Specification-Based Browsing of Software Component Libraries*

Bernd Fischer
Abt. Softwaretechnologie, TU Braunschweig, D-38092 Braunschweig
fisch@ips.cs.tu-bs.de

Abstract

Specification-based retrieval provides exact content-oriented access to component libraries but requires too much deductive power. Specification-based browsing evades this bottleneck by moving any deduction into an off-line indexing phase. In this paper, we show how match relations are used to build an appropriate index and how formal concept analysis is used to build a suitable navigation structure. This structure has the single-focus property *(i.e.,* any *sensible subset of a library is represented by a single node) and supports* attribute-based *(via explicit component properties) and* object-based *(via implicit component similarities) navigation styles. It thus combines the exact semantics of formal methods with the interactive navigation possibilities of informal methods. Experiments show that current theorem provers can solve enough of the emerging proof problems to make browsing feasible. The navigation structure also indicates situations where additional abstractions are required to build a better index and thus helps to understand and to re-engineer component libraries.*

1 Introduction

Large software libraries represent valuable assets but the larger they grow, the harder it becomes to capitalize them for reuse purposes. The main problems are to keep the overview over the library and to extract appropriate components. This requires better library organizations and retrieval algorithms than a linear search through a flat list of components.

Libraries are thus often structured by syntactic means, e.g., inheritance hierarchies. But this is misleading because it need not to express any semantic relation between components. Information science offers semantic methods for library organization and component retrieval e.g., [17, 24], but these methods are informal because they rely only on the meaning conveyed by words.

*This work is supported by the DFG within the Schwerpunkt "Deduktion", grant Sn11/2-3.

As a more exact alternative, the application of formal specification methods to software libraries has been investigated, starting with [10, 23, 25]. The general idea is quite simple. Each component is indexed with a formal specification which captures its relevant behavior. Any desired relation between two components (e.g., refinement or matching) is expressed by a logical formula composed from the indices. An automated theorem prover is used to check the validity of the formula. If (and only if) the prover succeeds the relation is considered to be established. The most ambitious of these approaches is *specification-based retrieval* [21, 22, 19, 27, 5]. It allows arbitrary specifications as search keys and retrieves all components from a library whose indexes satisfy a given match relation with respect to the key.

However, in spite of all research efforts (cf. [20] for a detailed survey), it is still far away from being practicable. Notwithstanding all progress in automated deduction, the required theorem proving capabilities remain the main bottleneck. Here, we investigate a more practical approach, *specification-based browsing* of component libraries. Its crucial success factor is that *any* time-consuming deduction can be moved into an off-line indexing phase ("preprocessing") and can thus be separated from navigation. The user works only on the pre-processed, fixed *navigation structure* which reflects the semantic properties of the components with respect to the index.

We show that *different* match relations must be used to build an appropriate index and how formal concept analysis can be used to build a concept lattice which serves as navigation structure. Both techniques—specification-based library organization [9, 19] and concept-based browsing [8, 13]—have been proposed before, but their combination is new and unique to this research. It thus combines the exact semantics of formal methods with the interactive navigation of informal methods.

Experiments show that this approach is feasible. Apart from writing the specifications in the first place, indexing can be fully automated. Current theorem provers can solve enough of the emerging problems, even with modest timeouts. Calculation of the concept lattice is fast enough and

navigation works without delay.

Specification-based browsing is not only useful for reuse but also for analyzing, understanding, and re-engineering component libraries. Although browsing is defined via specifications, they are not actually required for navigation. Instead, symbolic names can be used which "hide" the actual formulas. An intelligent choice of such abstractions can thus speed-up and improve understanding. The lattice even indicates situations where additional abstractions are required to build a better index.

2 Browsing vs. retrieval

Library browsing and retrieval are closely related but following [20] a clear distinction can be made. *Retrieval* consists in extracting components which satisfy a *predefined matching criterion*. Its main operation is thus the satisfaction check or *matching*. The criterion is usually given by an arbitrary user-defined search key or *query* which is matched against the candidates' indices. Retrieval supports a top-down design approach: the desired component is first designed (i.e., specified) and then looked up in the library. Its main concern is thus *precision*: components should not be retrieved unless they are absolutely relevant.

Browsing consists in inspecting candidates for possible extraction, but without a predefined criterion. Its main operation is thus *navigation* which determines in what order the components are visited and whether they are visited at all. Browsing supports a bottom-up design approach: the library is first inspected and then the system is designed (i.e., composed) to take maximal advantage of the library. Its main concern is thus *recall*: components should not be rejected unless they are absolutely irrelevant.

Browsing usually works stepwise and we denote the set of all components which can be visited in the next step as the *focus*. In contrast to retrieval, it requires no search key but works on a pre-processed, usually hierarchical navigation structure. The obvious although not optimal way to compute such a structure is to order the components by inclusion on their retrieval results using their own index as query.

In the specification-based case, these differences prove to be crucial for the greater practicability of browsing. The pre-processing of the navigation structure allows us to resort to off-line proving and thus to evade the deductive bottleneck. Less obvious but equally important, the construction of the hierarchy via a cross-match of the component library against itself benefits the proof problems. Since no arbitrary user specifications are involved, the specifications are much more uniform in style. This allows some obvious prover tuning; however, the real gain comes from the absence of data mismatches. Consider for example a graph library where the graphs are represented as map from nodes to node sets and a query using a representation as a list of node pairs. Then, the prover must repeatedly, for each candidate, show that both data representations are equivalent. Although signature matching can mitigate the data mismatch problem [5], it is still the major source of complexity in deduction-based retrieval.

3 Refinement lattices reconsidered

Formal specifications can be used to order components and hence to organize libraries hierarchically. These hierarchies can then be exploited to optimize retrieval or to compute a navigation structure. The obvious question is how to order the components and the obvious answer is by *refinement* or plug-in-compatibility [21, 5]. Given two components G and S with respective axiomatic specifications $(pre_G, post_G)$ and $(pre_S, post_S)$, S is said to refine G (or to be more specific than G, $S \sqsupseteq G$, or G to subsume S), iff

$$(pre_G \Rightarrow pre_S) \wedge (pre_G \wedge post_S \Rightarrow post_G) \qquad (1)$$

holds.[1] Intuitively, (1) expresses the fact that S can be plugged into any place where G is used because it has a wider domain and produces more specific results than G. Using a relational view (i.e., specifications are considered as sets of valid *(input, output)*-pairs), [19] show that (1) defines a partial order which induces a lattice-like structure on the set of all specifications. This structure is generally known as the *refinement lattice* although strictly speaking it is no lattice.

Turning the refinement lattice into a navigation structure for library browsing exposes, however, some unexpected problems. First of all, libraries do not offer enough structure, i.e., the refinement hierarchies they induce are too shallow. While this is a good thing from a design point of view—it simply says that the library contains only little redundancy—it is a bad thing for browsing. It can be overcome by the introduction of meta-nodes or *abstractions*. Such specifications do not represent real, existing components but just factor out similarities between some of them. As an example, consider the specification of an abstract element filter:[2]

```
filter_some (l : list) r : list
pre   l ≠ [ ]
post  ∃l1, l2 : list, i : item · l = l1 ⌢ [i] ⌢ l2 ∧ r = l1 ⌢ l2
```

filter_some specifies only that a singleton element is removed from the list (hence it cannot be empty) but not

[1] For the sake of brevity, we shall omit the quantification over the respective argument and return variables and their identification via type compatibility predicates. For arguments $\vec{x}$ and result variables $\vec{y}$, the full form is $\forall \vec{x}_G, \vec{x}_S, \vec{y}_G, \vec{y}_S \cdot T(\vec{x}_G \cdot \vec{y}_G, \vec{x}_S \cdot \vec{y}_S) \Rightarrow ((pre_G(\vec{x}_G) \Rightarrow pre_S(\vec{x}_S)) \wedge (pre_G(\vec{x}_G) \wedge post_S(\vec{x}_S \cdot \vec{y}_S) \Rightarrow post_G(\vec{x}_G \cdot \vec{y}_G)))$.

[2] We use VDM-SL for our example specifications. Here, $\frown$ means concatenation of lists, $[\,]$ the empty list, $[i]$ a singleton list with item i.

which one. It is thus via (1) refined by both components tail and lead:

tail (l : list) r : list	lead (l : list) r : list
pre $l \neq [\,]$	pre $l \neq [\,]$
post $\exists i : item \cdot l = [i] \frown r$	post $\exists i : item \cdot l = r \frown [i]$

However, a naïve introduction of meta-nodes yields unexpected results. If we introduce another meta-node segment

segment (l : list) r : list
pre true
post $\exists l1, l2 : list \cdot l = l1 \frown r \frown l2$

to capture the property that both components return continuous sublists of their argument, this does not work: neither tail nor lead refine segment. The reason for this at first glance counterintuitive behavior is that segment is specified as a total function ($pre_{segment} = \text{true}$) but both tail and lead are partial. And while we can fix this particular flaw by setting segment's precondition also to $l \neq [\,]$, this soon becomes increasingly infeasible. If the library also contains components which work on sorted lists only, we have to integrate this property into the precondition, too. In effect, if we want an abstraction which captures all segment-like components we have to adjoin all occurring preconditions conjunctively. If, however, two of them are contradictory the result becomes false and segment subsumes the entire library.

The solution to this dilemma is easy. While we can use refinement to index components with abstractions, we additionally need a second relation to model the above situation. Since we are only interested in the effect of the calculation (i.e., the postcondition $post_G$) we can drop pre_G. We want $post_G$ to hold on the appropriate domain only, hence

$$pre_S \wedge post_S \Rightarrow post_G \tag{2}$$

which is also known as conditional compatibility [5] or weak post match [21] in deduction-based retrieval. We can then consider G as *derived attribute* or *feature* [22] of $S, S \sqsupseteq_f G$ because it holds whenever the execution of S was legal (pre_S holds) and terminated ($post_S$ holds.) In our example, segment is a feature of tail and lead, as expected. It is easy to verify that features are inherited along with the refinement relation, i.e., if R refines S and G is a feature of S, then G is a feature of R, too.

A similar problem arises when we want to consider preconditions only. While we can use the abstraction total

total (l : list) r : list
pre true
post true

to subsume all total functions, it is much harder to index partial functions properly. The meta-node

requires_non_empty (l : list) r : list
pre $l \neq [\,]$
post true

correctly subsumes all functions which work on non-empty lists only but it is not really appropriate: it also subsumes all total functions and is thus not discriminative.

Hence, we need a third relation. Since we are now only interested in the properties of the legal domains, we can drop the postconditions. But in contrast to refinement we want the domain of S now to be more restricted, hence

$$pre_S \Rightarrow pre_G \tag{3}$$

Again, G is a derived attribute of S—it is a *requisite*, $S \sqsupseteq_r G$—and using (3), requires_non_empty now works as index. Requisites are also compatible with refinement but in contrast to features their *absence* is propagated. If R refines S and G is no requisite of S, then G cannot be a requisite of R.

	top_sort	run	lead	tail	duplicate_first	requires_non_empty	segment	front_segment	works_on_empty	total	filter_some
top_sort	$\sqsupseteq$								$\sqsubseteq\sqsupseteq$		
run		$\sqsupseteq$					$\sqsupseteq_f$	$\sqsupseteq_f$	$\sqsubseteq\sqsupseteq$	$\sqsupseteq$	
lead			$\sqsupseteq$			$\sqsupseteq_r$	$\sqsupseteq_f$	$\sqsupseteq_f$			$\sqsupseteq$
tail				$\sqsupseteq$		$\sqsupseteq_r$	$\sqsupseteq_f$				$\sqsupseteq$
duplicate_first					$\sqsupseteq$	$\sqsupseteq_r$					

Figure 1. Example index

Figure 1 shows the index for the examples in this paper. The components are represented as rows, the attributes as columns; the symbols indicate which relations have been used to index the components with the respective attributes. We also see that the library is indeed shallow: each *component* indexes only itself.

However, (1-3) are not the only sensible relations we could use. Instead of indexing a component S with its requisites, we could also index S with all requisites it *does not* require, i.e., with all its valid border conditions. In terms of preconditions, this is formalized by

$$\neg pre_G \Rightarrow pre_S \tag{4}$$

and denoted by $S \sqsupseteq_{\bar{r}} G$: G is not a requisite for S, or S also works on G. Hence, we have of course tail $\not\sqsupseteq_{\bar{r}}$

requires_non_empty but for a topological sort function

top_sort (l : list) r : list
pre acyclic(l)
post top_ordered(l) ∧ permutation(l,r)

we have top_sort $\sqsupseteq_{\bar{r}}$ requires_non_empty as expected. However, in principle, (4) is not necessary. We can achieve the same effect using a modified version

works_on_empty (l : list) r : list
pre l = [ ]
post true

of requires_non_empty and refinement: top_sort $\sqsupseteq$ works_on_empty.[3] However, this hides the fact that requires_non_empty and works_on_empty are complementary to each other.

We now use (1-3)[4] to compute an appropriately modified version of the refinement lattice but even this variant is not yet adequate for browsing. It still lacks the *single-focus property*, i.e., it does not contain enough structure to represent the focus by a single node. Consider for example lead and tail. Apart from further refinements, they are the only two components which have the feature segment and are subsumed by filter_some at the same time.[5] Yet there is no meta-node to represent this and a user has to keep his focus on both distinguishing properties to capture the conceptual similarity of the components.

The deeper reason for this is that even the modified refinement lattice has lattice-like properties only on the set of *all possible* specifications, not on arbitrary subsets or libraries. True lattices, on the other hand, have the single-focus property by definition and we will show how to transform the refinement lattice into a true lattice using formal concept analysis.

4 Concept lattices

4.1 Formal concept analysis

Formal concept analysis [31, 7, 2] applies lattice-theoretic methods to investigate abstract relations between objects and their attributes. A concept lattice is a structure with strong mathematical properties which reveals hidden structural and hierarchical properties of the original relation. It can be computed automatically from any given relation.

[3] Notice that this relies on the fact that $post_{\text{works\_on\_empty}} = \text{true}$—otherwise, the postcondition part of (1) would not be valid.

[4] We still need refinement to represent all information of interest. E.g., we cannot split total into a requisite and a feature which have both the value true because both of them index the entire library.

[5] By filter_some we have to remove an element, but by segment we are not allowed to split the list. Hence, there are only the two choices to remove the element at either end of the list.

Definition 1 *A formal* context *is a triple $(\mathcal{O}, \mathcal{A}, \mathcal{R})$ where $\mathcal{O}$ and $\mathcal{A}$ are sets of objects and attributes, respectively, and $\mathcal{R} \subseteq \mathcal{O} \times \mathcal{A}$ is an arbitrary relation.*

Contexts can be imagined as cross tables where the rows are objects and the columns are attributes. Hence, the index shown in Figure 1 can also be considered as a formal context, provided that the different relations (i.e., $\sqsupseteq, \sqsupseteq_r$ and $\sqsupseteq_f$) are merged.

Definition 2 *Let $(\mathcal{O}, \mathcal{A}, \mathcal{R})$ be a context, $O \subseteq \mathcal{O}$ and $A \subseteq \mathcal{A}$. The* common attributes *of O are defined by $\alpha(O) \stackrel{def}{=} \{a \in \mathcal{A} \mid \forall o \in O : (o, a) \in \mathcal{R}\}$, the* common objects *of A by $\omega(A) \stackrel{def}{=} \{o \in \mathcal{O} \mid \forall a \in A : (o, a) \in \mathcal{R}\}$.*

Objects from a context share a set of common attributes and vice versa. Concepts are pairs of objects and attributes which are synonymous and thus characterize each other.

Definition 3 *Let $\mathcal{C}$ be a context. $c = (O, A)$ is called a* concept *of $\mathcal{C}$ iff $\alpha(O) = A$ and $\omega(A) = O$. $\pi_O(c) \stackrel{def}{=} O$ and $\pi_A(c) \stackrel{def}{=} A$ are called c's* extent *and* intent*, respectively. The set of all concepts of $\mathcal{C}$ is denoted by $B(\mathcal{C})$.*

Concepts can be imagined as maximal rectangles (modulo permutation of rows and columns) in the context table, e.g., ({*lead, tail*}, {segment, requires_non_empty, filter_some}). They are partially ordered by inclusion of extents (and intents) such that a concept's extent includes the extent of all of its subconcepts (and its intent includes the intent of all of its superconcepts).

Definition 4 *Let $\mathcal{C}$ be a context, $c_1 = (O_1, A_1), c_2 = (O_2, A_2) \in B(\mathcal{C})$. c_1 and c_2 are ordered by the* subconcept relation*, $c_1 \leq c_2$, iff $O_1 \subseteq O_2$. The structure of B and $\leq$ is denoted by $\mathcal{B}(\mathcal{C})$.*

The intent-part follows by duality. As an immediate consequence of the preceding definitions we get that the strict order corresponds to strict inclusion of extents and intents, i.e., $c_1 < c_2$ iff $O_1 \subset O_2$ and $A_1 \supset A_2$.

The following basic theorem of formal concept analysis states that the structure induced by the concepts of a formal context and their ordering is always a complete lattice and that infimum and supremum can also be expressed by the common attributes and objects. (Cf. Figure 2 for an example lattice.)

Theorem 5 ([31]) *Let $\mathcal{C}$ be a context. Then $\mathcal{B}(\mathcal{C})$ is a complete lattice, the* concept lattice *of $\mathcal{C}$. Its infimum and supremum operation (for any set $I \subset B(\mathcal{C})$ of concepts) are given*

by

$$\bigwedge_{i\in I}(O_i, A_i) = \left(\bigcap_{i\in I} O_i,\ \alpha(\omega(\bigcup_{i\in I} A_i))\right)$$
$$\bigvee_{i\in I}(O_i, A_i) = \left(\omega(\alpha(\bigcup_{i\in I} O_i)),\ \bigcap_{i\in I} A_i\right)$$

The concept lattice is sometimes also refered to as the *Galois lattice* because α and ω form a Galois connection between $\mathcal{O}$ and $\mathcal{A}$. Hence, $\alpha\circ\omega$ and $\omega\circ\alpha$ are closure operators; in Theorem 5 their application maintains the "maximal rectangle" property of the resulting concepts.

Each attribute and object has a uniquely determined defining concept in the lattice which allows a sparse labelling of the lattice. The defining concepts can directly be calculated from the attribute or object, respectively, and need not to be searched in the lattice.

Definition 6 *Let $\mathcal{B}(\mathcal{O}, \mathcal{A}, \mathcal{R})$ be a concept lattice. The* defining concept *of an attribute $a \in \mathcal{A}$ (object $o \in \mathcal{O}$) is the greatest (smallest) concept c such that $a \in \pi_A(c)$ ($o \in \pi_O(c)$) holds. It is denoted by $\mu(a)$ ($\sigma(o)$).*

Theorem 7 ([2]) *In any concept lattice we have $\mu(a) = (\omega(\{a\}), \alpha(\omega(\{a\})))$ and $\sigma(o) = (\omega(\alpha(\{o\})), \alpha(\{o\}))$.*

4.2 From refinement lattices to concept lattices

[13] has shown that keyword-indexed components can be considered as a formal context with the components as objects and the (informal) keywords as attributes. We now lift this idea to formal specifications.

Definition 8 *Let $\mathcal{L} = (L, R, F, A)$ be a formally specified library with components L, requisites R, features F, and abstractions A. Its* induced context *is defined by $\mathcal{C}_\mathcal{L} = (L, L \cup R \cup F \cup A, \sqsupseteq_r \cup \sqsupseteq_f \cup \sqsupseteq)$.*

Again, we consider the components as objects, and, of course, the keywords are replaced by (the names of) the specifications[6] but the context table is slightly more complicated. To prevent different components from "collapsing" into a single concept if the index is insufficient, the component specifications L double as objects and attributes. The relations, however, remain original.

We then calculate the concept lattice from this context. Figure 2 shows the result for the example context. Each bullet represents a concept. The labels over the bullet are the attributes defined at this concept. E.g., the concept (iii) defines the attribute filter_some. However, since attributes in this representation are inherited downwards, its intent π_A is

[6] Wlog. we assume that L, R, F, and A are pairwise disjoint.

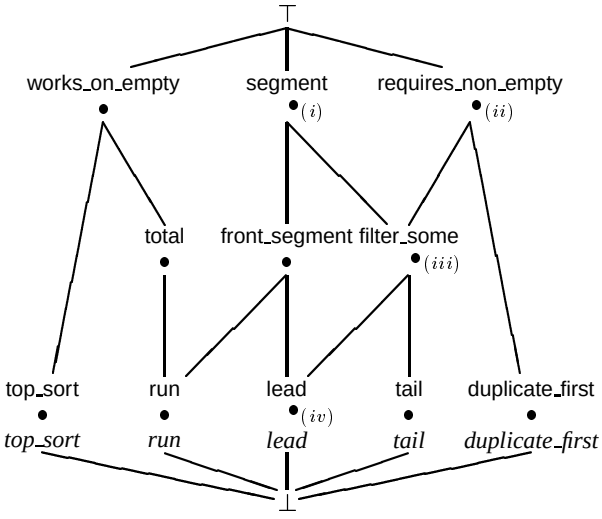

Figure 2. Example lattice

the set {segment, requires_non_empty, filter_some}. None of the attributes are equivalent in the sense that they index the same set of components. Hence, each concept introduces only one attribute. The labels under the bullet denote the objects defined at this concept, e.g., *lead* at (iv). Since none of the actual components subsumes an other, each concept introduces at most one object and is atomic if it introduces an object at all.

The concept lattice is not an "extension" of the refinement lattice: for two attributes a_1, a_2 with $\mu(a_1) \leq \mu(a_2)$ it is possible to be completely unrelated, i.e., neither of the relations (1-3) holds. However, for two reasons, it is an adequate representation of the *index*. First, subconcepts preserve refinement of the original components. Second, a superconcept can be distinguished from any subconcept by an attribute which is *not* valid for at least one component in the extent of the superconcept but is valid for *all* components in the extent of the subconcept. Formally:

Proposition 9 *Let $\mathcal{B}(\mathcal{C}_\mathcal{L})$ be the concept lattice of the context $\mathcal{C}_\mathcal{L}$ induced by a library $\mathcal{L}$ and $c_1, c_2 \in \mathcal{C}_\mathcal{L}$ with $c_1 < c_2$. Then exists $n \in \pi_O(c_2)$ such that either*

1. $\exists\, m \in \pi_O(c_1) \cdot m \neq n \wedge m \sqsupseteq n$, *or*
2. $\exists\, a \in R \cdot a \in \pi_A(c_1) \wedge a \notin \pi_A(c_2) \wedge n \not\sqsupseteq_r a \vee$
   $\exists\, a \in F \cdot a \in \pi_A(c_1) \wedge a \notin \pi_A(c_2) \wedge n \not\sqsupseteq_f a \vee$
   $\exists\, a \in A \cdot a \in \pi_A(c_1) \wedge a \notin \pi_A(c_2) \wedge n \not\sqsupseteq a$.

This proposition, which follows from definitions 3 and 4, makes the concept lattice already suitable for specification-based navigation: when we move from a superconcept to a subconcept, we either follow an original refinement relation on components, or we discard at least one and thus due to

the lattice structure all components from the extent which do not share the property n. However, we can impose even more structure if we double R and use $\sqsupseteq_{\bar{r}}$ in addition to define the induced context. Then, Proposition 9 holds appropriately and, additionally, we get

Proposition 10 *Let $\mathcal{B}(\mathcal{C}_\mathcal{L})$ be the concept lattice of the context $\mathcal{C}_\mathcal{L}$ induced by a library $\mathcal{L}$. Then, for any two complementary requisites $a, \bar{a} \in R$ we have $\forall c \in L \cdot n \sqsupseteq_r a \Leftrightarrow \not\sqsupseteq_{\bar{r}} \bar{a}$ and consequently $\mu(a) \sqcap \mu(\bar{a}) = \bot$ and $\mu(a) \sqcup \mu(\bar{a}) = \top$.*

Hence, the defining concepts of two complementary requisites are complementary to each other in the lattice. Moreover, their extents divide the entire library into two partitions which is not the case for two arbitrary complementary nodes in the lattice.

5 Navigation in concept lattices

[13] has also shown how concept lattices can be used as navigation structure for interactive and incremental retrieval (i.e., browsing in our terminology). The focus is represented by (the extent of) a concept. Narrowing the focus is a downward movement in the lattice and is done in two steps:

1. The user selects an additional attribute. As a consequence of the lattice structure, the system can support this selection by calculating all attributes which actually narrow the focus but do not sweep it entirely. It can thus prevent navigation into dead ends (i.e., an empty focus.)

2. The system calculates the new focus in the lattice as the meet (which exists due to Theorem 5) of the actual focus and the defining concept of the selected attribute (obtained by Theorem 7.)

Similarly, the focus can also be widened again by de-selecting an attribute. The system then calculates the new focus using the join operation.

In the specification-based case, navigation works quite similar. We use the derived properties (i.e., R, F, and A) as navigation attributes. Since the property sets are pairwise disjoint, we can even split the set of navigation attributes into three dimensions. These dimensions are not independent of each other but can be selected independently because all interdependencies are contained in the concepts of the lattice. If we use the modified context (i.e., double R and use (1-4)), we get a fourth dimension. This is still independent but due to Proposition 10, independent selection from R and $\bar{R}$ is not beneficial. Instead, we can toggle between them, in addition to selection/de-selection.

Initially, all attributes are de-selected and the focus concept is $\top$: the focus is the entire library. Now, for an example, assume that we select segment. This reduces the focus to $\pi_O(i) = \{$*run, lead, tail*$\}$. Further refinement is possible by attributes whose defining concepts have a strictly smaller but non-bottom meet with the current focus concept. Thus, for (i), any navigation attribute is possible. If we select requires_non_empty, the new focus concept is $(i) \sqcap (ii) = (iii)$, i.e., the choice of requires_non_empty eliminates *run* from the focus. Moreover, it leaves front_segment as the only possible further refinement.

This navigation style is *attribute-based*: the focus is essentially a function of the selected attributes. Due to their dual nature, concept-lattices also allow *object-based navigation*. Here, the user selects or de-selects a single component and the system calculates the new focus similarly. However, selecting an additional component widens the focus and is thus realized by the join operation.

While attribute-based navigation depends on the explicit and learned choice of functional properties and thus is more suited for reuse purposes, object-based navigation exposes implicit conceptual similarities of components: the intent of the focus concept contains all properties which are common to all selected components; its extent also contains all other components which share these properties, even if they have not been selected explicitly. Hence, it is more appropriate for library understanding and re-engineering.

6 Practical aspects

We made a series of experiments to support the claim that browsing is more practical in the specification-based case than retrieval. For these, we used a variant of the list processing library which we also used in our retrieval experiments [5]. It comprises 5 requisites, 31 features, and 86 components and abstractions. All example specifications in this paper are taken from that library.

6.1 Calculation of the refinement lattice

Even if the calculation of the refinement lattice is done in advance and is thus not time-criticle in principle, it is not obvious that it is feasible at all. Two questions are of main concern:

1. How high is the computational effort in practice?

2. How difficult are the proof problems in practice? Are current theorem provers powerful enough?

The answer to both questions depends on the number and structure of the arising proof problems.

At first glance, it seems that we have to check each requisite, feature, abstraction, and component against each other

to calculate the modified refinement lattice. However, in practice this can be optimized due to three observations. First, we do not need to compare the components and abstractions pairwise but can use recursive comparison as in [9] because refinement is transitive. Then, we do not need to check requisites and features against each other but only against the components and abstractions. Finally, since the former are compatible with refinement, we can "sink them in" once we have the refinement lattice on the other nodes ready. In the worst case, the number of problems is thus $|R \cup F \cup A \cup L| \cdot |A \cup L|$. Nevertheless, still too many problems arise to be handled manually. As in other software engineering applications, a fully automated system is required which feeds and controls the prover. However, the sheer numbers become a problem only because most of the proof problems (approximately 85% in our experiments) are logically invalid and thus not provable at all. But theorem provers do usually not check for unprovability and are thus stopped by time-out only. Hence, dedicated disproving filters are required.

Nevertheless, the computation is practically feasible. Using techniques from [5] we generated the full set of more than 14.000 proof tasks (i.e., "ready-to-run" versions of the problems which also contain appropriate axioms and prover control information) and filtered out approximately 9.100 as unprovable. This took approximately 7 hours on a Sun UltraSparc 170. For simplicity, we did not use the optimizations explained above. This would have reduced the original number of tasks to about 11.000.

We then used the automated theorem prover SPASS [30] on a network of 16 PCs to check the surviving tasks. With a time-out of 60 seconds, SPASS was able to solve 1.250 tasks. For the remaining 3.080 problems, we re-generated the tasks, using a different subset of the axioms. After a third iteration, SPASS had solved a total of 1.460 or almost 80% of the valid problems. This required a total of approximately 210 hours runtime, or equivalently, a weekend of real time.

6.2 Calculation of the concept lattice

Concept lattices can grow exponentially in the number of attributes and objects. In practice, however, the worst case rarely occurs and a polynomial behavior is usual. [13] contains more experimental evidence for this.

For our example library, the concept lattices from the full (i.e., manually computed) and the approximated (i.e., computed using SPASS) refinement lattices contained 153 and 180 concepts, respectively. Their computation took approximately a second and is thus negligible compared to the time required for proving.

6.3 Navigation

During our experiments it became quickly obvious that neither the modified refinement lattice nor the concept lattice are suitable for presentation because they are too big and complex. [13] makes the same observation and describes a simple text-based interface which works on the attribute and object names only. We are currently adapting this system. The navigation process itself, however, is very fast: the system responds without noticeable delay, even for much larger concept lattices than we are currently investigating.

6.4 Scale-up

Scaling specification-based browsing to large libraries is a serious challenge: a library with 10 requisites, 100 features and 1000 abstractions and components gives in the worst case already rise to more than 1.1 million proof tasks.

To handle such many tasks, it is necessary to exploit the structure of the *subsumption* lattice as soon as it emerges. E.g., if front_segment $\sqsupseteq$ segment has already been established[7] then lead $\sqsupseteq_f$ front_segment should be checked before lead $\sqsupseteq_f$ segment. If the former holds, the latter holds automatically, due to transitivity.

Similarly, invalid proof tasks can be saved, if the *absence* of features (requisites, abstractions) is exploited. If, e.g., top_sort $\not\sqsupseteq_f$ segment can be shown, top_sort cannot satisfy any of the features more specific than segment, and the corresponding proof tasks can be dismissed. However, due to the undecidability of first-order logics, it is not legal to conclude the absence of the feature segment from the failure to prove top_sort $\sqsupseteq_f$ segment. Instead, in practice an appropriately modified version

$$pre_S \wedge post_S \Rightarrow \neg post_G$$

of the proof task must be checked which unfortunately increases the total number of tasks.

Computationally more complicated components, e.g., graph or numerical algorithms, obviously induce more complicated proof tasks. Here the key to scaling is to find an abstract domain representation which factors out most of the complexity, supported by using the right abstractions and features. Then the conceptual difference between the specifications S and G which accounts for most of the difficulties can be kept small and the prover has a reasonable chance to succeed.

Using the above techniques, even large, diverse libraries of functional components can be tackled. Other component types, e.g., objects or entire modules, fit in principle also into this framework but require an appropriate redefinition

[7] Note that this need not necessarily to be checked because both front_segment and segment are features.

of the different match conditions. However, components whose effects cannot be expressed naturally in a pre/post-condition style, e.g., graphical routines, cannot be handled and there is no obvious way to extend specification-based browsing appropriately.

6.5 Knowledge acquisition

The formal specifications of the library components and some initial abstractions[8] must be supplied. Once this seed is available, specification-based browsing can already support further knowledge acquisition.

Consider for example a seed comprising filter_some, segment, tail, lead, and

```
run (l : list) r : list
pre   true
post  ∃l1 : list · l = r ⌢ l1  ∧  ordered(r)
  ∧   ∀i : item, l2 : list · l = r ⌢ [i] ⌢ l2  ⇒ ¬ordered(r ⌢ [i])
```

which computes the longest ordered initial subsegment (i.e., run) of a list. From this seed, an initial concept lattice is calculated. Object-based navigation confirms that both tail and lead already have a common superconcept, which has the attributes filter_some and segment, and, as expected, no other objects. But it also reveals that there is no concept which has the extent of lead and run only—selecting both also causes tail to appear. To disambiguate tail, the user must introduce a feature

```
front_segment (l : list) r : list
pre   true
post  ∃l1 : list · l = r ⌢ l1
```

which factors out the common property of lead and run.

7 Related work

Most work on applying specification-based techniques to software libraries examines retrieval only. Relevant for browsing are the investigation of different match relations [21] and their effect on software reuse [6, 5]. [22] introduced features as indexes to speed up retrieval. The deductive synthesis system AMPHION [29] composes programs from retrieved matching components but does not support user-guided library exploration.

[9] builds a two-tiered hierarchy from the library. The lower level is based on a modified definition of subsumption which works modulo arbitrary user-defined congruences on literals and is thus unsound in general. The upper level uses a similarity metric derived from the normal forms of the specifications. This hierarchy is then visualized to support browsing. [19] only use subsumption to build a hierarchical representation of a library and exploit that only to optimize retrieval.

In programming language research, [15] and [12] apply formal methods to the specification and verification of object-oriented class libraries. There, behavioral subtyping corresponds to subsumption.

Concept lattices or Galois lattices have been developed as a means to structure arbitrary observations. They have already been applied to various problems in software engineering, e.g., inference of configuration structures [11] or identification of modules [14, 28] and objects [26] in legacy programs. Their application to software component libraries, however, seems to be obvious only in retrospect, and there is only little related work. [8] also uses concept lattices for navigation but presents the entire lattice to the user and offers only a subset of all possible attributes for selection. As far as navigation is concerned, [13] is thus most closely related to our own work. But there, object-based navigation, which is instrumental in knowledge acquisition, is not supported.

8 Conclusions

Only specification-based methods can provide exact content-oriented access to software components. Retrieval, however, still requires more deductive power than current theorem provers and hardware can offer. Browsing can evade this bottleneck by moving any time-consuming deduction into an off-line indexing phase.

In this paper, we have shown that different match relations must be used to index a library properly and how this index is turned into a navigation structure using formal concept analysis. Experiments show that it is feasible to calculate an approximation of the index which is accurate enough for browsing purposes, using current theorem provers and hardware (e.g., SPASS on a small network of PCs.) The computational effort, however, is still high.

The concept lattice reveals the implicit structure of a library as it follows from the index. It can even indicate situations where a finer index is required. Due to its dual nature, the lattice allows two complementary navigation styles which are based either on attributes or on objects. Due to the lattice nature, both navigation styles automatically have the single-focus property and refrain the user from dead ends.

In our approach, theorem provers are used to derive formally defined properties of components. For navigation, these formal definitions are still available but not actually required—symbolic property names suffice. However, since informally defined and derived properties (e.g., reliability) are usually also represented by symbolic names (e.g., *trusthworty*), concept-based browsing allows a smooth inte-

[8] Initial requisites and features can be derived automatically by splitting of the supplied specifications. Any resulting indiscriminate attributes are merged into a single concept by construction of the lattice.

gration of formal and informal attributes and thus refutes a conjecture of [1] that formal and informal methods are incompatible. Moreover, informal attributes can even be used to distinguish functional equivalent variants of a component from each other.

Future work especially concerns scale-up. We expect the fraction of non-theorems to grow further with increasing library size; dedicated disproving techniques are thus one area of interest. Since the remaining tasks are homogeneous in style, learning theorem provers [4, 3] can be expected to perform well on them.

Acknowledgments

Christian Lindig's work on concept-based retrieval also triggered this research; discussions with him greatly improved my own understanding of formal concept analysis. Comments by Christian, Jens Krinke, Gregor Snelting, and the reviewers improved the presentation of this paper. Christoph Weidenbach did the actual theorem proving at the MPII.

References

[1] N. Boudriga, A. Mili, and R. Mittermeir. Semantic-based software retrieval to support rapid prototyping. *Structured Programming*, 13:109–127, 1992.

[2] B. A. Davey and H. A. Priestley. *Introduction to Lattices and Order*. Cambridge University Press, Cambridge, UK, 2nd edition, 1990.

[3] J. Denzinger, M. Kronenburg, and S. Schulz. DISCOUNT: A distributed and learning equational prover. *J. Automated Reasoning*, 18:189–198, 1997.

[4] J. Denzinger and S. Schulz. Learning domain knowledge to improve theorem proving. In McRobbie and Slaney [18], pages 62–76.

[5] B. Fischer, J. M. P. Schumann, and G. Snelting. Deduction-based software component retrieval. In W. Bibel and P. H. Schmitt, editors, *Automated Deduction - A Basis for Applications*, pages 265–292. Kluwer, Dordrecht, 1998.

[6] B. Fischer and G. Snelting. Reuse by contract. In G. T. Leavens and M. Sitaraman, editors, *Proc. ESEC-FSE Workshop on Foundations of Component-Based Sytems*, pages 91–100, Zürich, Sept. 1997.

[7] B. Ganter and R. Wille. *Formale Begriffsanalyse - Mathematische Grundlagen*. Springer, Berlin, 1996.

[8] R. Godin, J. Gecsei, and C. Pichet. Design of a browsing interface for information retrieval. In N. J. Belkin and C. J. van Rijsbergen, editors, *Proc. Twelfth Annual Intl. ACM SIGIR Conf. on Research and Development in Information Retrieval*, pages 32–39, Cambridge, Massachusetts, June 1989. ACM Press.

[9] J. Jeng and B. H. C. Cheng. Using formal methods to construct a software component library. In I. Sommerville and M. Paul, editors, *Proc. 4th European Software Engineering Conf.*, volume 717 of *Lect. Notes Comp. Sci.*, pages 397–417, Garmisch-Partenkirchen, Sept. 1993. Springer.

[10] S. Katz, C. A. Richter, and K. S. The. PARIS: A system for reusing partially interpreted schemas. In *Proc. 9th Intl. Conf. Software Engineering*, pages 377–385, Montery, CA, Mar. 1987. IEEE Comp. Soc. Press.

[11] M. Krone and G. Snelting. On the inference of configuration structures from source code. In B. Fadini, editor, *Proc. 16th Intl. Conf. Software Engineering*, pages 49–57, Sorrento, Italy, May 1994. IEEE Comp. Soc. Press.

[12] G. T. Leavens and W. E. Weihl. Specification and verification of object-oriented programs using supertype abstraction. *Acta Informatica*, 32(8):705–778, Nov. 1995.

[13] C. Lindig. Concept-based component retrieval. In J. Köhler, F. Giunchiglia, C. Green, and C. Walther, editors, *Working Notes of the IJCAI-95 Workshop: Formal Approaches to the Reuse of Plans, Proofs, and Programs*, pages 21–25, Montréal, Aug. 1995.

[14] C. Lindig and G. Snelting. Assessing modular structure of legacy code based on mathematical concept analysis. In T. Maibaum and M. Zelkowitz, editors, *Proc. 18th Intl. Conf. Software Engineering*, pages 349–359, Berlin, Mar. 1996. IEEE Comp. Soc. Press.

[15] B. Liskov and J. M. Wing. A behavioral notion of subtyping. *ACM Trans. Programming Languages and Systems*, 16(6):1811–1841, Nov. 1994.

[16] M. Lowry and Y. Ledru, editors. *Proc. 12th Intl. Conf. Automated Software Engineering*, Lake Tahoe, Nov. 1997.

[17] Y. S. Maarek, D. M. Berry, and G. E. Kaiser. An information retrieval approach for automatically constructing software libraries. *IEEE Trans. Software Engineering*, SE-17(8):800–813, 1991.

[18] M. A. McRobbie and J. K. Slaney, editors. *Proc. 13th Intl. Conf. Automated Deduction*, volume 1104 of *Lect. Notes Artifical Intelligence*, New Brunswick, NJ, July-Aug. 1996. Springer.

[19] A. Mili, R. Mili, and R. Mittermeir. Storing and retrieving software components: A refinement-based system. *IEEE Trans. Software Engineering*, SE-23(7):445–460, July 1997.

[20] A. Mili, R. Mili, and R. Mittermeir. A survey of software reuse libraries. *Annals of Software Engineering*, 1998. To appear.

[21] A. Moorman Zaremski and J. M. Wing. Specification matching of software components. In G. E. Kaiser, editor, *Proc. 3rd ACM SIGSOFT Symp. Foundations of Software Engineering*, pages 6–17, Washington, DC, Oct. 1995. ACM Press.

[22] J. Penix, P. Baraona, and P. Alexander. Classification and retrieval of reusable components using semantic features. In *Proc. 10th Knowledge-Based Software Engineering Conf.*, pages 131–138, Boston, MA, Nov. 1995. IEEE Comp. Soc. Press.

[23] D. E. Perry. The Inscape environment. In *Proc. 11th Intl. Conf. Software Engineering*, pages 2–12. IEEE Comp. Soc. Press, May 1987.

[24] R. Prieto-Díaz. Implementing faceted classification for software reuse. *Communications of the ACM*, 34(5):89–97, May 1991.

[25] E. J. Rollins and J. M. Wing. Specifications as search keys for software libraries. In K. Furukawa, editor, *Proc. 8th*

Intl. Conf. Symp. Logic Programming, pages 173–187, Paris, June 24-28 1991. MIT Press.

[26] H. A. Sahraoui, W. Melo, H. Lounis, and F. Dumont. Applying concept formation methods to object identificaton in procedural code. In Lowry and Ledru [16], pages 210–218.

[27] J. M. P. Schumann and B. Fischer. NORA/HAMMR: Making deduction-based software component retrieval practical. In Lowry and Ledru [16], pages 246–254.

[28] M. Siff and T. Reps. Identifying modules via concept analysis. In M. J. Harrold and G. Visaggio, editors, *Proc. IEEE Intl. Conf. on Software Maintenance*, pages 170–179, Bari, Italy, 1997. IEEE Comp. Soc. Press.

[29] M. Stickel, R. Waldinger, M. Lowry, T. Pressburger, and I. Underwood. Deductive composition of astronomical software from subroutine libraries. In A. Bundy, editor, *Proc. 12th Intl. Conf. Automated Deduction*, volume 814 of *Lect. Notes Artifical Intelligence*, pages 341–355, Nancy, June-July 1994. Springer.

[30] C. Weidenbach, B. Gaede, and G. Rock. Spass and Flotter version 0.42. In McRobbie and Slaney [18], pages 141–145.

[31] R. Wille. Restructuring lattice theory: An approach based on hierarchies of concepts. In I. Rival, editor, *Ordered Sets*, pages 445–470. Reidel, 1982.