Automated Software Engineering, 7, 179–200, 2000
© 2000 Kluwer Academic Publishers. Manufactured in The Netherlands.

Specification-Based Browsing of Software Component Libraries

BERND FISCHER fisch@ptolemy.arc.nasa.gov
RIACS/NASA Ames Research Center, Moffett Field, CA-94035 USA

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.

Keywords: software reuse, software component libraries, browsing, retrieval, formal specifications

1. Introduction

Large software libraries represent valuable assets but the larger they grow, the harder it becomes to capitalize on them for reuse purposes. The two main problems are to keep the overview over the library and to extract appropriate components. Solving both problems requires better library organizations and better 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 syntactic means need not express any semantic relation between components. Information science offers semantic methods for library organization and component retrieval (Maarek et al., 1991; Prieto-Díaz, 1991) but these methods are informal because they rely only on the meaning conveyed by words.

As a more exact alternative, the application of formal specification methods to software libraries has been investigated, starting with Katz et al. (1987), Perry (1989), and Rollins and Wing (1991). The general idea is quite simple. Each component is associated with a formal specification which captures its relevant behavior. Any desired relation between two components (e.g., refinement, matching, or reusability) is expressed by a logical formula composed from the associated specifications. 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* (Jeng and Cheng, 1995; Moorman Zaremski and Wing, 1997; Penix et al., 1995; Mili et al., 1997; Schumann and Fischer, 1997; Fischer et al., 1998). 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. (Mili et al., 1998) 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* difficult and time-consuming deduction can be moved into an off-line indexing phase ("pre-processing") 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 discuss how formal concept analysis can be used to build a concept lattice which serves as navigation structure. Both techniques—specification-based library organization (Jeng and Cheng, 1993; Mili et al., 1997) and concept-based browsing (Godin et al., 1989; Lindig, 1995a, b)—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 computational resources. 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 Mili et al. (1998) a clear distinction can be made. *Retrieval* consists of extracting components that 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 of 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 are visible in one step as the *focus*. In contrast to retrieval, it requires no search key but works on a pre-processed, usually hierarchical navigation structure which is computed from an indexing scheme. Browsing can be implemented via retrieval if the indexing scheme is considered as a set of queries; the canonical navigation structure is then obtained by ordering the indexing scheme by inclusion on the retrieval results.

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 browsing setup also benefits the proof problems in two ways. First, since all involved specifications are written by a *library administrator* and not by arbitrary users, the specifications can be kept much more uniform in style. This allows some obvious prover tuning. Second, the *data mismatch* problem (i.e., the use of different data representations in the query and component specifications, respectively) can be mitigated. Consider for example a graph library where the graphs are represented as map from nodes to node sets. The library administrator can take this into account and avoid a mismatch when formulating the indexing scheme while a user of a retrieval system does not have this information and might thus as well choose a representation as a list of node pairs. This, however, forces the prover to show for each candidate that both data representations are equivalent. Moreover, the library administrator can provide data reification functions (Jones, 1990) which can be stored parallel to the library and used to resolve occurring mismatches.

A restricted class of mismatches can even be resolved automatically. A *signature mismatch* is a simple structural data mismatch between two components, e.g., different number or order of parameters. Signature matching techniques (Rittri, 1991; Moorman Zaremski and Wing, 1995) resolve such simple mismatches using unification of the type terms, taking the typing rules and semantic models of the applied programming language into account. More advanced approaches as for example by DiCosmo (1995) even construct isomorphisms which can subsequently be used as reification functions. As a side effect, signature matching also identifies and renames the appropriate parameters of the different specifications. For convenience, all example specifications in this paper already mirror this identification and renaming step.

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 (Jones, 1990; Moorman Zaremski and Wing, 1997; Fischer et al., 1998). 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

$$(pre_G \Rightarrow pre_S) \wedge (pre_G \wedge post_S \Rightarrow post_G) \tag{1}$$

holds.[1] Intuitively, (1) expresses the fact that a more specific component S can be plugged into any place where the more general component G is used or required because it has a wider domain and produces more detailed results than G. Using a relational view (i.e., specifications are considered as sets of valid (*input*, *output*)-pairs), Mili et al. (1997) 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 $\mathsf{l} \neq [\ ]$
post $\exists \mathsf{l1}, \mathsf{l2} : \mathsf{list}, \mathsf{i} : \mathsf{item} \cdot \mathsf{l} = \mathsf{l1} \frown [\mathsf{i}] \frown \mathsf{l2} \wedge \mathsf{r} = \mathsf{l1} \frown \mathsf{l2}$

filter_some specifies only that a singleton element is removed from the list (hence it cannot be empty) but not which one. It is thus via (1) refined by both components tail and lead:

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

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

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 $\exists \mathsf{l1}, \mathsf{l2} : \mathsf{list} \cdot \mathsf{l} = \mathsf{l1} \frown \mathsf{r} \frown \mathsf{l2}$

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 ($pre_{\mathsf{segment}} = \mathsf{true}$) but both tail and lead are partial. And while we can fix this particular flaw by setting segment's precondition also to $\mathsf{l} \neq [\ ]$, 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 (Fischer et al., 1998) or weak post match (Moorman Zaremski and Wing, 1997) in deduction-based retrieval. We can then consider G as *derived attribute* or *feature* (Penix et al., 1995) of S, $S \sqsupseteq_f G$ because it holds whenever the execution of the component associated with 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. We can use the simple abstraction total

```
total  ( )
pre    true
post   true
```

which does not even refer to the input- and output-parameters of the actual component to subsume all total functions. But it is much harder to index partial functions properly. The meta-node

```
requires_non_empty (l : list)
pre   l ≠ [ ]
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.

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

While we need all three relations to express all indexing information of interest,[3] (1)–(3) are not the only sensible relations we could use. Instead of indexing a component S with

	p_order_list	reverse	run	lead	tail	copy_first	requires_non_empty	segment	front_segment	works_on_empty	total	filter_some
cf. page	184	198	194	182	182	198	183	182	196	184	183	196
p_order_list	$\sqsupseteq$									$\sqsupseteq$		
reverse		$\sqsupseteq$								$\sqsupseteq$	$\sqsupseteq$	
run			$\sqsupseteq$					$\sqsupseteq_f$	$\sqsupseteq_f$	$\sqsupseteq$	$\sqsupseteq$	
lead				$\sqsupseteq$			$\sqsupseteq_r$	$\sqsupseteq_f$	$\sqsupseteq_f$			$\sqsupseteq$
tail					$\sqsupseteq$		$\sqsupseteq_r$	$\sqsupseteq_f$				$\sqsupseteq$
copy_first						$\sqsupseteq$	$\sqsupseteq_r$					

Figure 1. Example index.

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 function

```
p_order_list (l : list) r : list
pre   partial_order(l)
post  partially_ordered(l) ∧ permutation(l,r)
```

which partially orders a list according to an underlying relation (provided that this relation constitutes a partial order on the elements in the argument list), we have p_order_list $\sqsupseteq_{\bar{r}}$ requires_non_empty as expected: (4) rewrites to l $=$ [ ] $\Rightarrow$ partial_order(l) and empty lists are always partially ordered. In principle, (4) is not necessary. We can achieve the same effect using a modified version

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

of requires_non_empty and refinement: p_order_list $\sqsupseteq$ works_on_empty. However, this relies on the fact that $post_{\text{works\_on\_empty}}$ $=$ true—otherwise, the postcondition part of (1) would not be valid. More important, it also hides the fact that requires_non_empty and works_on_empty are complementary to each other. We will later show how this fact can be used to improve browsing.

We now use (1)–(3) 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: 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 of removing the element either at the beginning or at the end of the list. 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 (Wille, 1982; Ganter and Wille, 1996; Davey and Priestley, 1990) 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.

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 $\mathcal{R} \subseteq O \times \mathcal{A}$ 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 $\alpha(O) \overset{\text{def}}{=} \{a \in \mathcal{A} \mid \forall o \in O : (o, a) \in \mathcal{R}\}$, the *common objects* of A by $\omega(A) \overset{\text{def}}{=} \{o \in \mathcal{O} \mid \forall a \in A : (o, a) \in \mathcal{R}\}$.

Objects from a concept 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 $\omega(A) = O$. $\pi_O(c) \overset{\text{def}}{=} O$ and $\pi_A(c) \overset{\text{def}}{=} A$ 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 greatest lower bound or *meet* and least upper bound or *join* can also be expressed by the common attributes and objects. (Cf. figure 2 for an example lattice.)

Theorem 5 (Wille, 1982). *Let $\mathcal{C}$ be a context. Then $\mathcal{B}(\mathcal{C})$ is a complete lattice, the* concept lattice *of $\mathcal{C}$. Its meet and join operation (for any set $I \subset B(\mathcal{C})$ of concepts) are given by*

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

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

The concept lattice is sometimes also referred 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 on $\mathcal{A}$ and $\mathcal{O}$, respectively; in Theorem 5 their application maintains the "maximal rectangle" property of the resulting concepts. Consider for example the meet of the concepts $(i) =$ ({*run*, *lead*, *tail*}, {segment}) and $(ii) =$ ({*lead*, *tail*, *copy_first*}, {requires_non_empty}) in figure 2. The intersection of their extents is {*lead*, *tail*}, i.e., the objects common to both concepts. The straightforward union of their intents, however, would be too small. Since both *lead* and *tail* also have the attribute filter_some in common, ({*lead*, *tail*}, {segment, requires_non_empty}) is not a valid concept. This omission is repaired by the application of the closure operator. Essentially, the meet and join operations thus only factor out the common objects and attributes, respectively, of any given set of concepts.

Each attribute and object has a uniquely determined defining concept in the lattice which allows a sparse labeling of the lattice. The defining concepts can be calculated directly from the attribute or object, respectively, and need not 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 (Davey and Priestley, 1990). *For any concept lattice we have $\mu(a) = (\omega(\{a\}), \alpha(\omega(\{a\})))$ and $\sigma(o) = (\omega(\alpha(\{o\})), \alpha(\{o\}))$.*

For example, for the defining concept of the attribute filter_segment we first calculate $\omega(\{\text{filter\_segment}\}) = \{\textit{lead}, \textit{tail}\}$ and then obtain $\mu(\text{filter\_segment}) = (\textit{iii})$ by application of the closure operator (cf. figure 2).

4.2. *From refinement lattices to concept lattices*

Lindig (1995a) 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[4] 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 context's relation is obtained by merging the different relations; the individual relations, however, remain unchanged.

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 above the bullet are the

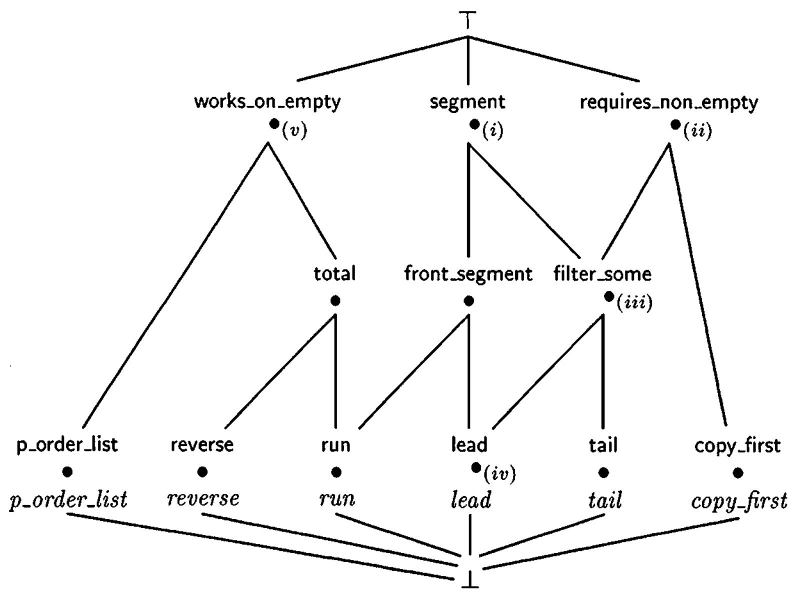


Figure 2. Example concept lattice.

attributes defined by 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 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 below the bullet denote the objects defined by this concept, e.g., *lead* is defined by the concept (*iv*). Objects are inherited upwards such that *lead* is also in the extent of all of its superconcepts. Since none of the actual components in the test library subsumes an other, each concept introduces at most one object and is atomic if it introduces an object at all.

In general, 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. E.g., $\mu(\text{filter\_some}) \leq \mu(\text{segment})$ but segment is not a feature of the abstraction filter_some. Moreover, the actual position of the defining concept of an attribute depends on the components actually contained in the library. If for example *run* is removed from the library, total is "moved downwards" and merged with reverse because *reverse* then remains as the only component in the library bearing this attribute. However, for two reasons, the concept lattice is an adequate representation of the *indexing scheme*. 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 $\mathcal{B}(C_L)$ 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 that do not share the property a. For example, by moving from (i) to (iii), we discard the component *run* from the context because it does not share the the properties filter_some or requires_non_empty which are both in the intent of the concept (iii).

However, we can impose even more structure if we duplicate 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 $\mathcal{B}(C_L)$ 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 $\forall c \in L \cdot c \sqsupseteq_r a \Leftrightarrow c \not\sqsupseteq_{\bar{r}} \bar{a}$ and consequently $\mu(a) \wedge \mu(\bar{a}) = \perp$ and $\mu(a) \vee \mu(\bar{a}) = \top$.*

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 elements of the lattice. In the example, it is easy to show that $c \sqsupseteq \mathsf{works\_on\_empty} \Leftrightarrow c \sqsupseteq_{\bar{r}} \mathsf{requires\_non\_empty}$; hence, we can define $\mathsf{works\_on\_empty} = \overline{\mathsf{requires\_non\_empty}}$. From this, we know that the respective defining concepts (v) and (ii) are complementary lattice elements and, moreover, that $\pi_O(v) \cup \pi_O(ii) = L$ and $\pi_O(v) \cap \pi_O(ii) = \emptyset$ hold. This partioning property does not hold for the other complementary elements, e.g., (v) and (iii).

5. Navigation in concept lattices

Lindig (1995a) 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 similarly. 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., duplicate 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 $(i) \wedge (ii) = (iii)$, 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 that are common to all selected components; its extent also contains all other components that 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 (Fischer et al., 1998). It comprises 5 requisites, 31 features, 44 abstractions, and 39 components. 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-critical 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 (Jeng and Cheng, 1993) 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 or decision procedures for (at least some) of the involved theories are required.

Nevertheless, the computation is practically feasible. Using techniques from (Fischer et al., 1998) 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 (Weidenbach et al., 1996) 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 problems, we re-generated a different version of the tasks, using a different axiomatization of the background theory and different prover control parameters. After a third iteration, SPASS had solved a total of 1.460 or almost 80% of the original 1836 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

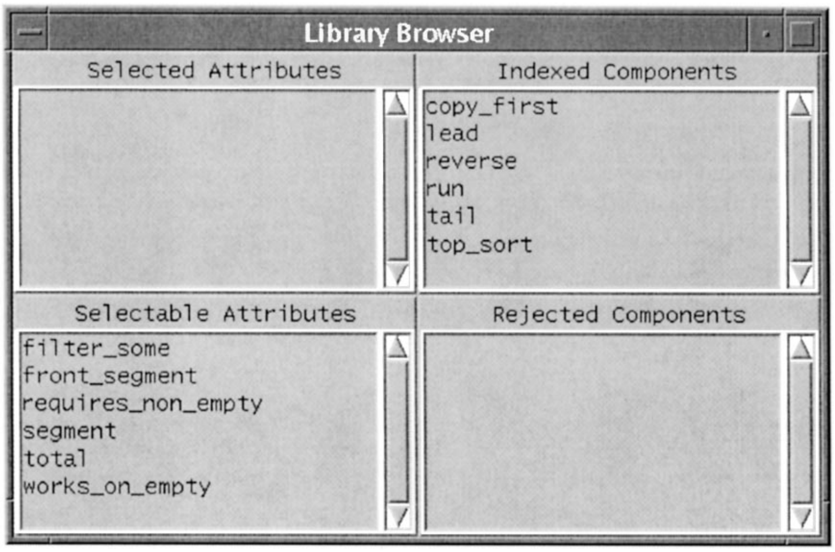
Concept lattices can grow exponentially in the number of attributes and objects. In practice, however, the worst case rarely occurs and a non-exponential behavior is usual. Godin et al. (1993) and Lindig (1995a) give more experimental evidence for this. Moreover, algorithms to construct the concept lattice incrementally are known (Godin and Missaoui, 1994 ; Godin et al., 1995).

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

The granularity of the concept lattice is adequate for browsing purposes. In the optimal case, the lattice contains $153 - 39 - 1 = 113$ inner nodes (i.e., essentially the non-atomic elements) which represent all meaningful—as defined by the indexing attributes—of the possible 2^{39} subsets of library components. 33 of the 113 inner nodes are combinations of the original attributes which are discovered by concept analysis. These can be used to build a better index (cf. Section 6.5).

In the experiment, the approximated concept lattice derived from the automatically computed context contains more elements. The additional elements are required to cover the sparser and more rugged context which contains more but smaller maximal rectangles. However, even though the automatically computed index is always a subcontext of the full index, due to the soundness but incompleteness of the applied theorem prover(s), the derived concept lattices are arbitrary with respect to each other. In particular, no obvious algebraic relation (e.g., subdirect image) between the two lattices holds.

In practice, the difference between the lattices plays no major role. As with specification-based retrieval, the incompleteness of the applied prover translates into a loss of recall, i.e., a concept might not contain all components that actually share its attributes. Due to its soundness, however, all components in the focus share all displayed attributes.



```
copy_first
lead
reverse
run
tail
top_sort
```

```
filter_some
front_segment
requires_non_empty
segment
total
works_on_empty
```

Figure 3. Browser interface: Initial state.

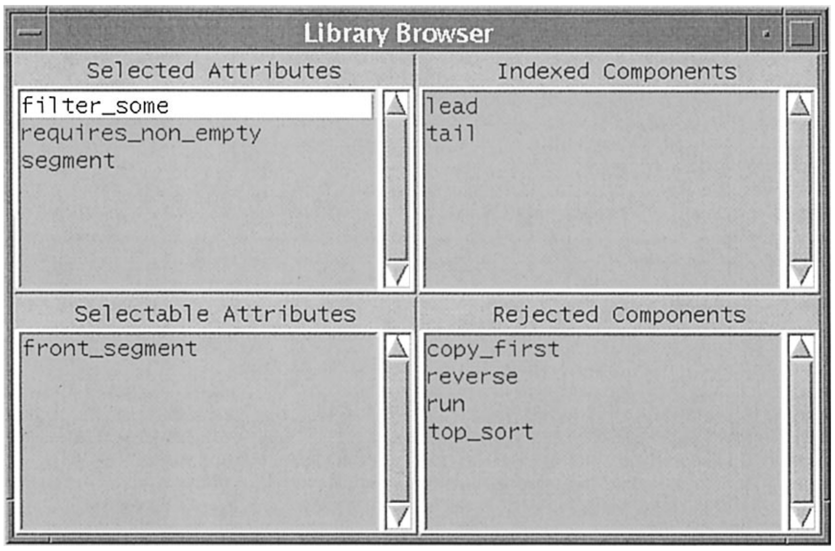
6.3. Navigation

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. Lindig (1995a) makes the same observation and describes a simple text-based interface which works on the attribute and object names only. The navigation process is very fast: the system responds without noticeable delay, even for much larger concept lattices than we are currently investigating.

This system can easily be adapted to our case; moreover, it can be modified to support object-based navigation also. Figures 3 and 4 show two snapshots of a conceived simple interface which does not distinguish between the different attribute types.

The system displays selectable and selected attributes, respectively, to the left, and the indexed and rejected components, respectively, to the right. Each of the four lists is *active* in the sense that the user can control the navigation from it: clicking on a selectable attribute or indexed component narrows the focus, clicking on a selected attribute or rejected component widens the focus. For any action, the system calculates the new focus as described in Section 5 and updates the display appropriately.

A closer inspection of the two snapshots seems to reveal that the interface "drops" attributes during browsing while it always displays all components in the library. This, however, is a feature and not a bug. In order to prevent an attribute-based navigation into a dead end, the system offers only the attributes in a sensible neighborhood of the current focus (i.e., attributes having a non-bottom meet with the focus) for further selection. Non-selected and non-selectable attributes (i.e., total and works_on_empty) can be ignored because they can in no way contribute to navigation. For object-based navigation, the situation is different



Selected Attributes:
filter_some
requires_non_empty
segment

Indexed Components:
lead
tail

Selectable Attributes:
front_segment

Rejected Components:
copy_first
reverse
run
top_sort

Figure 4. Browser interface: After selecting segment and requires_non_empty.

because it has no equivalent notion of dead ends—the entire library (i.e., the extent of the top concept) is a suitable selection. Hence, all *components* are selectable and must thus be displayed always.

The actual definitions of the attributes as well as the original source codes of the components can be displayed in separate windows; these can be requested by a further single mouse mouse click on the respective attribute or component name.

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 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. However, since both front_segment and segment are features, establishing front_segment $\sqsupseteq$ segment initially increases the number of tasks compared to the estimate in Section 6.1; the effectiveness of this optimization thus depends on the particular library.

Similarly, invalid proof tasks can be saved, if the *absence* of features (requisites, abstractions) is exploited. If, e.g., p_order_list $\not\sqsupseteq_f$ segment can be shown, p_order_list 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 p_order_list $\sqsupseteq_f$ 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 again 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 that factors out most of the complexity, supported by using the right abstractions and features. Then the conceptual difference between the specifications *S* and *G* that accounts for most of the difficulties can be kept small and the prover has a reasonable chance to succeed.

The above techniques should be sufficient to tackle even large, diverse libraries of functional components as for example the Standard Template Library (Stepanov and Lee, 1994). Other component types, e.g., objects or entire modules, fit in principle also into this framework but require an appropriate redefinition 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

All specification-based library access methods critically depend on adequate and correct component specifications. These can either be supplied and verified by the component designer during the forward engineering phase, using standard program verification techniques, or in a reverse engineering phase reconstructed from existing libraries, using program understanding techniques as for example described by Gannod et al. (1998).

These component specifications can then be used to construct an initial indexing seed. Initial abstractions can be derived semi-automatically from the original specifications by logical weakening; Gannod et al. (1998) also describe a syntax-directed method for this. Initial requisites and features can be derived automatically by splitting of the original specifications; any resulting indiscriminate attributes are merged into a single concept by construction of the lattice. Once such an initial seed is available, specification-based browsing can already support further knowledge acquisition.

Consider for example a seed comprising the same component specifications as in figure 2 where

```
run (l : list) r : list
pre  true
post $\exists l1 : list \cdot l = r \frown l1 \wedge ordered(r)$
   $\wedge\ \forall i : item, l2 : list \cdot l = r \frown [i] \frown l2 \Rightarrow \neg ordered(r \frown [i])$
```

computes the longest ordered initial subsegment (i.e., run) of a list, but only the initial properties works_on_empty, requires_non_empty, total, and segment. From this seed, an

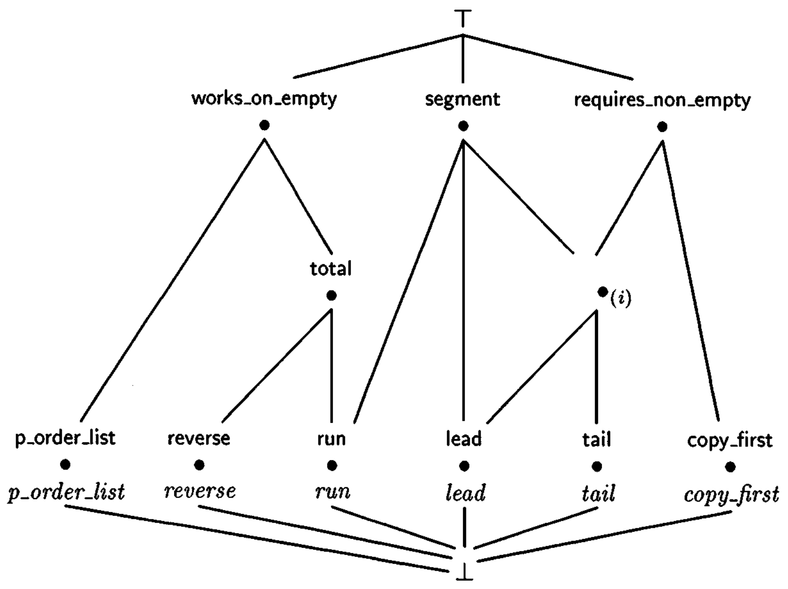


Figure 5. Concept lattice for initial seed.

initial concept lattice is calculated. The structure of this lattice, as shown in figure 5, then helps to improve indexing.

Typically, two situations prompt improvements:

– unlabeled concepts
– unexpected extents

A concept remains unlabelled when a set of components (i.e., its extent) is characterized only by a combination of more general attributes but not by a single specific attribute. In figure 5, the concept (i) remains unlabelled, i.e., defines no attribute. Its extent *lead* and *tail* is thus characterized by segment and requires_non_empty, but both properties are defined by more general concepts and thus index larger subsets of the library than only *lead* and *tail*. To capture the similarity of both components, the user can introduce the missing specific attribute, e.g., filter_some. In general, the least specific attribute indexing the same subset of the library (i.e., having the same extent) results from the separate conjunction of the pre- and postconditions of all attributes in the intent of the concept; this attribute is a feature (requisite) only if all involved attributes are features (requisites) and an abstraction otherwise.

The introduction of missing attributes to relabel previously unlabelled concepts provides more explicit information about the concepts but does not change the structure of the concept lattice. In principle, it does thus not improve the accuracy of the actual browsing process. In practice, however, the lattice may well be re-arranged, due to the incompleteness of the prover. A theoretical improvement can be guaranteed if the introduced attributes apply to some set of components previously not occurring as extent of a concept and the prover is complete. Such attributes require a genuine understanding of the involved specifications; they can thus not be derived automatically as the attributes for unlabelled concepts but object-based navigation can at least help to identify appropriate candidate sets: if the common superconcept of a set of similar components unexpectedly also includes some "intuitively" different components in its extent, then this intuitive difference can be formalized and used as additional attribute. Consider for example again the initial seed lattice of figure 5. Object-based navigation confirms that both tail and lead already have a common superconcept, that has the attributes requires_non_empty 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 the feature

front_segment (l : list) r : list
pre true
post $\exists l1 : \text{list} \cdot l = r \frown l1$

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 (Moorman Zaremski and Wing, 1997) and their effect on software reuse (Fischer and Snelting, 1997; Fischer et al., 1998). Penix et al. (1995) introduce features as indexes to speed up retrieval. The deductive synthesis system AMPHION (Stickel et al., 1994) composes programs from retrieved matching components but does not support user-guided library exploration.

Jeng and Cheng (1993) build 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. Mili et al. (1997) only use subsumption to build a hierarchical representation of a library and exploit that only to optimize retrieval.

In programming language research, (Liskov and Wing, 1994) and (Leavens and Weihl, 1995) 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 (Krone and Snelting, 1994), identification of modules (Lindig and Snelting, 1996; Siff and Reps, 1997) and objects (Sahraoui et al., 1997) in legacy programs, or reorganization of inheritance hierarchies (Snelting and Tip, 1998). Snelting (1998) gives a comprehensive overview of applications in program analysis. The application of concept lattices to software component libraries, however, seems to be obvious only in retrospect, and there is only little related work. Godin et al. (1989) 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, the work by Lindig (1995a, b) 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 reaching 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., *trustworthy*), concept-based browsing allows a smooth integration of formal and informal attributes and thus refutes a conjecture of Boudriga et al. (1992) 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 essentially concerns scaling-up to larger components and larger libraries. The most important aspect here is the tractability of the emerging proof tasks, especially when data mismatches need to be resolved via complicated reification functions. We also expect the fraction of non-theorems to grow further with increasing library size; dedicated disproving techniques are thus another area of interest. Since the remaining tasks are homogeneous in style, learning theorem provers (Denzinger and Schulz, 1996; Denzinger et al., 1997) can be expected to perform well on them. Finally, to handle really large libraries the simple interface described in Section 6.3 probably needs to be adapted, e.g., by integrating some of the hierarchical structure provided by the lattice.

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, Johann Schumann, and the reviewers improved the presentation of this paper. Christoph Weidenbach at the MPII Saarbrücken provided the resources for the actual theorem proving. This work started at the TU Braunschweig, Germany, and has been supported by the DFG within the Schwerpunkt “Deduktion”, grant Sn11/2-3.

Appendix

This short appendix contains the remaining specifications used in the text.

$\mathsf{copy\_first\ (l : list)\ r : list}$
$\mathsf{pre}\quad \mathsf{l} \neq [\ \ ]$
$\mathsf{post}\ \ \mathsf{r} = [\mathsf{hd\ l}] \curvearrowright \mathsf{l}$

$\mathsf{reverse\ (l : list)\ r : list}$
$\mathsf{pre}\quad \mathsf{true}$
$\mathsf{post}\ \ \mathsf{len\ l} = \mathsf{len\ r} \wedge \forall\, \mathsf{i : nat} \cdot \mathsf{i} \leq \mathsf{len\ l} \Rightarrow \mathsf{l(i)} = \mathsf{r(1} + \mathsf{len\ l} - \mathsf{i)}$

Notes

1. For the sake of brevity, we 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-style specifications for our examples. Here, $\curvearrowright$ means concatenation of lists, $[\ \ ]$ the empty list, $[i]$ a singleton list with item i.
3. For example, we cannot split the abstraction total into a requisite and a feature which have both the value true because both of them index the entire library.
4. Without loss of generality we can assume that L, R, F, and A are pairwise disjoint.

References

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

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

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

Denzinger, J. and Schulz, S. 1996. Learning domain knowledge to improve theorem proving. In (McRobbie and Slaney, 1996). pp. 62–76.

DiCosmo, R. 1995. *Isomorphisms of Types: From λ-Calculus to Information Retrieval and Language Design*, Boston: Birkäuser. Progress in Theoretical Computer Science, Vol. 14.

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

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

Gannod, G.C., Chen, Y., and Cheng, B.H.C. 1998. An automated approach for supporting software reuse via reverse engineering. In D.F. Redmiles and B. Nuseibeh, editors, *Proc. 13th Intl. Conf. Automated Software Engineering*, Honolulu, Hawaii, pp. 79–86.

Ganter, B. and Wille, R. 1996. *Formale Begriffsanalyse—Mathematische Grundlagen*. Berlin: Springer.

Godin, R., Gecsei, J., and Pichet, C. 1989. 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*, Cambridge, Massachusetts, pp. 32–39.

Godin, R. and Missaoui, R. 1994. An incremental concept formation approach for learning from databases. *Theoretical Computer Science*, 133(2):387–419.

Godin, R., Missaoui, R., and Alaoui, H. 1995. Incremental concept formation algorithms based on galois (concept) lattices. *Computational Intelligence*, 11(2):246–267.

Godin, R., Missaoui, R., and April, A. 1993. Experimental comparison of navigation in a galois lattice with conventional information retrieval methods. *Intl. J. Man-Machine Studies*, 38:747–767.

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

Jeng, J.-J. and Cheng, B.H.C. 1995. Specification matching for software reuse: A foundation. In M.H. Samadzadeh and M.K. Zand, editors, *Proc. ACM SIGSOFT Symp. Software Reusability*, Seattle, Washington, pp. 97–105.

Jones, C.B. 1990. *Systematic Software Development Using VDM*, 2nd ed. Englewood Cliffs, New Jersey: Prentice-Hall.

Katz, S., Richter, C.A., and The, K.S. 1987. PARIS: A system for reusing partially interpreted schemas. In *Proc. 9th Intl. Conf. Software Engineering*, Montery, CA, pp. 377–385.

Krone, M. and Snelting, G. 1994. On the inference of configuration structures from source code. In B. Fadini, editor, *Proc. 16th Intl. Conf. Software Engineering*, Sorrento, Italy, pp. 49–57.

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

Lindig, C. 1995a. 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*, Montréal, pp. 21–25.

Lindig, C. 1995b. Komponentensuche mit Begriffen. In G. Snelting, editor, *Proc. Softwaretechnik 95*, Braunschweig, pp. 67–75.

Lindig, C. and Snelting, G. 1996. Assessing modular structure of legacy code based on mathematical concept analysis. In T. Maibaum and M. Zelkowitz, editors, *Proc. 18th Intl. Conf. Software Engineering*, Berlin, pp. 349–359.

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

Lowry, M. and Ledru, Y. (editors). 1997. *Proc. 12th Intl. Conf. Automated Software Engineering*, Lake Tahoe, IEEE Comp. Soc. Press.

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

McRobbie, M.A. and Slaney, J.K. (editors). 1996. *Proc. 13th Intl. Conf. Automated Deduction*. Lect. Notes Artifical Intelligence, Vol. 1104, New Brunswick, NJ, Springer.

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

Mili, A., Mili, R., and Mittermeir, R. 1998. A survey of software reuse libraries. *Annals of Software Engineering*, 5, 349–414.

Moorman Zaremski, A. and Wing, J.M. 1995. Signature matching: A tool for using software libraries. *ACM Trans. Software Engineering and Methodology*, pp. 146–170.

Moorman Zaremski, A. and Wing, J.M. 1997. Specification matching of software components. *ACM Trans. Software Engineering and Methodology*, 6(4):333–369.

Penix, J., Baraona, P., and Alexander, P. 1995. Classification and retrieval of reusable components using semantic features. In D. Setliff, editor, *Proc. 10th Knowledge-Based Software Engineering Conf.*, Boston, MA, pp. 131–138.

Perry, D.E. 1989. The inscape environment. In *Proc. 11th Intl. Conf. Software Engineering*, Pittsburg. PA, pp. 2–12.

Prieto-Díaz, R. 1991. Implementing faceted classification for software reuse. *Comm. ACM*, 34(5):89–97.

Rittri, M. 1991. Using types as search keys in function libraries. *J. Functional Programming*, 1(1):71–89.

Rollins, E.J. and Wing, J.M. 1991. Specifications as search keys for software libraries. In K. Furukawa, editor, *Proc. 8th Intl. Conf. Symp. Logic Programming*, Paris, pp. 173–187.

Sahraoui, H.A., Melo, W., Lounis, H., and Dumont, F. 1997. Applying concept formation methods to object identificaton in procedural code. In (Lowry and Ledru, 1997). pp. 210–218.

Schumann, J.M.P. and Fischer, B. 1997. NORA/HAMMR: Making deduction-based software component retrieval practical. In (Lowry and Ledru, 1997). pp. 246–254.

Siff, M. and Reps, T. 1997. Identifying modules via concept analysis. In: M.J. Harrold and G. Visaggio, editors, *Proc. IEEE Intl. Conf. on Software Maintenance*, Bari, Italy, pp. 170–179.

Snelting, G. 1998. Concept Analysis—A new framework for program understanding. In T. Ball and F. Tip, editors, *Proc. ACM SIGPLAN/SIGFSOFT Workshop on Program Analysis for Software Tools and Engineering*, Montreal, Canada, pp. 1–10.

Snelting, G. and Tip, F. 1998, Reengineering class hierarchies using concept analysis. In B. Scherlis, editor, *Proc. 6th ACM SIGSOFT Symp. Foundations of Software Engineering*, Lake Buena Vista, Florida, pp. 99–110.

Stepanov, A.A. and Lee, M. 1994. The standard template library. Technical Report X3J16/94-0095, WG21/N0482, ISO Programming Language C++ Project.

Stickel, M., Waldinger, R., Lowry, M., Pressburger, T., and Underwood, I. 1994. Deductive composition of astronomical software from subroutine libraries. In A. Bundy, editor, *Proc. 12th Intl. Conf. Automated Deduction*. Lect. Notes Artifical Intelligence, Vol. 814, Nancy, pp. 341–355.

Weidenbach, C., Gaede, B., and Rock, G. 1996. Spass and flotter version 0.42. In (McRobbie and Slaney, 1996). pp. 141–145.

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