On the Retrieval of Reusable Software Components**

Patrick Shicheng Chen
Fakultät f. Math. & Informatik
Universität Passau
D-8390 Passau

Rolf Hennicker
Institut f. Informatik
Universität München
D-8000 München 40

Matthias Jarke
Lehrstuhl f. Informatik V
RWTH Aachen
D-5100 Aachen

Abstract

Starting from the principle of software reusability through formal specifications we suggest a model for the retrieval of reusable components utilizing the search techniques in database management systems. The formal specification language of software components is ***ASL****. Component specifications will be translated into a specification written in the knowledge representation language* ***Telos*** *for storage and other manipulation. The retrieval of software components is based on signature matching between the signatures of goal specifications and those of reusable components. In this way, we overcome some of the main problems with respect to retrieval such as representation of reusable components, representation of goal specification, and name differences in the software. The retrieval mechanism is supported by the Database Management System* ***ConceptBase****.*

1 Introduction

Software reusability serves two purposes: one is to reduce the costs of software development, and the other is to enhance the reliability of software. Reusable software should be retrievable. This means we should organize the libraries of reusable modules and provide a good database searching mechanism such that client programmers can find appropriate modules easily. Retrieval is in the broad sense combined with other operations such as adaptation and integration. Until now, cataloguing is a technique widely in use for retrieving components from software libraries. With the growing amount of software components in the library, say some thousands or some ten thousands of components, a simple lookup in the catalogue cannot find suitable components for our reuse need.

Main problems in the retrievability of software components are numerous and hard to tackle with. We just mention some of them:

1. Keyword-based systems are inadequate because they are based on the syntax not on the semantics of the components (cf. [15]).

2. Lack of formalism makes the systematic retrieval difficult in which a standard description of reusable components and their storage should be provided.

3. A software component is a complex object that can only be described completely with a suitable data modeling language.

**A version of the paper is presented at the Second International Workshop on Software Reuse.

4. For the identification of software components abstraction from particular names with the same meaning (such as Int, Integer) should be taken into account.

5. It happens often that the stored components are not exactly the ones we need. In this case the database should *suggest* us some components of interest instead of giving us a negative answer.

2 Previous Approaches and Our Suggestion

One of the influential systems applying techniques of information retrieval (IR) in retrieving software components is the one developed by Prieto-Diaz (cf. [13]) which classifies software through detailed analysis. This work has attracted great attention. Systems described in [6], [15], [10] utilize the same principle. These approaches present advantages in terms of cost, portability, and scalability. Information about the software can be obtained from the documentation of the software (such as in [10]), or from the categories that a software belongs to (such as in [15]).

Influenced by Artificial Intelligence, many researchers argue that knowledge is needed for retrieving components in the software library which includes structural information of the software organization and semantic information of the software. An example of intelligent systems is described in [1] where a component descriptor is a knowledge frame. Critics on these approaches include requirement of domain analysis, a great deal of pre-encoded, manually provided semantic information, great expense for building the search tool, and bad extensibility.

Our approach sets out from another viewpoint which particularly focuses on the correctness of software reuse. We believe that the correct identification and reuse of a software component is only possible if a formal specification of the component's behaviour is provided which allows to abstract from irrelevant implementation details. Then, for the implementation of a given goal specification SPg appropriate candidate components can be identified by comparing their abstract specifications with the goal specification SPg. For the representation of reusable components and goal specifications we use an object-oriented database environment which provides a powerful retrieval mechanism. The key concepts of our approach are:

1. Software reuse is based on formal specifications. A reusable component consists of an abstract algebraic specification and at least one concrete implementation.

2. The abstract formal specifications of components are mapped into database objects. Reusable components are thus represented by objects in the database where they can be retrieved by the knowledge base management system.

3. The goal specification to be implemented is mapped into a query to the database which can be processed by the knowledge base management system.

4. The retrieved components should be adapted before reuse, since the new program systems are in general different from the stored ones. Hence best (not necessarily exact) match of goal specification and reusable components is supported. In contrast to keyword-based systems abstraction of names is considered to be necessary.

3 Software Reuse through Formal Specifications

3.1 Reusable Components

For the formal description of reusable components we use the approach of [8] which is based on the algebraic specification language ASL. A reusable component is defined as an (unordered) tree of algebraic specifications where any two consecutive nodes are related by a formally defined implementation relation, denoted by the symbol $\sim\!\!\rightsquigarrow$. Hence a reusable component represents several levels of abstraction where the root of a component RC, denoted by *root(RC)*, provides the most abstract problem description and the leaves represent different implementations.

3.2 Algebraic Specifications

For writing specifications we use the algebraic specification language ASL (cf. [14]). For instance, Figure 1 shows a specification of double-ended queues over arbitrary elements. On the right of the keyword `sorts`, two sorts `Dequeue` and `ElemOfDequeue` are declared which indicate that the data structure of double-ended queues has two basic sets of data: a set of queues (denoted by `Dequeue`) and a set of elements (denoted by `ElemOfDequeue`). After the keyword `opns`, four operation symbols together with their associated functionality are declared: The constant `empty` denotes the empty queue, the operation `insert` adds a new element to a queue and the operations `first` and `last` deliver the first and the last element of a queue, respectively. The properties of the operations are specified by the axioms of the specification which, in general, can be arbitrary formulas of first order logic.

The sorts and the operation symbols constitute the `signature` of a specification SP, in the following denoted by *sig(SP)*. Formally, a signature is a pair Σ = (S, F) consisting of a set S of sorts and a set F of operation symbols (also simply called operations) whereby any operation f $\in$ F is equipped with a functionality $s_1, ..., s_n \rightarrow s$ with $s_1, ..., s_n, s \in S$. Semantically, any sort s $\in$ S denotes a set of data and the operation symbols are interpreted as functions on the underlying data sets.

The specification language ASL contains several specification building constructs which allow to build large structured specifications on top of existing ones. For the following it is enough to consider basic specifications and the constructs for renaming and declaration of export interfaces since these constructs are particularly suited for the adaptation of specifications (and implementations) in order to make them reusable for new purposes.

3.2.1 Basic Specifications

Let Σ = (S, F) be a signature and E be a set of axioms describing the properties of the operations. Then

sorts S **opns** F **axioms** E

is a basic specification with signature Σ. For instance, DEQUEUE is a basic specification.

3.2.2 Renaming

For the definition of renamings we need the notion of a signature morphism: Let Σ = (S, F) and Σ' = (S', F') be signatures with sets of sorts S, S' resp. and sets of operation symbols F, F' resp.. A

```
spec DEQUEUE =
 sorts {Dequeue, ElemOfDequeue}
 opns
  {empty: --> Dequeue
   first: Dequeue --> ElemOfDequeue
   last: Dequeue --> ElemOfDequeue
   insert:ElemOfDequeue,Dequeue
                          --> Dequeue}
 axioms
  {first(insert(x,q)) = x
   last(insert(x,empty)) = x
   last(insert(x,insert(y,q))) =
                   last(insert(y,q))}
```

Figure 1: Specification of double-ended queues

signature morphism $\rho : \Sigma \to \Sigma'$ is a pair (ρ-sorts, ρ-opns) of mappings ρ-sorts: S $\to$ S', ρ-opns: F $\to$ F' which is compatible with the functionality of operations, i.e. for all f $\in$ F with functionality $s_1, ..., s_n \to s$, ρ-opns(f) has functionality ρ-sorts(s_1), ..., ρ-sorts(s_n) $\to$ ρ-sorts(s). In the following bijective signature morphisms are also-called *renamings*.

Now, let SP be a specification, Σ be a signature and $\rho : sig(SP) \to \Sigma$ be a renaming. Then

rename SP **by** ρ

is a specification with signature Σ, i.e. the sorts and operations of SP are renamed w.r.t. ρ.

3.2.3 Declaration of Export Interfaces

Let SP be a specification and $\Sigma \subseteq$ sig(SP) be a subsignature, i.e. Σ consists of a subset of the sorts of SP and a subset of the operation symbols of SP. Then

export Σ **from** SP

is a specification with signature Σ, i.e. all sorts and operations of SP not belonging to Σ are forgotten.

The semantics of a specification SP is determined by its signature *sig(SP)* and by its class of models, denoted by *Mod(SP)*. For instance, the model class of DEQUEUE consists of all data structures with two sets of data and four functions (corresponding to the sorts and to the operation symbols of DEQUEUE) such that the axioms of DEQUEUE are satisfied. For a detailed definition of syntax and semantics of all specification building constructs of ASL we refer to [14].

3.3 Implementation of Specifications

Based on the semantics of specifications an implementation relation is defined which says that a specification SP-IMPL is an `implementation` of a specification SP, written SP $\leadsto$SP-IMPL, if sig(SP-IMPL) = sig(SP) and Mod(SP-IMPL) $\subseteq$ Mod(SP). Hence any model of an implementing specification SP-IMPL satisfies the requirements of the abstract specification SP. The implementation relation provides the formal basis for correctness proofs. In particular, it will be used for the correct adaptation of reused components to new applications.

It is important to remark that the implementation relation is transitive and that all specification building constructs are monotonic w.r.t. the implementation relation. In particular, if SP $\leadsto$SP-IMPL, then the following statements also hold:
rename SP **by** ρ $\leadsto$**rename** SP-IMPL **by** ρ and
export Σ **from** SP $\leadsto$**export** Σ **from** SP-IMPL.

3.4 A Formal Method for the Reuse of Components

Based on the formal framework of reusable components, we suggest the following method for the systematic reuse of available components. Thereby the signature of a reusable component RC is defined as the signature of the root (and hence of all nodes) of RC and denoted by *sig(RC)*.

Assume that we want to construct an implementation for a given goal specification SPg by reusing existing implementations. Then, the process is carried out as follows:

1. Search a reusable component RC, the root of which can be adapted to SPg. We will concentrate here on the case where the adaptation can be performed by a matching which first forgets unnecessary sorts and operations of RC and then renames the remaining ones according to the signature of SPg. Such an adaptation is correct if the manipulated root specification of RC is an implementation of SPg. Formally; this means that we have to find a reusable component RC such that

   SPg $\leadsto$**rename** (**export** Σ **from** root(RC)) **by** ρ

   is an implementation relation for some subsignature $\Sigma \subseteq$ sig(RC) and some renaming ρ: $\Sigma \rightarrow$ sig(SPg). [1]

2. Select a leaf, SP-leaf, of the reused component RC and apply the matching operations to SP-leaf, i.e. construct the specification

   rename (**export** Σ **from** SP-leaf) **by** ρ

[1] We have assumed here that the specification SPg is not too large since otherwise a matching may be difficult or even impossible. If we want to deal with large goal specifications SPg then first an appropriate decomposition is necessary such that we can apply our method not to the whole goal specification but to parts of it. Since algebraic specification languages like ASL, ACT TWO [5], PLUSS [3] and others provide facilities for modular structuring of specifications decompositions can be obtained following the structure of a modular specification.

Due to the monotonicity of the specification building operators and the transitivity of the implementation relation, the correctness of the matching implies that we have constructed an implementation of SPg, i.e.

SPg $\rightsquigarrow$**rename** (**export** Σ **from** SP-leaf)
by ρ

Of course, the essential problem of our method is how to find appropriate reusable components whose root can be matched with the goal specification SPg. If we restrict ourselves to matching of signatures this problem can be formulated as follows:

Given: A goal specification SPg and a software library with reusable components $RC1, \ldots, RCn$

To find: All $i \in \{1, \ldots, n\}$ with

sig(SPg) = **rename** (**export** Σ **from** sig(RCi)) **by** ρ

for some subsignature $\Sigma \subseteq$ sig(RCi) and some renaming ρ: $\Sigma \rightarrow$ sig(SPg).
Matching of signatures will be implemented with the help of an object-oriented database. Our strategy is:

1. Represent the signature of all reusable components RCi as *objects* of the database.
2. Represent the signature of the goal specification SPg as a *query* to the database.
3. All objects which are retrieved by the system as an answer to the query represent exactly those signatures (of reusable components) which can be matched with the signature of SPg.

After successful matching of signatures it remains to prove that the selected component RC satisfies the semantic requirements of the goal specification SPg, i.e. Mod(**rename** (**export** Σ **from** root(RC)) **by** ρ) $\subseteq$ Mod(SP). For this purpose, other techniques involving theorem proving are necessary. In Section 5 we give an outlook for a software development system which integrates retrieval of reusable components on the basis of signature matching and interactive checks of semantical requirements.

Note that in the above discussion we have assumed that implementations are specifications as well and hence, by definition, have a signature. However, for our retrieval concept the particular form of the implementations is not relevant, i.e. implementations can be arbitrary program modules, as long as we can assign an algebraic signature to such a module. For instance, if an implementation is given by a functional module written in Standard ML, then the SML system computes automatically the signature of the module. Similarly, it should not be a difficult task to provide automatic computations of signatures for modules written in usual (typed) programming languages where programming units consist of a set of procedure- and/or function declarations (like ADA packages, Eiffel classes etc.). In this case, the necessary information for computing the signature of a module is recorded by the procedure- and/or function headings.

4 Representation and Retrieval of Reusable Components in the Database

4.1 Database and Data Modeling Language

The data modeling language used here is Telos (cf. [12]) which is supported by the knowledge base management system ConceptBase (cf. [9]).

In Telos a *class* is a set of objects which are classified by a set of attributes. The members of a class C are called *instances* of C. Classes itself can be instances of other classes thus obtaining a class hierarchy. On the lowest level, there are the so-called *tokens*, which are particular classes having no instances. For example, the number 0 is a token which is an instance of the class Integer.

Between classes several relationships can be defined, on the one hand by user-defined attributes and, on the other hand, by the predefined relations *"isA"* and *"in"*. The relationship "x in C" expresses that an object x is an instance of a class C. The isA-relation is the usual inheritance relation of object oriented systems. For example "C1 isA C" expresses that the class C1 is a specialization (subclass) of the class C. Then, all instances of C1 are instances of C as well.

An attribute of a class becomes an *attribute category* for each instance of the class. This means that an instance of a class may have several (even none) attributes for any attribute category whereby each attribute has exactly one value. For instance, let

```
Class C with
attribute
   attribute_of_C : Integer
end
```

be a class with one attribute, called `attribute_of_C`. Then `attribute_of_C` becomes an attribute category for each particular instance of `C`. For example, we can define a token `T` in `C` which has two attributes `attribute1_of_T`, `attribute2_of_T` with values, say `7` and `15`. Both attributes belong to the attribute category `attribute_of_C`.

```
Token T in C with
  attribute_of_C
      attribute1_of_T : 7
      attribute2_of_T : 15
end
```

Note that it is also possible to omit an explicit naming of attributes by giving only the values of (anonymous) attributes of the attribute category:

```
Token T1 in C with
  attribute_of_C : 7
                 : 15
end
```

In Telos, the so-called *query rules* are used to formulate queries to the database: A query is represented by a special class, called *query class*, whose instances are deduced according to a given query rule.

All rules are defined by formulas which are built over particular literals using the quantifiers "forall" and "exists". If a variable x ranges over all instances of a class C this is denoted by x/C. The two most important literals are of the form: `x.ac = y`, `x^a = y`. The first literal says that an object `x` has an object `y` as value for some attribute (of `x`) which belongs to the attribute category `ac`. For instance `T.attribute_of_C = 7` and `T.attribute_of_C = 15` are valid in the above example. The meaning of the second literal is similar, but in this case `a` is an explicitly named attribute of `x` which has the object `y` as value. For instance `T^attribute2_of_T = 15` holds.

We are now prepared to realize our strategy for the retrieval of reusable components as sketched at the end of Section 3.4. In the first step we will show how the signatures of reusable components can be translated into corresponding objects of the database and then we will provide a translation scheme for the representation of (signatures of) goal specifications as queries to the database.

4.2 Representation of Component Signatures

The signature of a reusable component consists of a set of sorts and a set of operations, each operation being equipped with a particular functionality. Hence component signatures will be represented by objects of a class `Component` which has two attributes, `sorts` and `opns`, the first one for the sorts and the second one for the operations of a component signature. Of course, sorts and operations are again instances of corresponding classes `Sort` and `Operation`. In order to store the functionality of operations the class `Operation` has three attributes `arity, domain` and `range`. The attribute `arity` determines the number of argument sorts, `domain` stores all argument sorts, and `range` determines the result sort of an operation.

```
Class Sort end

Class Operation with
attribute
  arity : Integer
  domain : Sort
  range : Sort
end

Class Component with
attribute
  sorts : Sort
  opns : Operation
end
```

Based on this class declarations we define an injective mapping *translate-comp* which maps signatures of reusable components to Telos objects: Let $\Sigma = (S, F)$ be the signature of a reusable component RC with sorts $S = \{s1,\ldots, sp\}$ and operations $F = \{f1,\ldots, fq\}$. Then the corresponding Telos object RC is an instance of the class `Component` which is constructed in the following way:

1. Declaration of sorts:

   For all $s \in S$ define

   ```
   Token s in Sort end
   ```

2. Declaration of operations:

   For all f ∈ F with functionality $s1, ..., sn \rightarrow s$ define

```
Token f in Operation with
  arity : n
  domain
    domain_1 : s1
       :
       :
    domain_n : sn
  range : s
end
```

   (Note that the attributes `domain_1, ..., domain_n` belong to the attribute category `domain`. The attributes of the attribute categories `arity` and `range` are anonymous since only their values `n` and `s` are provided.)

3. Declaration of the component:

   Define

```
Token RC in Component with
  sorts : s1
          :
          :
        : sp
  opns : f1
          :
          :
        : fq
end
```

As an example, we consider the reusable component RC_DEQUEUE = (DEQUEUE ⇝DEQUEUE_IMPL) which consists of the specification DEQUEUE from Figure 1 together with an implementation, called DEQUEUE_IMPL, which implements double-ended queues, for instance, by arrays with two pointers, one for the first element and one for the last element of a queue. The particular form of the implementation is not relevant here. Then the signature of RC_DEQUEUE is represented in Telos as shown in Figure 2.

4.3 Representation of Goal Specifications as Queries

Let us now assume that a set of reusable components is already present in the database and that we want to retrieve appropriate candidate components to be reused for the implementation of a given goal specification SPg. As already described in Section 3.4, our search criterion is to find all reusable components RC whose signature can be matched with the signature of SPg by first

```
Token Dequeue in Sort end

Token empty in Operation with
  arity: 0
  range: Dequeue
end

Token last in Operation with
  arity: 1
  domain
     domain_1 : Dequeue
  range: ElemOfDequeue
end

Token RC_DEQUEUE in Component with
  sorts: Dequeue
       : ElemOfDequeue
  opns : empty
       : first
       : last
       : insert
end
```

```
Token ElemOfDequeue in Sort end

Token first in Operation with
  arity: 1
  domain
    domain_1 : Dequeue
    range: ElemOfDequeue
end

Token insert in Operation with
  arity: 2
  domain
    domain_1 : ElemOfDequeue
    domain_2 : Dequeue
  range: Dequeue
end
```

Figure 2: Representation of double-ended queues in Telos

forgetting unnecessary sorts and operations of RC, and then renaming (some of) the remaining ones. In other words: we are interested in finding all reusable components, the signature of which comprises up to renaming the signature of the goal specification SPg. In order to make precise how this search intention can be transformed into a query to the database, we consider an example where the goal specification is the following specification LIST of lists over arbitrary elements:

```
spec LIST =
  sorts {List, ElemOfList}
  opns
   {empty: --> List
    first: List --> ElemOfList
    append: ElemOfList, List--> List}
  axioms
   {first(append(x, l)) = x}
```

```
QueryClass LIST isA Component with
query_rule:
 $forall m/LIST
  exists l,e/Sort f0,f1,f2/Operation
  ((m.sorts = l) and (m.sorts = e) and
   not(l = e) and
  (m.opns = f0) and (f0.arity = 0) and
                    (f0.range = l) and
  (m.opns = f1) and (f1.arity = 1) and
                   (f1^domain_1 = l) and
                   (f1.range = e) and
  (m.opns = f2) and (f2.arity = 2) and
                   (f2^domain_1 = e) and
                   (f2^domain_2 = l) and
                   (f2.range = l)) $
end
```

Figure 3: A Telos representation for LIST

Our query intention is then

To find: All instances of the class `Component` which represent a signature that can be matched with the signature of LIST.

The solution of this problem is to transform sig(LIST) into the following database query: *Find the subclass of `Component` whose objects have*

— *at least two different sorts l, e*

— *at least three operations f0, f1, f2 such that*

- *f0 has arity 0 and range l,*
- *f1 has arity 1, range e and domain_1 l*
- *f2 has arity 2, range l, domain_1 e and domain_2 l.*

Formally this query is expressed by the query class LIST given in Figure 3 where the query rule requires that all objects m of LIST have exactly the above properties.

After searching, the system will find the token RC_ DEQUEUE which represents one of the potential reusable components for the implementation of LIST. Note that it would be also possible to abstract from the particular ordering of the argument sorts and to allow permutations. For instance, instead of the conjunction `(f2^domain_1 = e)` and `(f2^domain_2 = l)` in the above query rule one could require more generally `((f2^domain_1 = e)` and `(f2^domain_2 = l))` or `((f2^domain_1 = l)` and `(f2^domain_2 = e))`.

```
QueryClass SPg isA Component with
query_rule:
 $forall m/SPg
  exists s1,...,sk/Sort
         f1,...,fr/Operation
  ((m.sorts = s1) and ... and
   (m.sorts = sk) and
  not(s1 = s2) and ... and
  not(s(k-1) = sk) and
  (m.opns = f1) and
  (f1.arity = ar(f1)) and
  (f1^domain_1 = asort(1,f1))
      and ... and
  (f1^domain_ar(f1) =
             asort(ar(f1),f1)) and
  (f1.range = rsort(f1)) and
               :
               :

   (m.opns = fr) and
   (fr.arity = ar(fr)) and
   (fr^domain_1 = asort(1,fr))
       and ... and
   (fr^domain_ar(fr) =
             asort(ar(fr),fr)) and
   (fr.range = rsort(fr)) and
   not(f1 = f2) and ... and
   not(f(r-1) = fr)$
end
```

Figure 4: A mapping of goal specifications into Telos

The above transformation can be generally defined by a mapping *translate-goal* which maps signatures of goal specifications into Telos query classes:

Let Σg = (Sg, Fg) be the signature of a goal specification SPg with sorts Sg = {s1, ..., sk} and operations Fg = {f1, ..., fr}. For any operation symbol f $\in$ Fg we denote the arity of f by `ar(f)`, the result sort of f by `rsort(f)` and the i-th argument sort of f by `asort(i,f)` (for i = 1, ..., ar(f)). Using this notation *translate-goal(Σg)* is defined as the query class shown in Figure 4. (Note that in the query rule `s1,...,sk` and `f1,...,fr` are not objects but variables ranging over all instances of the classes `Sort` and `Operation` respectively.)

¿From the definition of the mappings *translate-comp* and *translate-goal* follows that the transformation of a goal specification yields a query class which retrieves exactly those objects of the class `Component` which represent signatures of reusable components that can be matched with the signature of SPg. More precisely we obtain the following theorem:

Theorem
Let SPg be a goal specification. Then an object x is an instance of the class *translate-goal(sig(SPg))* if and only if

sig(SPg) = **rename** (**export** Σ **from** *translate-comp*$^{-1}$*(x)*) **by** ρ

for some subsignature $\Sigma \subseteq$ *translate-comp*$^{-1}$*(x)* and some renaming $\rho : \Sigma \to sig(SPg)$.

It is important to remark that we can even obtain explicitly the subsignature Σ and the renaming ρ (for which the matching is successful) if the system provides a deduction trace which shows all instantiations of the existential variables for which the query rule is satisfied.

5 Outlook

In the previous sections we have considered retrieval of reusable components on the basis of signature matching. But signatures provide no semantical information they only show us the sorts (types) of a component and the available (names of) functions together with their functionality. This corresponds to the information recorded by a deferred class in Eiffel (cf. [11]). Hence for the correct reuse of a component it remains to check that the semantical requirements of a goal specification SPg (expressed by the axioms of SPg) are satisfied by a component. Formally this means that the (modified) root specification of a component has to be an implementation (cf. Section 3.3) of SPg. For that purpose we can use the system ISAR (cf. [2]) which is an interactive system for algebraic implementation proofs. Hence we suggest the following architecture for a complete retrieval system based on ConceptBase and ISAR:

- ISAR allows to edit, to store and to prove the correctness of (two-level) reusable components of the form SP $\rightsquigarrow$SP-IMPL.
- The signatures of all reusable components are automatically translated into Telos objects and administrated by ConceptBase.
- Goal specifications are edited with the ISAR editor and translated into Telos query classes.
- The objects of a query class retrieved by ConceptBase refer to reusable components that can be syntactically matched with the goal specification.
- The semantic matching of goal specification and reusable component is checked by the ISAR system.

6 Concluding Remarks

We have described the use of an object-oriented database system for storing and retrieving reusable software components. The database management system supports a suitable data modeling language which allows an automatic mapping of component specifications and goal specifications into the data model. Since retrieval is based on the abstract specification of the behaviour of a component, our retrieval concept is independent from the particular form of component implementations which may be given by an algebraic specification (as in [8]), by a model of a specification (as in [7]) or by some concrete program (e.g. written in an imperative style as in [4]). In the latter case, an automatic computation of the signature of an implementation should be easily manageable for any implementation written in a typed programming language where programming units consist of a collection of procedure- and/or function declarations.

Our retrieval concept is based on formal signature matching. However, if the component library is large, many candidate components could be retrieved as an answer to a query (output-overload problem). Hence, it might be useful to supplement signature matching by classification of software components which can be realized by the subclass (inheritance) mechanism of Telos.

We did not consider here the storage of relations between single components like enrichment, refinement, generalization etc.. Of course, this would be a useful extension of our concept allowing to navigate through a component library according to the vertical and/or horizontal structuring properties of a component.

Acknowledgments

The authors would like to thank Martin Wirsing for valuable discussions and anonymous referees for helpful comments. This work is partially supported by the project SARET, by the BMFT-project KORSO, and by ESPRIT project NATURE.

References

[1] Allen, B. P. & Lee, S. D.: A Knowledge-Based Environment for the Development of Software Parts Composition Systems. In: *Proceedings of the 11th ICSE,* Pittsburgh, pp.104-112, 1989.

[2] Bauer, B. & Hennicker, R.: ISAR: An Interactive System for Algebraic Implementation Proofs. In: A. Voronkov (ed.): *Proc. of the International Conference LPAR '92*, St. Petersburg, Springer Lecture Notes in Artificial Intelligence 624, pp. 451-453, 1992.

[3] Bidoit, M.: Pluss, a language for the development of modular algebraic specifications. These dBEtat, Univ. Paris-Sud, Orsay, 1989.

[4] Cramer, J. & Fey, W. & Goedicke, M. & Große-Rhode, M.: Towards a Formally Based Component Description Language - A Foundation for Reuse. *Structured Programming*, Vol. 2, No. 12, Springer Verlag, 1991.

[5] Ehrig, H. & Mahr, B.: Fundamentals of algebraic specification 2. EATCS Monographs on Theor. Comp. Science, Vol. 21, Springer Verlag, 1990

[6] Frakes, W. B. & Nejmeh, B. A.: Software Reuse through Information Retrieval. In: *Proceedings of the 20th Annual HICSS,* Kona, HI, pp. 530-535, 1987.

[7] Gaudel, M. C. & Moineau, T.: A Theory of Software Reusability. In: H. Ganzinger (ed.): *Proceedings ESOP '88*, Springer Lecture Notes in Computer Science 300, pp. 115-130, 1988.

[8] Hennicker, R. & Wirsing, M.: Formal Aspects of Software Reusability. In: *Proceedings of the First International Workshop on Software Reusability*, Technical Report No. 57, Departement of Computer Science, University of Dortmund, 1991.

[9] Jarke M.: *ConceptBase V3.0 User Manual.* Jarke, M. (Ed.) University of Passau, 1991.

[10] Maarek, Y. S. : An Information Retrieval Approach for Building Reuse Libraries. In: *Proceedings of the First International Workshop on Software Reusability*, Techn. Rep. 57, Depart. of Computer Science, University of Dortmund, 1991.

[11] Meyer, B.: Object-Oriented Software Construction. New York, Prentice Hall, 1988.

[12] Mylopoulos, J. & Borgida, A. & Jarke, M. & Kourbakis, M.: Telos: A Language for Representing Knowledge about Information Systems. *ACM Transactions on Information Systems*, Vol. 8, No. 4, 1990.

[13] Prieto-Diaz, R.: *A Software Classification Scheme.* Ph.D.-Thesis, Univ. of California, Irving, 1985 .

[14] Wirsing, M.: Structured Algebraic Specification: A Kernel Language. *Theoretical Computer Science* 42, pp.123-249, 1986.

[15] Wood, M. & Sommerville, I. : An Information Retrieval System for Software Components. *IEE/BCS, Software Engineering Journal,* pp.198-207, Sep. 1988.