

In earlier work, we have shown that formal concept analysis is a useful tool for analysing old software. As a particular reengineering problem, we have chosen the analysis of configurations in UNIX source files. We have shown how configuration spaces can be extracted from old source code, and how dependencies and interferences between configurations can be detected using a concept lattice [Kr93, KS94, LS95]. More recent work described how to automatically detect interferences, and how source files can be simplified according to lattice-generated information [Le96, Sn95]. Still, automatic restructuring of configurations is an open problem. Fortunately, the theory of concept lattices offers several promising approaches not only to analysis of old software, but also to automatic restructuring.

In this paper, we are concerned with decompositions of concept lattices, namely horizontal decompositions, subdirect decompositions, and subtensorial decompositions. The aim of this work is twofold. First, we are interested in decomposition in its own right. We will study algorithms for automatic lattice decomposition, and we will analyse the complexity of these algorithms. Second, we want to test the actual implementations of the algorithms and present several examples for the application of concept lattice decompositions in software reengineering. The paper assumes some familiarity with the mathematics of concept analysis.

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

	A	*B*	*C*	*D*
I	×			
II		×	×	
III		×		×
IV		×	×	×

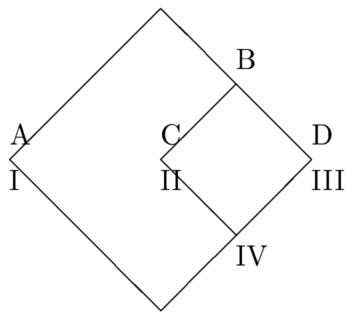


Figure 1: A small source text and its configuration lattice

1.1 Configuration reengineering based on formal concept analysis

Software configuration management is the discipline of controlling the evolution of software systems. A configuration is a set of software elements (usually code pieces, functions or modules) which meets the needs of a particular client or platform. Therefore, configuration management must be able to build a software system from selected components, where selection is done according to certain features (attributes) of the target configuration.

Many UNIX programs use the preprocessor CPP for configuration management. Configuration-specific code is enclosed in `#ifdef ... #endif` brackets. During compiler invocation, CPP variables are set which cause configuration-specific code to be selected and compiled. Since `#ifdef`s can be nested and can use arbitrary complex *governing expressions* to control selection of code pieces, such programs are often incomprehensible.

Formal concept analysis can be used to infer configuration structures from old source code. First, the source file is transformed into a *configuration table*, which summarizes dependencies of code pieces on CPP variables. From the table, a concept lattice is computed which not only displays intent and extent of configurations, but also all *dependencies* between configurations, e.g. "A code piece which is part of the sun configuration is part of the HP configuration as well". Such statements are not easy to obtain manually from complicated source files!

As an example, consider the small code fragment, its configuration table, and its concept lattice presented in figure 1. In general, governing expressions may contain boolean operations, thus construction of the configuration table is not trivial (see [KS94]).

The configuration lattice can be computed, displayed and analysed by the tool NORA/RECS. Figure 2 presents a real-world example: the configuration structure of the RCS stream editor. This 1656-line UNIX program uses 21 CPP variables for configuration management. The lattice is quite flat: there is little interdependence between CPP variables, which is good from a software engineering viewpoint. However in the right part there are some suspicious infima, which show that there is interference between some configurations: they have common code, where they should not. Indeed, one of the infima revealed a bug in the code!

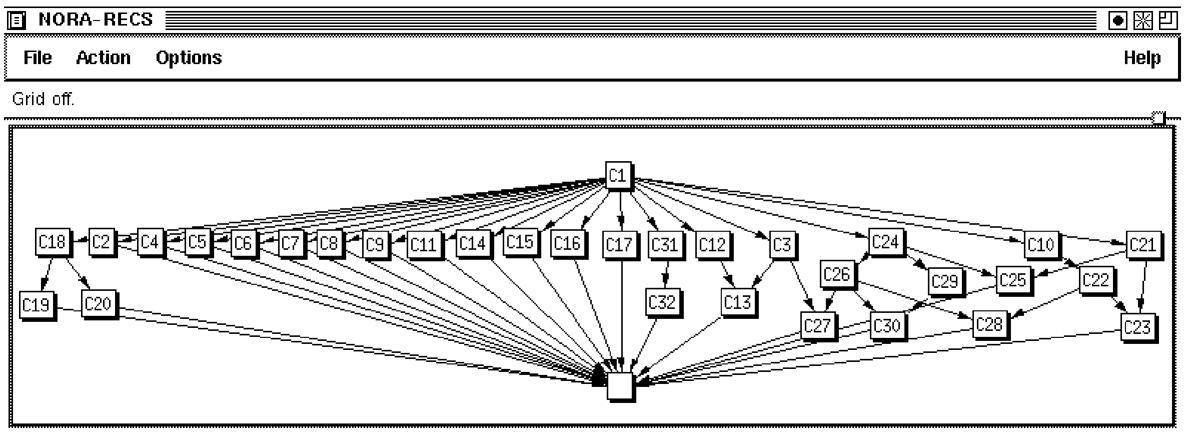


Figure 2: Configuration structure of the RCS stream editor

Formally, a configuration table is a formal context $C = (O, A, R)$ where O is a set of code pieces (the objects), A is a set of CPP variables (the attributes), and $R \subseteq O \times A$. For a formal context $C = (O, A, R)$, the corresponding concept lattice is denoted $\mathcal{B}(C)$. The lattice elements (formal concepts) are pairs written $I = (X, Y)$ where $X \subseteq O$ is the extent, and $Y \subseteq A$ is the intent of the concept: $X = ext(I)$, $Y = int(I)$. For $o \in O$, the smallest concept I where $o \in ext(I)$ is written $\gamma(o) = \bigwedge_{o \in ext(c)} c$, and for $a \in A$, the largest concept where $a \in int(I)$ is $\mu(a) = \bigvee_{a \in int(c)} c$. γ and μ are used to label the concepts with objects and attributes. The attribute labels of a concept $\alpha(c)$ are given by $a \in \alpha(\mu(a))$; the object labels $\omega(c)$ are given by $o \in \omega(\gamma(o))$.

The CPP behaviour is abstractly described by a *configuration function*: for a configuration table C, the configuration function $\mathcal{K}_C : 2^A \to 2^O$ is given by $\mathcal{K}_C(X) = \{o \in O \mid o' \subseteq X\}$, where – as usual – the "derivative" $o' = \{a \in A \mid (o, a) \in R\}$. For any $X \subseteq A$, $\mathcal{K}_C(X)$ is the set of code pieces (the configuration) selected by X.

2 Subdirect Decompositions

Subdirect products and subdirect decompositions are standard algebraic constructions, and algorithms for subdirect decomposition are interesting in their own right. Subdirect factorization seems also promising for restructuring, as it might be a basis for automatic modularization. If a concept lattice is subdirectly decomposable, this means that its concepts are in fact combinations of simpler concepts – perhaps this information can be used for code restructuring.

2.1 Basic definitions and properties

Let $\underline{A}, \underline{A_1}, \underline{A_2}$ be algebras. $\underline{A}$ is a subdirect product of $\underline{A_1}$ and $\underline{A_2}$ iff $\underline{A}$ is (isomorphic to) a subalgebra of the direct product of $\underline{A_1}$ and $\underline{A_2}$: $\underline{A} \subseteq \underline{A_1} \times \underline{A_2}$; where both projections $\pi_1 : A \to A_1$; $\pi_1(x, y) = x$ and $\pi_2 : A \to A_2$; $\pi_2(x, y) = y$ are surjective.

The projections $\pi_{1,2}$ are in fact homomorphisms. They are required to be surjective, as otherwise not all of A_1 or A_2 would be needed in order to generate A. If $\underline{A} \subseteq \underline{A_1} \times \underline{A_2}$, we say there is a subdirect decomposition of A into $A_{1,2}$. In case there are only trivial decompositions

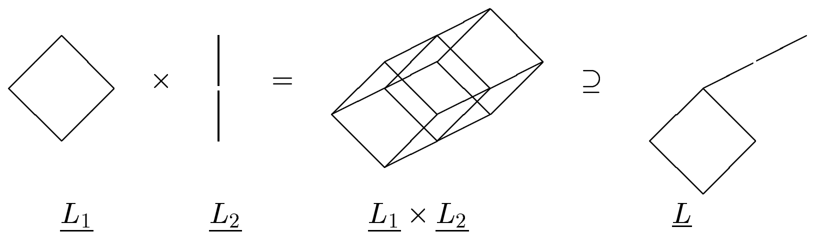


Figure 3: A subdirect product of two lattices

(i.e. either π_1 or π_2 is an isomorphism), A is said to be subdirectly irreducible.

As an example, consider the lattices L_1 and L_2 and their direct product $L_1 \times L_2$ (figure 3). Then the lattice L in the right part of the picture is a sublattice of $L_1 \times L_2$, and both $L_{1,2}$ are homomorphic images of L via $\pi_{1,2}$. Hence L can be subdirectly decomposed into $L_{1,2}$.

Subdirect products can be generalized to more than two factors. The importance of subdirect products compared to direct products is due to Birkhoff's famous

Theorem. Every algebra is the subdirect product of subdirectly irreducible algebras.

As the projections from a subdirect product to its factors are surjective homomorphisms, they induce congruences θ_1 and θ_2: $\theta_{1,2} = \{(x, y) \mid \pi_{1,2}(x) = \pi_{1,2}(y)\}$. For $x \in A$, the congruence classes of x are written $[x]_{\theta_1}$ resp. $[x]_{\theta_2}$, and the factor algebras are written $\underline{A}/\theta_1$ resp. $\underline{A}/\theta_2$. θ_1 and θ_2 have a characteristic separation property:

Definition. Congruences θ_1 and θ_2 on $\underline{A}$ have the separation property, iff $\theta_1 \cap \theta_2 = \Delta_A$.[1]

θ_1 and θ_2 have the separation property iff all intersections of respective congruence classes have at most one element: $|[x]_{\theta_1} \cap [y]_{\theta_2}| \leq 1$. Hence a pair of θ_1 resp. θ_2 congruence classes which are not disjoint exactly identifies an element in A: $[x]_{\theta_1} \cap [x]_{\theta_2} = \{x\}$. It is well known that $\underline{A}$ is a subdirect product iff the corresponding congruences are separating:

Theorem.

$$\underline{A} \subseteq \underline{A_1} \times \underline{A_2} \Longleftrightarrow \theta_1 \cap \theta_2 = \Delta_A$$

For a pair $\theta_{1,2}$ of separating congruences, $\underline{A}$ is isomorphic to a subalgebra of $\underline{A}/\theta_1 \times \underline{A}/\theta_2$. The embedding of $\underline{A}$ into $\underline{A}/\theta_1 \times \underline{A}/\theta_2$ is given by

$$\psi: \ \underline{A} \to \underline{A}/\theta_1 \times \underline{A}/\theta_2, \quad \psi(x) = ([x]_{\theta_1}, [x]_{\theta_2})$$

2.2 An algorithm for subdirect decomposition

The theorem opens a simple way to the computation of subdirect factors of a concept lattice $L = \mathcal{B}((O, A, R))$.

1. Determine all congruences of L (see below).

[1] $\Delta_A = \{(x, x) \mid x \in A\}$ is the trivial congruence which has singleton congruence classes.

2. For all pairs of congruences θ_1 and θ_2, check whether they have the separation property:[2] test whether

$$\forall [x] \in \theta_1 \; \forall [y] \in \theta_2 : \; |[x] \cap [y]| \leq 1$$

The congruences are determined by the standard method: for L's context $C = (O, A, R)$ the arrow relations $\nearrow$ and $\swarrow \subseteq O \times A$ are computed (see [WG93]). Every congruence then corresponds to an arrow-closed subset $S \subseteq O \cup A$. All these subsets are computed by a standard algorithm on the arrow graph: for every $x \in O \cup A$, its $\swarrow/\nearrow$-closure is computed by depth-first search. As unions of arrow-closed subsets are arrow-closed as well, finally the $\cup$-closure of the arrow-closed subsets must be determined (see section 4.2 for the prototypical implementation of a closure operator).

For an arrow-closed subcontext $D = (O \cap S, A \cap S, R_{|O \cap S \times A \cap S})$ of C, the corresponding congruence θ_D is determined through $\mathcal{B}(C)/\theta_D = \mathcal{B}(D)$; furthermore, $\mathcal{B}(D) = \phi(\mathcal{B}(C))$ where $\phi : \mathcal{B}(C) \to \mathcal{B}(D)$ merges concepts by removing objects and attributes not in S: $\phi((X, Y)) = (X \cap S, Y \cap S)$. This property is used for the actual computation of subdirect factors $\mathcal{B}(C)/\theta_{1,2}$.

The arrow relations can be computed in worst-case time $\mathcal{O}(|O| \cdot |A| \cdot |L|)$, the arrow-closed subsets can be determined in time $\mathcal{O}((|O| + |A|)^2)$, and their $\cup$-closure in worst-case time $\mathcal{O}((|O| + |A|)^3)$. If there are n congruences, the subsequent computation of all subdirect factorizations will take time $\mathcal{O}(n^2 \cdot |L|^3)$ (note that a congruence has at most $|L|$ classes, and that determining $[x]$ via ϕ can have worst-case complexity $\mathcal{O}(|L|)$ as well; thus the separation property can be tested in $\mathcal{O}(|L|^3)$).

The overall complexity poses no problems even for large lattices on today's workstations; furthermore, some improvements are easy to apply.

2.3 Application examples

NORA/RECS offers to compute all congruences; congruence classes can be displayed in the original lattice using color[3]. The factor lattice can also be displayed by merging congruence classes into one element. Furthermore, NORA/RECS offers to compute all subdirect decompositions, without displaying the congruences first [Fu96].

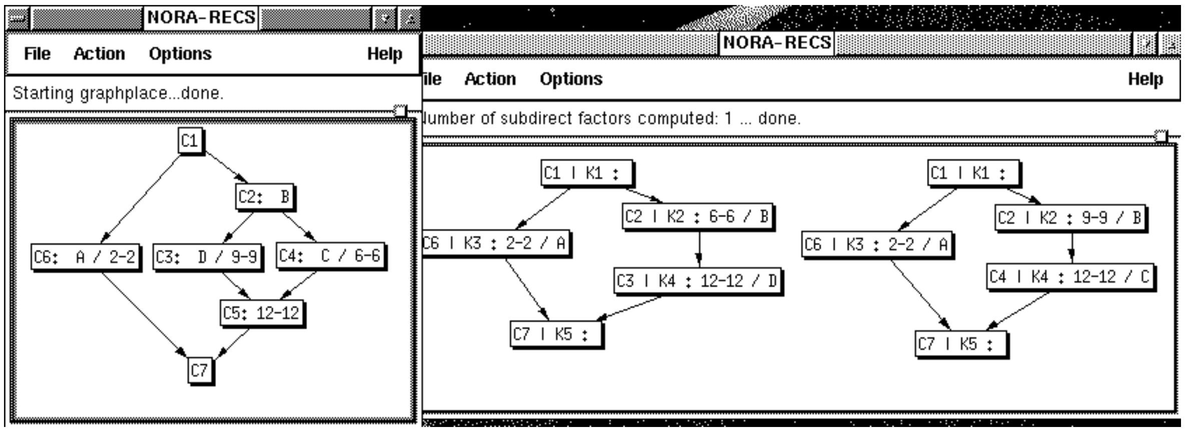
As a first example, consider the lattice of section 1.1 (repeated in the left part of figure 4)[4]. This lattice has indeed a subdirect decomposition, the factors are presented in the right part of figure 4. Among others, there are two congruences $\theta_{1,2}$, where θ_1 merges C3/C5 and C2/C4, θ_2 merges C2/C3 and C4/C5. In the two factors $L/\theta_{1,2}$, congruence classes are numbered (K1 etc) and are represented by elements of the original lattice (C1 etc). The labelling in the factor lattices correspond to the arrow-closed subcontexts which generate the factors.

$\theta_{1,2}$ are indeed separating. For example, C5 in the original lattice is the one and only element in the intersection of congruence class K4 of θ_1 and K4 of θ_2. Informally, the small square in the original lattice is the direct product of the chains C2/C3 resp. C2/C4 in the factors, whereas the rest of the factor lattices is copied but not duplicated. Although there are more congruences (e.g. $\theta_1 \cup \theta_2$), there are no more subdirect decompositions, as there are no more separating congruence pairs.

[2]Usually, one is interested in subdirect factors as small as possible; therefore it is reasonable to check pairs of small factors first.

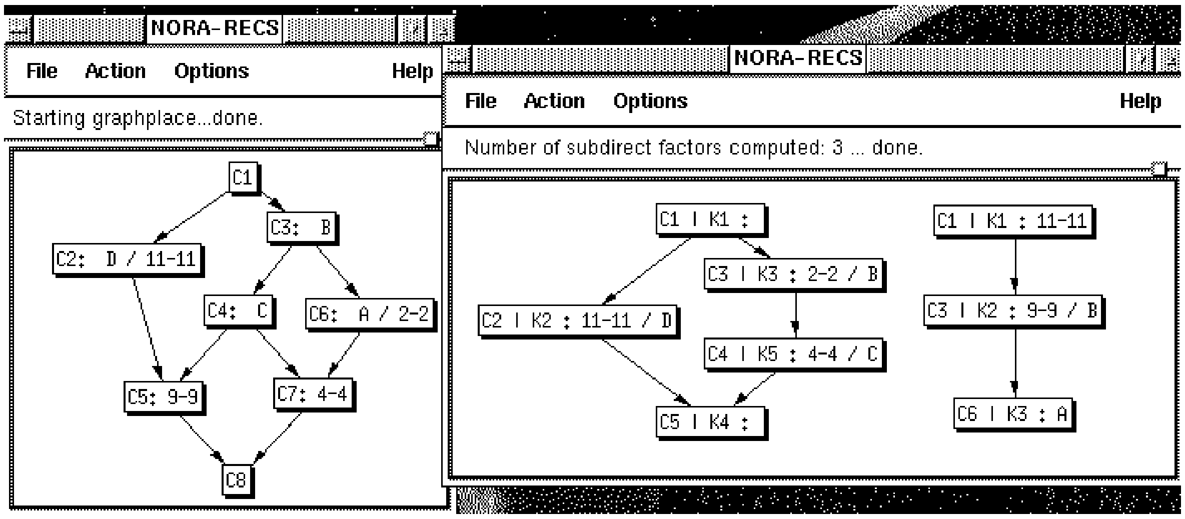
[3]At the moment, congruence classes are numbered.

[4]NORA/RECS uses line number intervalls for identification of code pieces.


Starting graphplace...done.

Number of subdirect factors computed: 1 ... done.

Figure 4: Subdirect decomposition of example lattice from figure 1


Starting graphplace...done.

Number of subdirect factors computed: 3 ... done.

Figure 5: Another subdirect decomposition

Our second example is very similar. The lattice in the left part of figure 5 has several subdirect decompositions; the decomposition presented in figure 5 has the smallest factor lattices and is therefore the "best". The first congruence merges C3/C6, C4/C7, and C5/C8; the second merges C1/C2, C3/C4/C5, and C6/C7/C8. The reader may easily verify that the congruences are separating. The example will return in chapter 4, where it will turn out that – to the surprise of the authors – the subdirect factors are subtensorial factors as well.

We applied subdirect decomposition also to several lattices obtained from real UNIX programs. Unfortunately, these lattices are usually subdirectly irreducible. The reason is that these lattices do not have congruences, as they are quite flat[5]. Some lattices had subdirect decompositions, but the factors were almost as large as the original lattices – hence unusable for automatic restructuring. For example, "rcsedit" (figure 2) has a subdirect decomposition, as merging C31/C32 and C10/C22 results in two separating congruences. But the factor

[5]Remember that lattice congruence classes must be intervals, that is, all elements between an interval top and a bottom element.

lattices have 32 elements each, whereas the original latice has 33 elements. Another program crying most loudly for restructuring had a lattice of 143 elements, but just one congruence!

It should be mentioned that full lattice congruences are maybe too strong a requirement for restructuring. Congruences on the supremum-semilattice should be much more common, and already provide a reasonable partitioning of the attribute space, which might guide manual restructuring.

3 Horizontal Decompositions

One of the most valuable properties of the configuration lattice is its ability to visualize code pieces which depend on two or more CPP symbols. Such code pieces must be infima with non-empty extent. For example, in figure 2, C27 indicates that C3 and C26 have common code. Such common code may be problematic from a software engineering viewpoint. In particular, infima between sublattices indicate so-called interference if the CPP symbols in the sublattices deal with independent configuration aspects. It is therefore of practical importance to determine all interferences automatically; this is achieved via horizontal decompositions. A discussion of interferences from a software engineering viewpoint is presented in [Sn95].

3.1 Basic definitions

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

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

where $\top \geq x$, $\bot \leq x$ for all $x \in \sum_{i=1}^{n} L_i$. The L_i are assumed to be disjoint and are called *summands*. Horizontal sums can be generalized to bounded partial orders.

Conversely, a lattice L is horizontally decomposable, if it is a horizontal sum. But in practice, L might not just have a top element, but a *top chain* $t^1 < t^2 < \ldots < \top$ (and instead of a bottom element, it might have a bottom chain $t_1 > t_2 > \ldots > \bot$). For purposes of interference analysis, such top or bottom chains are irrelevant, as – in our application – they just reveal a top-level nesting of `#ifdef`s. We therefore generalize horizontal decomposability by applying it to a factor order L/θ, where the congruence θ merges the top/bottom chains into one element: $t^1\theta t^2\theta \ldots \theta\top$, and $t_1\theta t_2\theta \ldots \theta\bot$ ($[x]_\theta = \{x\}$ otherwise).

Definition. A lattice L is called *horizontally decomposable* if it has a top or bottom chain with corresponding congruence θ, and the factor lattice is a horizontal sum: $L/\theta = \sum_{i=1}^{n} L_i$. Any $L_i \setminus \{\top, \bot\} \cup \{[\top]_\theta, [\bot]_\theta\}$ is called a *summand* of L.

In case a lattice (or bounded partial order) is not horizontally decomposable, it might be that it is decomposable after a small number of interferences have been removed.

Definition. Two attributes a and b *interfere* if $ext\,(\mu\,(a) \wedge \mu\,(b)) \neq \emptyset$. Two sets A, B of attributes interfere, if there exist interfering $a \in A, b \in B$.

In our application, interference means that configurations have common code. Interference is a syntactic phenomenon and can be detected automatically (see below). It is much more difficult to decide whether an interference is harmful, as this depends on the semantics of the involved attributes (CPP symbols, in our application). Only a human with good knowledge

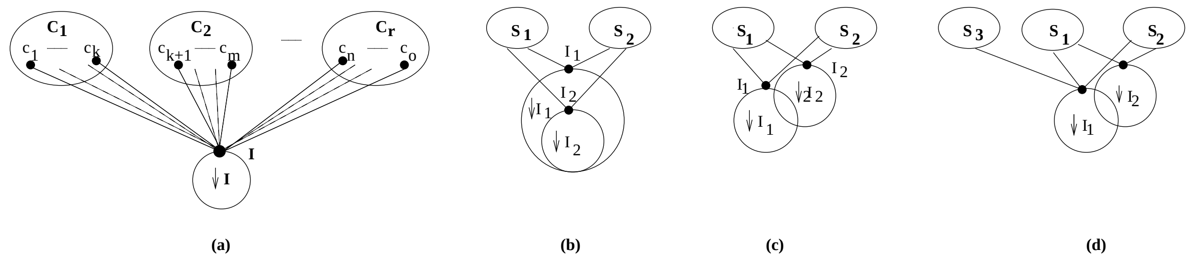


Figure 6: Simple and complex interferences

of software engineering principles can decide whether code common to two configurations should be considered beneficial reuse, or violation of software design principles.

For an algorithmic analysis of interferences, it is reasonable to investigate interferences between big sublattices or horizontal summands first, as – in our application – these are more likely to reveal bad system structure. This leads to the following definition.

Definition.[6] Let $L_1, \ldots, L_r$ be sublattices of L. Let I be $\wedge$-reducible: $I = \bigwedge_{i=1}^{r}(c_i)$, ($c_i$ direct predecessor of I), where $ext(I) \neq \emptyset$. *I is called simple interference* between $L_1, \ldots, L_r$, if $c_i \in L_i$, and I connects horizontal summands: there are sublattices $L_{r+1}, \ldots, L_n$ such that

$$L \backslash \downarrow I \cup \{\bot\} = \sum_{i=1}^{n} (L_i \backslash \downarrow I \cup \{\bot_i\})$$

r is called the *valence* of the interference. Figure 6a and b both present simple interferences. In the latter example, there is another infimum $I_2 \leq I_1$ between S_1 and S_2, but removal of $\downarrow I_2$ does not make $S_{1,2}$ horizontal summands. Thus interferences must be maximal infima between sublattices. From an application viewpoint, the two interferences $I_{1,2}$ in figure 6b cannot be considered worse than the simple interference I_1, as $I_2 \leq I_1$ is equivalent to $ext(I_2) \subseteq ext(I_1)$, and we already know that the configuration subspaces S_1 and S_2 interfere in the configuration code established by $ext(I_2)$.

In figure 6c however, we have two overlapping interferences I_1 and I_2 of valence 2. Here, $\downarrow I_1 \cap \downarrow I_2 \neq \emptyset$, but neither $I_1 \leq I_2$ nor $I_2 \leq I_1$. Hence I_1 and I_2 are not simple interferences, but together they constitute a complex interference. In general it might be that k infima must be removed in order to make the lattice decomposable. This leads to the following

Definition. An *interference of connectivity k* between $L_1, \ldots, L_r$ consists of k mutually incomparable infima $I_1, \ldots, I_k$, if there are $L_{r+1}, \ldots, L_n$ such that

$$L \backslash \downarrow \{I_1, \ldots, I_k\} \cup \{\bot\} = \sum_{i=1}^{n} (L_i \backslash \downarrow \{I_1, \ldots, I_k\} \cup \{\bot_i\})$$

and no subset of $\{I_1, \ldots, I_k\}$ is an interference of connectivity $k-1$ between $L_1, \ldots, L_r$.

Thus simple interferences are interferences of connectivity 1. Note that k simple interferences are not one interference of connectivity k – the definition requires that only simultaneous removal of the interferences decomposes the lattice. Figure 6c shows an interference of connectivity 2 between S_1 and S_2. Figure 6d displays an interference of connectivity 2 between S_1, S_2 and S_3. The latter example can also be considered an interference between S_1 and S_2 alone. However, figure 6b shows an interference of connectivity 1, as $I_1 \geq I_2$.

[6]For $X \subseteq L$, $\downarrow X = \{x \in L \mid \exists y \in X : x \leq y\}$, and $\uparrow X = \{x \in L \mid \exists y \in X : x \geq y\}$.

Interference as defined above implies interfering attributes. But the converse is not true, as there might be interfering attributes which do not show up as interferences – they are hidden in sublattices and will only be detected if such sublattices are investigated in isolation. Thus the above interference definitions are biased towards top-level interferences.

3.2 An algorithm for horizontal decomposition

The algorithm for detecting interferences of minimal connectivity implements the definitions from the previous section. It proceeds as follows:

1. Try a horizontal decomposition of the lattice. In case $ext([\bot]_\theta) \neq \emptyset$, the top element of the bottom chain represents an interference, thus decomposition fails. Otherwise, remove the top and bottom chains and determine the connected components of the (undirected) lattice graph by depth-first search. If successful, there are no top level interferences (connectivity $= 0$). Reattach $[\top]_\theta$ and $[\bot]_\theta$ (with unchanged labelling) to each connected component, and apply the remaining steps recursively to the sublattices.

2. Simple interferences in L can be detected by computing the $\wedge$-reducible *articulation points* of $L \setminus ([\top]_\theta \cup [\bot]_\theta)$. This is done by an extension of the standard algorithm for *biconnected components*, which itself is a simple extention of the depth-first search. Unfortunately not all the simple interferences can be detected that way. As explained above, two infima I_1, I_2 between sublattices where $I_2 \leq I_1$ are considered of connectivity 1 but cannot be detected through biconnected components (figure 6b). Such interferences are found together with interferences of higher connectivity.

3. For computing interferences of higher connectivity first determine the set of potential interference candidates $C = \{(I, \{(a_1, \ldots, a_r)\}) \mid I$ is $\wedge-$reducible and $a_1, \ldots a_r \in$ direct predecessors$(I)\}$ where $\uparrow a_i \cap \uparrow a_j = \emptyset\}$.

   Now determine the interferences of higher connectivity[7]

```
FOR ALL  $k$-subsets $S = \{c_1, \ldots, c_k\}$ of $C$ DO
LET  any $c_i = (I_i, \{a_i^1, \ldots, a_i^{r_i}\})$
  IF  $I_1, \ldots, I_k$ are mutually incomparable THEN
    FOR $i := 1$ TO $k$ DO  remove $\downarrow I_i$ from the lattice graph;
    choose a candidate $c_i$ from $S$;
    IF $\exists a_i^m, \ldots, a_i^{m+n-1} \subseteq \{a_i^1, \ldots, a_i^{r_i}\}$  where the $a_i^\nu$ are mutually unconnected
        AND $\forall c_j \in S \setminus c_i \; \exists a_j^m, \ldots, a_j^{m+n-1} \subseteq \{a_j^1, \ldots, a_j^{r_j}\}$
        where $a_i^\nu \to a_j^\nu$ $(\nu = m \ldots m+n-1)$ THEN
        $I_1, \ldots, I_k$ is interference of connectivity $k$ and valence $n$!
```

Step 1 and 2 are both based on depth-first search and thus have time complexity $\mathcal{O}(|L|)$. All k-combinations of candidates can be determined by taking the standard algorithm for computing the strong isotonic words[8] of length k over the alphabet C with length of alphabet $= |L|$. The removal of the $k \downarrow I_i$ can be done in worst-case time $\mathcal{O}(|L|)$. The subset $\{a_1, \ldots a_n\}$ of a

[7] $a \to b$ stands for a is connected with b

[8] the strong isotonic words of length s correspond to the s-combinations without repetition.

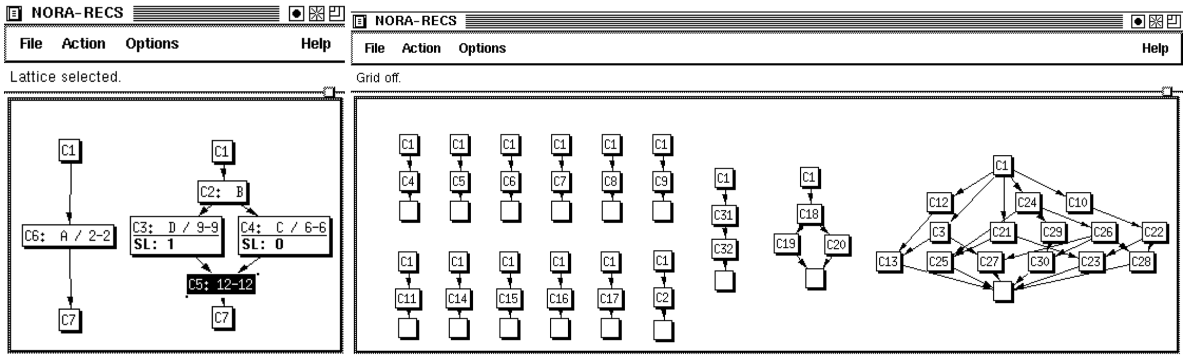
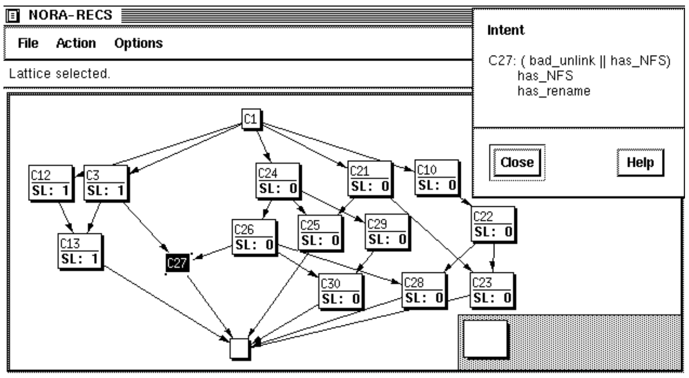


Figure 7: Horizontal decomposition of lattices from fig. 1 and fig. 2



```
C27: ( bad_unlink || has_NFS)
     has_NFS
     has_rename
```

Figure 8: Interference analysis in a horizontal summand

given candidate which defines a interference of connectivity k, valence n can be determined in $\mathcal{O}(|L|^3)$. This results from iteration over the $(I_j, \{a_1, \ldots a_k\})$ and the connectivity test using depth-first-search. Thus the overall time complexity for interferences of connectivity k is $\mathcal{O}\left(\binom{|L|}{k} \times |L|^3)\right)$.

3.3 Application examples

The lattice from figure 1 is horizontally decomposable, it has two summand lattices (left picture in figure 7). Note that the summands do not exactly correspond to subtables of the original context, as they have "artificial" top and bottom elements. These are required according to the decomposition definition, and represent the "environment" of the summand.

After initial horizontal decomposition, interference analysis in the right summand revealed a simple interference of connectivity 1, valence 2. This interference is highlighted in figure 7: code piece IV depends on both C and D.[9]

Interference analysis was also applied to several UNIX programs. The right picture in figure 7 presents the horizontal decomposition of the configuration space of the RCS stream editor (see figure 2). From left to right the summands become more complex: on the left are a lot of very small chains representing simple variants, on the right there is a grid-like structure concerning networking which is subject to interference analysis. It reveals an interesting interference with

[9]Interfering suborders are numbered (SL1, SL0). We plan to use color for suborders and interferences.

	t	u	v	w
I	×	×		×
II	×	×	×	
III		×	×	×
IV	×		×	×

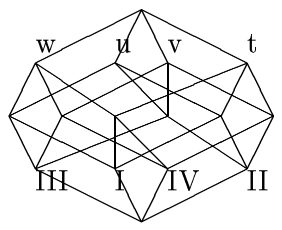


Figure 9: A lattice which is tensorially decomposable

connectivity 1, namly C27, which is infimum of C3 and C26. C3 is labeled "has_rename", C26 is labeled "has_NFS" and C27 is labelled 1426-1426[10]. Thus line 1426 is governed by both "has_NFS" and "has_rename". As these should be orthogonal, the interference is considered harmful. Networking issues and file access variants are not clearly seperated.[11]

NORA/RECS also offers "modularization" based on horizontal summands: for every summand, a simplified source file containing only code pieces from the summand can be created. It is also possible to generate a special "problematic" source file which contains the code producing an interference [Sn95, Le96].

4 Subtensorial Decompositions

In this final chapter, we describe an algorithm for subtensorial decompositions of concept lattices. Subtensorial decompositions are important, as they reveal hidden structures in objects and attributes. Note that there is a connection between subdirect and subtensorial decompositions: if the concepts are combinations of simpler concepts, this is valid for the respective objects and attributes as well. Therefore one expects subdirectly reducible lattices to be subtensorially reducible as well (but not vice versa).

For reengineering purposes, subtensorial decompositions are more promising than subdirect decompositions, as their mathematical features (as described in [GW94], section 3) correspond to "natural" features of "modules".

4.1 Basic definitions and properties

As an example, consider the context and its lattice in figure 9. As we will see, this lattice is a tensorial product of two smaller lattices, where the tensorial decomposition reveals that both objects and attributes are in fact combinations of "simpler" things. In a reengineering context, such substructures of governing symbols are usually not at all obvious, just as a proton hardly reveals that it consists of quarks.

[10]the labels are not shown in the figure, because the representation is very abstract; all labels, extents and intents can be obtained by a simple mouseclick

[11]In fact, C27 revealed a bug in the program (see [KS94]).

	a	b
1	×	
2		×

	x	y
3		×
4	×	

	$\binom{a,}{x}$	$\binom{a,}{y}$	$\binom{b,}{x}$	$\binom{b,}{y}$
$(1,3)$	×	×		×
$(1,4)$	×	×	×	
$(2,3)$		×	×	×
$(2,4)$	×		×	×

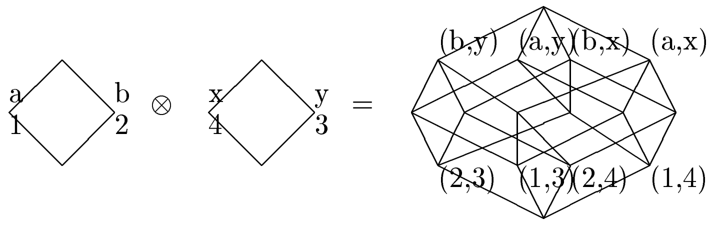


Figure 10: A direct product of contexts and the corresponding tensor product of lattices

Tensorial decompositions are special cases of subtensorial decompositions, which we need in general. First, we need the notion of a direct product of contexts. The direct product of contexts $C_1 = (O_1, A_1, R_1)$, $C_2 = (O_2, A_2, R_2)$ is given by

$$C_1 \times C_2 = (O_1 \times O_2, A_1 \times A_2, \nabla)$$

where $(o_1, o_2)\nabla(a_1, a_2)$ iff $o_1 R_1 a_1$ or $o_2 R_2 a_2$.

As an example, consider two small contexts and their direct product in figure 10. Each cross in an original table becomes a cross rectangle in the direct product. Note that the product context is isomorphic to the context above; this already shows that there is hidden structure in the attributes and objects of the motivating example.

A tensor product of two concept lattices is just the lattice which belongs to the direct product of two contexts [Wi85]:

$$\mathcal{B}(C_1) \otimes \mathcal{B}(C_2) = \mathcal{B}(C_1 \times C_2)$$

Figure 10 presents an example of a tensor product.[12] Note that the direct product of two contexts always contains copies of both original contexts as subcontexts; these can be obtained by deleting rows and columns in the product. Therefore, the tensor product of the lattices will contain copies of both original lattices as sublattices. This characteristic property is also valid for the more general definition of a subtensorial product.

A subtensorial product of two concept lattices is a factor of a tensor product such that the original lattices are still contained as sublattices [GW94]:

$$\mathcal{B}(C_1) \oslash \mathcal{B}(C_2) = (\mathcal{B}(C_1) \otimes \mathcal{B}(C_2))/\theta$$

θ must be a lattice congruence which preserves $\mathcal{B}(C_1)$ and $\mathcal{B}(C_2)$: $[x]_\theta = \{x\}$ for $x \in \mathcal{B}(C_1) \cup \mathcal{B}(C_2)$. Subtensorial products of concept lattices correspond to subdirect products of contexts, that is, certain arrow-closed subcontexts of the direct product of contexts. Ganter and Wille have proven the

Theorem [GW94]. A concept lattice L is a subtensorial product of two concept lattices L_1 and L_2 iff $L_{1,2}$ are sublattices whose union generates L, and every pair $(x_1, x_2) \in L_1 \times L_2$ is weakly distributive.

[12]The lattice is also directly decomposable, as it is isomorphic to the boolean Algebra $\mathbf{2}^4 \cong \mathbf{2}^2 \times \mathbf{2}^2$.

4.2 An algorithm for subtensorial decomposition

The above theorem is the basis for the subtensorial decomposition algorithm. The crucial problem in subtensorial decompositions is to find candidate sublattices, which must then be checked for further properties. Finding sublattices is not a trivial task. The naive approach of enumerating all subsets and checking whether they are sublattices has exponential time complexity and thus forbids itself. Fortunately, formal concept analysis provides the building blocks for an efficient algorithm.[13]

In order to understand the algorithm, we first observe that for a lattice L the mapping $\mathcal{U} : 2^L \to 2^L$ which maps every subset M of L to the sublattice generated by M is a closure operator: we have $M \subseteq \mathcal{U}(M)$, $\mathcal{U}(\mathcal{U}(M)) = \mathcal{U}(M)$, and for $M \subseteq N$, $\mathcal{U}(M) \subseteq \mathcal{U}(N)$. It is well known that Ganter's algorithm for the computation of all concepts of a given context is in fact an algorithm which computes all closed sets of a given closure operator [Ga87]. For computation of concept lattices, the closure operator is the composition of the Galois mappings of the context, denoted $''$. But it is not forbidden to use Ganter's algorithm for other closure operators as well, for example the closure operator $\mathcal{U}$.

In order to implement this idea, we first need an implementation of $\mathcal{U}$. Here is a simple algorithm:

```
UM := M;
REPEAT
  UM2 := UM;
  FOR x IN UM2 DO
    FOR y IN UM2 DO
      UM := UM ∪ $\{x \wedge y, x \vee y\}$;
UNTIL UM=UM2;
```

This algorithm will compute $UM = \mathcal{U}(M)$ for any $M \subseteq L$; it has time complexity $\mathcal{O}(|L|^3)$. Now the algorithm for subtensorial decomposition can be described as follows.

1. Run Ganter's algorithm on L, using closure operator $\mathcal{U}$.[14] This will produce all sublattices of L.
2. For every pair of sublattices L_1 and L_2, check whether their union generates L, and whether they are weakly distributive:
   (a) test whether $\mathcal{U}(L_1 \cup L_2) = L$
   (b) test whether for all $(x_1, x_2) \in L_1 \times L_2$,
   $$\forall g \in J(L) : g \leq x_1 \vee x_2 \iff g \leq x_1 \text{ or } g \leq x_2$$
   as well as
   $$\forall g \in M(L) : g \geq x_1 \wedge x_2 \iff g \geq x_1 \text{ or } g \geq x_2$$
   (Note that in these equivalences, one direction is trivial and need not be tested).

[13]Note the analogy to subdirect decompositions, where in a first step congruences must be found, which are then ckecked for the separation property. Naive generation of congruences is forbidding for complexity reasons. But the computation of the arrow relations opens the door to an efficient algorithm.

[14]This requires that the lattice elements are numbered first, as Ganter's algorithm utlizies the lexicographic order of element sets.

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

	A	B	C	D
I	×	×		
II	×	×	×	
III		×	×	×
IV				×

Figure 11: Source file to be modularized

This algorithm requires that the irreducible elements $J(L)$ resp. $M(L)$ are precomputed (both sets can be determined in time $\mathcal{O}(|L|)$). Both the check for weak distributivity and the test whether the candidates generate L have time complexity $\mathcal{O}(|L|^3)$. If there are n sublattices of L, the overall time complexity thus is $\mathcal{O}(n^2 \cdot |L|^3)$.

4.3 Modularization based on subtensorial decomposition

It is the aim of configuration restructuring to decompose the code into modules such that high cohesion and low coupling between modules is achieved, while the configuration space is left intact[15] [Sn95]. Subtensorial decomposition can be the basis for a restructuring method, as described in this chapter. This method is not yet fully implemented and has not been tested on real-world restructuring problems. But if it succeeds, it can very well be considered a breakthrough in automated restructuring.

The algorithm is best explained by an example. Consider the source text and its configuration table C presented in figure 11 (its lattice L is displayed in figure 5). As demonstrated in [GW94], the corresponding lattice has a subtensorial decomposition. The context tables C_1 and C_2 corresponding to the required sublattices L_1 and L_2 are given in figure 12 ($L_{1,2}$ are displayed in the right part of figure 5, as they are subdirect factors as well). Hence C_1, C_2 are subdirect factors of C. In these subdirect factors, "A, B"means that A and B govern the same code pieces, while "III, IV" means that code pieces III and IV are governed by the same CPP symbols. The modules corresponding to the subcontexts are displayed right to the tables. They are generated straightforward from the factor tables. The direct product of C_1 and C_2 is given in figure 13. The arrow-closed subcontext of $C_1 \times C_2$ which is isomorphic to the original table C is obtained by selecting only those rows and columns marked with a bullet (the reader should switch row III and row IV in figure 11 to see this).

Note that in this example, the code pieces are split into disjoint subsets, but this need not be the case - hence in general the modules are not completely redundant free. As this small example is fictious, we cannot say whether the modularization produced low coupling and high cohesion. In general, this depends on the meaning of A, B, C, D and requires human judgement [Sn95].

[15]the latter property is called *correctness* of the configuration restructuring method.

	A,B	C	D
I	×		
II	×	×	
III,IV			×

$\Rightarrow$

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

	A	B,C,D
I,II,IV		
III		×

$\Rightarrow$

```
#if defined(B)
    && defined(C)
    && defined(D)
...III...
#endif
```

Figure 12: Modularized source file

		$\binom{A,B;}{A}$ ●	$\binom{A,B;}{B,C,D}$ ●	$\binom{C;}{A}$	$\binom{C;}{B,C,D}$ ●	$\binom{D;}{A}$	$\binom{D;}{B,C,D}$ ●
$(I;I,II,IV)$	●	×	×				
$(I;III)$		×	×		×		×
$(II;I,II,IV)$	●	×	×	×	×		
$(II;III)$		×	×	×	×		×
$(IV;I,II,IV)$	●					×	×
$(IV;III)$	●		×		×	×	×

Figure 13: The configuration table of figure 11 as a subdirect product

We will now describe how the original configuration function can be reconstructed from the configuration functions of the subdirect factors of a given configuration table. This is essential, as it guarantees that the configuration space remains intact after modularization. We begin with direct decomposition of configuration tables. First, we show how the derivation function[16] of the product is obtained from the factor's derivation function.

Let $C = C_1 \times C_2 = (O_1 \times O_2, A_1 \times A_2, \nabla)$ be a direct product of $C_1 = (O_1, A_1, R_1)$ and $C_2 = (O_2, A_2, R_2)$. Let $\sigma, \sigma_1, \sigma_2$ be the corresponding derivation functions. Then

$$\begin{aligned}
\sigma((o_1, o_2)) &= \{(a_1, a_2) \in A_1 \times A_2 \mid (o_1, o_2)\nabla(a_1, a_2)\} \\
&= \{(a_1, a_2) \mid o_1 R_1 a_1 \vee o_2 R_2 a_2\} \\
&= \{(a_1, a_2) \mid o_1 R_1 a_1\} \cup \{(a_1, a_2) \mid o_2 R_2 a_2\} \\
&= \sigma_1(o_1) \times A_2 \cup A_1 \times \sigma_2(o_2)
\end{aligned}$$

[16]the "derivation function" has been written "$'$" in the introduction. In order to be able to distinguish several "derivations" for several tables, we now use σ instead: $\sigma_T(o) = \{a \mid (o, a) \in T\}$. Other indices for σ may be used as appropriate.

Now let $\mathcal{K}, \mathcal{K}_1, \mathcal{K}_2$ the configuration functions belonging to C, C_1, C_2. Let $X \subseteq A_1 \times A_2$. Then

$$\begin{array}{rcl}\mathcal{K}(X) & = & \{(o_1, o_2) \in O_1 \times O_2 \mid \sigma((o_1, o_2)) \subseteq X\} \\ & = & \{(o_1, o_2) \mid \sigma_1(o_1) \times A_2 \cup A_1 \times \sigma_2(o_2) \subseteq X\} \\ & = & \{(o_1, o_2) \mid \sigma_1(o_1) \times A_2 \subseteq X \wedge A_1 \times \sigma_2(o_2) \subseteq X\} \\ & = & \Big(\{o_1 \in O_1 \mid \sigma_1(o_1) \times A_2 \subseteq X\} \times O_2\Big) \cap \Big(O_1 \times \{o_2 \in O_2 \mid A_1 \times \sigma_2(o_2) \subseteq X\}\Big)\end{array}$$

Thus we can compute $\mathcal{K}$ from C_1 and C_2 alone – the original configuration table C is no longer needed.

Now we will investigate how congruences affect derivation functions. Let $L_1 = \mathcal{B}(C_1), L_2 = \mathcal{B}(C_2)$, and let $L_1 = L_2/\theta$. The congruence θ corresponds to an arrow-closed subcontext induced by $T = T_\theta \subseteq O \cup A$: $C_1 = (O_2 \cap T, A_2 \cap T, R_{2|O_2 \cap T \times A_2 \cap T})$. For $o \in O_1$, $\sigma_1(o) = ext_1(\gamma_1(o)) = ext_1(\bigwedge_{o \in ext(c)} c) = \bigcup_{c \geq \gamma_1(o)} \alpha_1(c)$. But in fact, $\gamma_1(o)$ as well as all the $c \geq \gamma_1(o)$ represent congruence classes in L_2: $ext_1(c) = \bigcup\{ext_2(c') \cap T \mid \phi(c') = c\}$.[17] Thus $\alpha_1(c) = \bigcup\{\alpha_2(c') \cap T \mid \phi(c') = c\}$ (note that θ is a congruence and thus preserves the lattice order). Hence $\sigma_1(o) = \bigcup_{c \geq \gamma_1(o)} \bigcup_{\phi(c')=c} \alpha_2(c') \cap T$. For any $a \in A_1$, we define $\Theta : A_1 \to 2^{A_2}$ where $\Theta(a) = \bigcup\{\alpha_2(c') \mid \phi(c') = \mu(a)\}$. Thus $\sigma_1(o) = \bigcup_{c_1 \geq \gamma_1(o)} \Theta(\alpha_1(c_1)) \cap T$. Furthermore, $\sigma_2(o) = \bigcup_{c_2 \geq \gamma_2(o)} \alpha_2(c) = \bigcup_{c_1 \geq \phi(\gamma_2(o))} \Theta(\alpha_1(c_1))$.

Therefore we obtain

$$\begin{array}{rcl}\mathcal{K}_2(\bigcup \Theta(X)) \cap T & = & \{o \in O_2 \mid \sigma_2(o) \subseteq \bigcup \Theta(X)\} \cap T \\ & = & \{o \in O_1 \mid \bigcup_{c_1 \geq \phi(\gamma_2(o))} \Theta(\alpha_1(c_1)) \subseteq \bigcup \Theta(X)\} \\ & = & \{o \in O_1 \mid \bigcup_{c_1 \geq \gamma_1(o)} \Theta(\alpha_1(c_1)) \subseteq \bigcup_{a \in X} \Theta(a)\} \\ & = & \{o \in O_1 \mid \bigcup_{c_1 \geq \gamma_1(o)} \alpha_1(c_1) \subseteq X\} \\ & = & \{o \in O_1 \mid \sigma_1(o) \subseteq X\} \\ & = & \mathcal{K}_1(X)\end{array}$$

Thus the configuration function of an arrow-closed subcontext can easily be computed from the configuration function of the original context. It is wise to use a precomputed table for Θ; furthermore, T must be available. Note that several details in the above computation have been left out for space limitations.

Let us now assume that we have a subtensorial decomposition: $L = \mathcal{B}(C) \cong \mathcal{B}(C_1 \times C_2)/\theta$. Putting both above parts together, we obtain

$$\begin{array}{rcl}\mathcal{K}_C(X) & = & \mathcal{K}_{C_1 \times C_2}(\bigcup \Theta(X)) \cap T_\theta \\ & = & \Big(\{o_1 \in O_1 \mid \sigma_1(o_1) \times A_2 \subseteq \bigcup \Theta(X)\} \times O_2\Big) \\ & & \cap \Big(O_1 \times \{o_2 \in O_2 \mid A_1 \times \sigma_2(o_2) \subseteq \bigcup \Theta(X)\}\Big) \cap T_\theta\end{array}$$

This shows how the original configuration function can be determined solely from the tables of the subtensorial factors, and completes our restructuring method. Note that $\mathcal{K}_C(X)$ can be computed quite efficiently, but is nevertheless much more complicated than the original CPP configuration function. Thus one has to pay a price for restructuring: configuration selection becomes more difficult to understand.

An open questions remains: Will subtensorial decomposition indeed produce modules which stick to the principles of high cohesion and low coupling? This can only be answered by empirical studies of real-world programs and will be checked once the implementation is completed.

[17] $\phi : L_2 \to L_1$ is the canonical homomorphism as described in section 2.2: $\phi(X, Y) = (X \cap T, Y \cap T)$.

5 Conclusions

Formal context analysis is a powerful tool for *analysis* of configuration structures, but as a tool for *restructuring*, the approach is still in its infancy. In particular, it turned out that horizontal decompositions do not preserve the complete configuration space and thus can only be used for what Parnas calls "amputation"; subdirect decompositions are very seldom in practice, and it is still unclear whether subtensorial decompositions can be applied to real-world problems. Nevertheless, decomposition of concept lattices has turned out to be a valuable tool for software reengineering and restructuring. Even if automatic restructuring is impossible, the decomposition algorithms lead to very powerful analysis tools.

6 References

[Fu96] P. Funk: Subdirekte Zerlegung von Begriffsverbänden. Diplomarbeit, FB Informatik, TU Braunschweig 1996.

[Ga87] B. Ganter: Algorithmen zur formalen Begriffsanalyse. In B. Ganter, R. Wille (Ed.): Beiträge zur formalen Begriffsanalyse. B.I. 1987, pp. 241-254.

[GW94] B. Ganter, R. Wille: Subtensorial decompositions of concept lattices. Bericht MATH-AL-1-1994, TU Dresden FB Mathematik, 1994.

[Kr93] M. Krone: Reverse Engineering von Konfigurationsstrukturen. Diplomarbeit, FB Informatik, TU Braunschweig 1993.

[KS94] M. Krone, G. Snelting: On the Inference of Configuration Structures from Source Code. Proc. 16th International Conference on Software Engineering, Mai 1994, IEEE Comp. Soc. Press, pp. 49-57.

[Le96] A. Lewien: Analyse und Vereinfachung von Konfigurationsräumen durch horizontale Zerlegung von Begriffsverbänden. Diplomarbeit, FB Informatik, TU Braunschweig 1996.

[LS95] C. Lindig, G. Snelting: Formale Begriffsanalyse im Software Engineering. Erscheint in R. Wille (Hrsg.) Begriffliche Wissensverarbeitung: Methoden und Anwendungen, BI-Wissenschaftsverlag.

[Sn95] G. Snelting: Reengineering of Configurations Based on Mathematical Concept Analysis. Informatik-Bericht 95-02, Januar 1995. To appear in ACM Transactions on Software Engineering and Methodology.

[Wi82] R. Wille: Restructuring lattice theory: An approach based on hierarchies of concepts. In: I. Rival, (Ed.), Ordered Sets, pp. 445-470, Reidel 1982.

[Wi83] R. Wille: Subdirect decomposition of concept lattices. Algebra Universalis 17 (1993), pp. 275-287.

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

[WG93] R. Wille, B. Ganter: Mathematische Theorie der formalen Begriffsanalyse. Skript, TH Darmstadt 1993.