Storing and Retrieving Software Components: A Refinement Based System

Rym Mili, Member, IEEE, Ali Mili, Member, IEEE, and Roland T. Mittermeir, Member, IEEE

Abstract—Software libraries are repositories which contain software components; as such, they represent a precious resource for the software engineer. As software libraries grow in size, it becomes increasingly difficult to maintain adequate precision and recall with informal retrieval algorithms. In this paper, we discuss the design and implementation of a storage and retrieval structure for software components that is based on formal specifications and on the refinement ordering between specifications.

Index Terms—Software libraries, software reuse, software components storage and retrieval, formal specifications, refinement ordering.

1 INTRODUCTION—SOFTWARE LIBRARIES

Software libraries are repositories where software components are stored and searched; as such, they represent a precious resource for the software engineer. Moorman Zaremski and Wing [66], [67] identify several uses for such an asset: An engineer can study library components to become familiar with a programming language or a programming style, look for common patterns of usage, get acquainted with an application domain, or reuse library components (rather than have to write them from scratch) [1], [4], [18], [52]. While software reuse is the most common application of software libraries, it is not necessarily the only one.

From a user’s (software engineer’s) viewpoint, the most salient feature of a software library is the effectiveness of the library’s retrieval algorithm. The effectiveness of a retrieval algorithm is judged on the basis of three criteria: precision, which is the ratio of relevant components retrieved over the total number of retrieved components; recall, which is the ratio of relevant components retrieved over the total number of relevant components in the library; and response time, which we measure in abstract terms as the ratio of relevant components that are inspected in a retrieval operation over the size the library, as well as the average time complexity of a match. Retrieval algorithms are strongly dependent on the representation of the components in the library; they can be divided into four broad families [16]: AI-based algorithms [11], [40]; hypertext-based algorithms [15]; library science/information science algorithms [47], [50]; and formal specifications-based algorithms [59], [63], [44], [35], [66], [67]. In practice, library science/information science techniques appear to be most popular with organizations that practice software reuse [16]; this is further borne out by studies to the effect that the most successful reuse environments use simple representation methods [62].

Yet, as the size of software libraries increases, and as components grow increasingly complex and their semantic differences become finer and finer (for example, maintaining several versions of a software component), it becomes increasingly difficult to trust the retrieval task to informal procedures, such as those of library science or information science. Because software reuse is a recent phenomenon, organizations have only recently created software libraries; as these libraries grow in size, the selection of algorithms that maintain adequate precision and recall will be increasingly crucial. Some organizations already have large software libraries: Bell Northern Research maintains an online library of about 16,000,000 lines of code [8]; the Asset library is expected to hold many thousands of components [31]; and IBM maintains a library of more than 1,200 components. Further weakening the case for library science/information science-inspired algorithms is the fact that their success depends critically on human expertise and human experience [16]; in organizations with high staff turnover (for example, United States corporations, where the average tenure is three years), this may prove to be a serious issue.

In this paper, we discuss the design and implementation of a software library, based on a representation of software components by means of formal specifications. Our solution can be characterized by the following premises:

• Software components are represented in the software library by a formal specification that describes their functional properties. Because such specifications may be arbitrarily abstract, they allow us to focus the description on those properties of the component that are most relevant in a retrieval operation. This improves the chances of a match, as it recognizes the relationship between two components that have the same important properties, but differ in minor details. In this regard, our technique is similar to that of Perry and Popovich [44], in that both are based on logical descriptions of the semantics of the software compo-
ments at hand; whereas we are concerned exclusively with the storage and retrieval problem, Perry and Popovich deal also with the issues of conceptualization and use of software components.

- A crucial feature for efficient retrieval in traditional databases is the ordering between keys: when such a feature is available, it becomes possible to perform logarithmic search, and even linear search can be speeded up by a factor of two. There is no total ordering between specifications; however, the refinement ordering (which expresses that one specification captures more requirements information, and imposes stronger requirements, than another), which is partial, affords us means to improve the response time of our software library because it allows us to rule out (potentially) large portions of the library with a single match. In our implementation, this ordering is defined by means of a first order theorem that we submit to a theorem prover for automatic/on-line verification.

- Given a search argument (the specification for which we are seeking components) and a key to a stored component (its specification), we do not require that the key be identical to the search argument; rather, we consider that there is a match as soon as the key refines the argument. Then, by definition, any program that is known to be correct with respect to the stored key is correct with respect to the search argument and hence can be returned as a result of the retrieval operation. This weaker condition is sufficient for the purpose of correctness preservation, while dramatically enhancing the chances of a match; in other words, it is sufficient for the sake of precision, while at the same time it enhances recall.

- It may well happen that no component in the software library matches (i.e., refines) a given search argument, but that some components satisfy parts of the requirements it expresses. Then we wish to identify software components in the library that satisfy the largest portion of the requirements of the search argument. Topologically, this amounts to identifying keys that minimize some measure of distance from the search argument. We call this approximate retrieval. In practice, approximate retrieval yields software components that do not necessarily satisfy the search key, but may be economically adapted to produce a component that does.

In [66], Moorman Zaremski and Wing identify two kinds of software components: functions, which are defined by their input-output behaviour; and modules, which are defined by a data structure and procedures that operate on the data structure. For the sake of simplicity, we will focus our discussion on the storage and retrieval of functions; in Section 5.4 we will briefly discuss how our technique can be adapted to deal with modules.

In Section 2, we briefly present our relation-based specification model; then we discuss the ordering properties and the lattice-like properties of the refinement relation between specifications, which is used as the basis for structuring our software library. In Section 3, we discuss the storage structure of our software library; in particular, we discuss how components can be arranged according to the refinement ordering relation. In Section 4, we investigate how the proposed storage structure can be used to perform component retrieval: specifically, we discuss exact retrieval, whereby we seek components that satisfy requirements formulated in a user query; then approximate retrieval, whereby we seek components that are optimally close to the requirements formulated in a user query. In Section 5, we conclude by summarizing the main features of our library, evaluating it, comparing it to alternative storage organizations, and sketching directions for further research.

2 RELATIONAL SPECIFICATIONS

2.1 Specifying with Relations

For the purposes of this paper, we take a program specification to be a description of functional requirements that a program must satisfy. For the sake of discussion, we represent a specification by the pair \((S, R)\), where \(S\) is a set (called the space of the specification), and \(R\) is a relation on \(S\) (called the relation of the specification). The space of a specification is usually structured as the cartesian product of named elementary spaces, and represented by means of Pascal-like variable declarations. If we let space \(S\) be defined by means of variables \(a, b\) of type integer, for example, then we denote variable elements of \(S\) by lower case \(s\) and refer to the \(a\)– and \(b\)-components of \(s\) by \(a(s)\) and \(b(s)\). The relation of a specification contains all the input/output pairs that the specifier considers correct. Because the specifier may admit more than one output for a given input, the relations that we consider are not necessarily deterministic —indeed the ability to be arbitrarily nondeterministic in specifying requirements is a key feature of our model. Whenever the space \(S\) is implicit from the context, we represent specifications by relations.

Given a specification of the form \((S, R)\), we consider programs that manipulate the variables that define space \(S\); we say that these programs are defined on space \(S\). Given a program \(p\) on space \(S\), we are interested in the function that this program computes on \(S\); we call this function the functional abstraction of program \(p\), and we define it as:

\[
[p] = \{(s, s') \mid \text{if program } p \text{ starts execution in state } s \\
\text{then it terminates execution in state } s' \}
\]

Because a relation \(R\) on \(S\) is a subset of \(S \times S\), it is usually represented by a predicate on \(S \times S\), as in

\[
R = \{(s, s') \mid p(s, s')\}.
\]

Special relations on \(S\) include the universal relation, defined by \(L = S \times S\), the empty relation, defined by \(\emptyset = \{\}\), and the identity relation, defined by \(I = \{(s, s') \mid s' = s\}\). The domain of a relation \(R\) is the set defined by

\[
\text{dom}(R) = \{s \mid \exists s' : (s, s') \in R\}.
\]

A relation on \(S\) is said to be total if and only if its domain equals all of \(S\).

Among the operations on relations we mention: The complement of relation \(R\) is denoted by \(\overline{R}\) and defined by

\[
\overline{R} = L \setminus R = \{(s, s') \mid (s, s') \not\in R\}.
\]
The product of relation $R$ by relation $R'$ is denoted by $R \circ R'$ (or, $RR'$, for short) and defined by

$$R \circ R' = \{(s, s') | \exists t : (s, t) \in R \land (t, s') \in R'\}.$$  

In order to illustrate how relations can be used to represent specifications, we consider a simple user requirement, for which we discuss a number of distinct interpretations. Let the user requirements be:

“We wish to specify a square root program.”

We let the space of our specification be $S = \text{real}$ and we consider the following distinct interpretations:

- No negative arguments are submitted. The image may be negative or positive:
  $$R_0 = \{(s, s') | s \geq 0 \land s^2 = s\}.$$  

- No negative arguments are submitted. We are interested in the non-negative square root of the argument:
  $$R_1 = \{(s, s') | s \geq 0 \land s^2 = s \land s' \geq 0\}.$$  

- For non-negative arguments, the image is the positive or negative square root; for negative arguments, the image is immaterial.
  $$R_2 = \{(s, s') | s \geq 0 \land s^2 = s \} \cup \{(s, s') | s < 0\}.$$  

- For non-negative arguments, the image is the positive square root; for negative arguments, the image is immaterial.
  $$R_3 = \{(s, s') | s \geq 0 \land s^2 = s \land s' \geq 0\} \cup \{(s, s') | s < 0\}.$$  

- For non-negative arguments, the image is the positive square root; for negative arguments, the image is immaterial.
  $$R_4 = \{(s, s') | s \geq 0 \land s^2 = s \land s' \geq 0\} \cup \{(s, s') | s < 0 \land s' = 0\}.$$  

### 2.2 Refinement Ordering

We are interested in defining an ordering among specifications which reflects the strength of their requirements. We start by means of illustrative examples on space $S = \text{real}$.

1) We consider the following specifications:

$$R_0 = \{(s, s') | s - 1 \leq s' \leq s + 1\}$$  

$$R_1 = \{(s, s') | s - 2 \leq s' \leq s + 2\}.$$  

We consider the question: Which, of $R_0$ and $R_1$, defines a stronger requirement? The answer is $R_0$, because of a combination of the reasons mentioned in the examples above. Note that $R_0$ is neither a subset nor a superset of $R_1$.

2) We consider the following specifications:

$$R_0 = \{(s, s') | s - 1 \leq s' \leq s + 1\}$$  

$$R_1 = \{(s, s') | s \geq 0 \land s - 1 \leq s' \leq s + 1\}.$$  

We consider the question: which, of $R_0$ and $R_1$, defines a stronger requirement? The answer is $R_1$ because $R_0$ has a larger set of inputs to deal with than $R_1$ (all the real numbers, vs. non-negative real numbers). Note that $R_0$ is a superset of $R_1$.

3) We consider the following specifications:

$$R_0 = \{(s, s') | 2 \leq s' \leq s + 1\}$$  

$$R_1 = \{(s, s') | 0 \leq s' \leq s + 2\}.$$  

We consider the question: Which of $R_0$ and $R_1$, defines a stronger requirement? The answer is $R_1$, because the combination of the reasons mentioned in the examples above. Note that $R_0$ is neither a subset nor a superset of $R_1$.

4) We consider the following specifications:

$$R_0 = \{(s, s') | s \geq 0 \land s - 1 \leq s' \leq s + 1\}$$  

$$R_1 = \{(s, s') | s - 2 \leq s' \leq s + 2\}.$$  

We consider the question: Which of $R_0$ and $R_1$, defines a stronger requirement? In this case, $R_0$ and $R_1$ are not comparable because, while $R_1$ has a larger domain than $R_0$, $R_0$ is more specific than $R_1$ in assigning images to arguments.

The ordering relation which we have illustrated in these examples is known as the refinement ordering; its definition is given below.

**Definition 1.** Relation $R$ is said to refine (or be a refinement of) relation $R'$ if and only if $$RL \cap R'L \cap (R \cup R') = R'.$$

Note that $RL$ is the product of $R$ by the universal relation $L$; it can be written as $RL = \{(s, s') | s \in \text{dom}(R)\}$. We admit without proof that the refinement relation (which we abbreviate by: $R \sqsubseteq R'$) is antisymmetric and transitive; hence, it is a partial ordering. For the sake of illustration, we consider the specifications $R_0$, $R_1$, $R_2$, $R_3$, $R_4$ given in the previous section for the square root problem; Fig. 1 shows how these specifications are ordered by the refinement relation.

![Refinement relations between Sqrt specifications](image-url)  

**2.3 Refinement Lattice**

Given that the refinement ordering is a partial ordering relation, it is interesting to discuss whether it has lattice properties; we introduce its lattice properties by means of illustrative examples. We let the space be $S = \text{real}$.  

---

*This content is a direct representation of the readable text from the image provided.*
1) Given the following specifications:
\[ R_0 = \{(s, s') | s - 1 \leq s' \} \]
\[ R_1 = \{(s, s') | s' \leq s + 1 \}, \]
we are interested in deriving a specification that carries all the requirements information of \( R_0 \) and all the requirements information of \( R_1 \) and nothing more. We find:
\[ R = \{(s, s') | s - 1 \leq s' \leq s + 1 \}. \]
Note that \( R \) is the intersection of \( R_0 \) and \( R_1 \).

2) Given the following specifications:
\[ R_0 = \{(s, s') | s \geq 0 \land s - 1 \leq s' \leq s + 1 \} \]
\[ R_1 = \{(s, s') | s' < 0 \land s - 1 \leq s' \leq s + 1 \}, \]
we are interested in deriving a specification that carries all the requirements information of \( R_0 \) and all the requirements information of \( R_1 \) and nothing more. We find:
\[ R = \{(s, s') | s - 1 \leq s' \leq s + 1 \}. \]
Note that \( R \) is the union of \( R_0 \) and \( R_1 \).

3) Given the following specifications:
\[ R_0 = \{(s, s') | s \geq 0 \land s - 1 \leq s' \leq s + 1 \} \]
\[ R_1 = \{(s, s') | s' - 2 \leq s' \leq s + 2 \}, \]
we are interested in deriving a specification that carries all the requirements information of \( R_0 \) and all the requirements information of \( R_1 \) and nothing more. We find:
\[ R = \{(s, s') | s < 0 \land s - 2 \leq s' \leq s + 2 \} \cup \{(s, s') | s \geq 0 \land s - 1 \leq s' \leq s + 1 \}. \]
Note that \( R \) is neither the intersection nor the union of \( R_0 \) and \( R_1 \).

4) Given the following specifications:
\[ R_0 = \{(s, s') | s' \leq s + 1 \} \]
\[ R_1 = \{(s, s') | s + 2 \leq s' \}, \]
we are interested in deriving a specification that carries all the requirements information of \( R_0 \) and all the requirements information of \( R_1 \) and nothing more. We find that no such specification exists.

In the examples above, relation \( R \) reflects all the requirements of \( R_0 \) and all the requirements of \( R_1 \) and nothing more; as such, it represents the least upper bound of \( R_0 \) and \( R_1 \). As the last example illustrates, the least upper bound is not always defined. We give the following propositions without proof.

**Proposition 1.** Two relations \( R \) and \( R' \) have a least upper bound if and only if the following condition is satisfied: \( RL \cap R'L = (R \cap R')L \). Then, the least upper bound is defined by:
\[ R \cup R' = RL \cap R \cup R'L \cap R' \cup R \cap R'. \]
Whenever it is defined, the least upper bound (also called the join) of \( R \) and \( R' \) represents the sum of requirements information of \( R \) and \( R' \).

**Proposition 2.** Any two relations \( R \) and \( R' \) have a greatest lower bound, which is defined by:
\[ R \cap R' = RL \cap R'L \cap (R \cup R'). \]
The greatest lower bound (also called the meet) of \( R \) and \( R' \) represents the requirements information that is common to \( R \) and \( R' \), as we briefly illustrate below.

**Example.** We consider the space \( S = \text{integer} \) and we consider the following specifications on \( R \):
\[ R = \{(s, s') | s \geq -1 \land s' = s^2 \}. \]
\[ R' = \{(s, s') | s \leq 1 \land s' = s^2 \}. \]
we find
\[ RL = \{(s, s') | s \geq -1 \}. \]
\[ R'L = \{(s, s') | s \leq 1 \}. \]
where
\[ R \cap R' = \{(s, s') | -1 \leq s \leq 1 \land s' = s^2 \}. \]
This relation does indeed capture the information that is common to both \( R \) and \( R' \).

This lattice structure is used as a basis for ordering software components in our software library; this ordering is used, in turn, to orient the search of components in the library. Also, the interpretation of the greatest lower bound is used as a basis for our definition of approximate retrieval.

### 2.4 Refining Spaces
So far, we have studied an ordering on relations that are defined on the same space. It is possible to generalize this ordering by conjugating it with an ordering on spaces: hence we would say that \((S, R)\) is a refinement of \((S', R')\) if and only if \( S \) is a refinement of \( S' \) and the projection of \( R \) on \( S' \) is a refinement of \( R' \). This requires that we formally define two notions: the refinement between spaces and the projection of a relation, defined on space \( S \), onto some space \( S' \) which is refined by \( S \). A trivial instance of refinement between space \( S \) and space \( S' \) is when \( S \) is defined by all the variables of \( S' \) and has some additional variables of its own.

In this paper, we content ourselves with discussing refinement between relations that are defined on a common space; we will, in Section 5.4, discuss extensions of our work whereby we compare specifications that are defined on distinct spaces.

### 3 A Storage Structure

#### 3.1 Entry Structure
In any database application, an important decision in the database’s design process is that of defining a key, i.e., a unique identifier of a database entry. The most obvious choice for a key of a software component is the source code of the component; this, however, is not practical, because keys are typically involved in a wide range of operations (such as comparison for equality, for inequality, for ordering), and a lengthy detailed source code does not lend itself to these operations. A second possible option is to let the key be the function computed by the software component; as we illustrate in the example below, this too is a poor choice, for two reasons:
• The function of a software component typically contains a great deal of detail, hence does not lend itself to easy manipulation.
• Typically, much of the functional detail of a software component is incidental, and is not relevant from a user’s viewpoint.

Consequently, it is best to represent a software component with a two-field entry: one field contains a reference to the source code, and is used for retrieval purposes; the second field contains a specification of the component, that is arbitrarily abstract, and that captures the most important functional features of the component, and is used for matching purposes. Consider the example below.

**Example.** We let purposes. Consider the example below.

The term captures the most important functional features of the component, and is used for matching purposes. Consider the example below.

**Example.** We let $S$ be the space defined as follows:

```plaintext
a: array [1..N] of real;
x: real; k: integer;
```

and we consider the following procedure defined on space $S$:

```plaintext
procedure lns;
(linear search)
var
i: integer; {used as index}
begin
k := 0;
for i := 1 to N do
  if x = a[i] then k := i
end;
The functional abstraction of this procedure is captured in the following formula:

$$[\text{lns}] = \text{Found} \cap \text{Last} \cup \text{Not found},$$

where

$$\text{Found} = \{(s, s') | (\exists h : a(s)[h] = x(s)) \land a(s)[k(s')] = x(s) \land a(s') = a(s) \land x(s') = x(s)\}.$$  

$$\text{Last} = \{(s, s') | (\forall h : k(s') < h \leq N : a(s)[h] \neq x(s))\}.$$  

$$\text{Not found} = \{(s, s') | (\forall h : a(s)[h] \neq x(s)) \land a(s') = a(s) \land x(s') = x(s) \land k(s') = 0\}.$$  

The term $\text{Found}$ captures the property that when $x$ does appear in $a$, the procedure sets the final value of $k$ to one of the indices where $x$ appears (it may appear more than once), while preserving $a$ and $x$. Note that in the definition of procedure $\text{Found}$, the first conjunct is a logical consequence of the second, hence we may write this relation as:

$$\text{Found} = \{(s, s') | a(s)[k(s')] = x(s) \land a(s') = a(s) \land x(s') = x(s)\}.$$  

The term $\text{Last}$ expresses that the final value of $k$ is the largest index where $x$ appears in $a$. The term $\text{Not found}$ provides that if $x$ does not appear in $a$ then variable $k$ is set to zero. It is fair to consider that this represents too much information for such a simple procedure; more importantly, it is fair to consider that some of this detail is typically irrelevant from the user’s viewpoint. Upon storing this procedure into the software library, an analyst may decide e.g., that the fact that $k(s')$ is the largest index at which $x$ appears in $a$ is typically irrelevant; then we would store this component as the pair

$$\langle \text{lns}, R \rangle,$$

where

$$R = \text{Found} \cup \text{Not found}.$$  

Alternatively, the analyst may find that typically users do not invoke this procedure unless $x$ is known to be in $a$; then we would store this component as the pair

$$\langle \text{lns}, R' \rangle,$$

where

$$R' = \text{Found} \cap \text{Last}.$$  

A third alternative is to consider that the preservation of $x$ and $a$ is not an important functional detail, and to store this component as

$$\langle \text{lns}, R'' \rangle,$$

with

$$R'' = \text{Found}' \cap \text{Last} \cup \text{Not found'},$$  

where

$$\text{Found}' = \{(s, s') | a(s)[k(s')] = x(s)\}.$$  

$$\text{Not found}' = \{(s, s') | (\forall h : a(s)[h] \neq x(s)) \land k(s') = 0\}.$$  

Finally, yet another alternative is to consider that users are not typically interested in the exact location of $x$ in $a$—they only want to determine whether $x$ does or does not appear in $a$; then we would store this component as

$$\langle \text{lns}, R''' \rangle,$$

where

$$R''' = \{(s, s') | (\exists h : a(s)[h] = x(s)) \Leftrightarrow (k(s') \neq 0)\}.$$  

**Example.** As another example, consider the following Pascal procedure on the same space as above:

```plaintext
procedure bns;
(binary search)
var
low, mid, high: integer;
begin
k := 0; low := 1; high := N;
while (low <= high) and (k = 0) do begin
  mid := (low + high) div 2;
  if a[mid] = x then k := mid
  else if a[mid] < x then low := mid + 1
  else high := mid - 1
end;
```

Unlike procedure $\text{lns}$ (linear search) this procedure works only on sorted arrays. If we try to capture the functional abstraction of this procedure in all its detail, we run into a serious problem: If array $a$ contains
duplicate elements, and if \( x \) contains a value that is duplicated in \( a \), then it is very difficult to characterize the exact location that procedure \( bns \) returns in \( k \). Hence, at the outset it is tempting to forego details about which exact location is returned into \( k \), and to merely mention instead that \( k \) points to a location where \( x \) appears in \( a \). Then, we may consider storing component \( bns \) with the following description:

\[
\langle bns, B \rangle,
\]

where

\[
B = \text{Sorted} \cap (\text{Found} \cup \text{Not found}),
\]

with

\[
\text{Sorted} = \{(s, s') \mid \forall k : 1 \leq k < N : a(s)[k] \leq a(s)[k + 1] \}.
\]

Alternatively, if we consider that users typically use this procedure whenever \( x \) is known to be in \( a \) and are interested to locate an index where \( x \) appears in \( a \), then we may want to store this component as:

\[
\langle bns, B' \rangle,
\]

where

\[
B' = \text{Sorted} \cap \text{Found}.
\]

If we do not take the hypothesis that \( x \) is known to be in \( a \) but do not care to record that \( bns \) preserves \( a \) and \( x \), we may store this component as:

\[
\langle bns, B'' \rangle,
\]

with

\[
B'' = \text{Sorted} \cap (\text{Found}' \cup \text{Not found}').
\]

Finally, if we consider that users typically are not interested in finding where \( x \) is located but only want to know whether \( x \) is in \( a \) (by inspecting \( k \)) then we may store this component as:

\[
\langle bns, B''' \rangle,
\]

where

\[
B''' = \text{Sorted} \cap R'''.
\]

Hence, when we store a component, we can use its specification part to focus on those functional properties that we consider most relevant for our community of users, and abstract away the functional properties that we find less important or more cumbersome. There are ample incentives to represent components by means of abstract specifications that reflect their most important/relevant functional features:

- **Ease of representation.** The example of binary search is a good illustration of this aspect: the exact value that procedure \( bns \) places in \( k \) when \( x \) is duplicated in \( a \) is not only very difficult to characterize, it is also eminently irrelevant in general. Also, if we know ahead of time that the search procedures \( (ins, bns) \) are used to find whether \( x \) is in \( a \) without regard to its exact location, then it is advantageous to use specifications \( R'' \) and \( B''' \), which are quite simple.
- **Tractability.** Abstract specifications are not only simpler to generate (for the analyst whose task it is to store the components in the software library); they are also more tractable, in that they facilitate the computations that are performed on the component (storing the component in the library, matching it against user queries, evaluating its adequacy with respect to a user query, etc.).

- **Recall.** If a component’s description is too detailed, and contains irrelevant detail along with important relevant features of the component, the recall of the retrieval algorithms may be adversely affected: a component may well satisfy a user query in all its important functional features but fail to satisfy it on minor details. The retrieval algorithm, which cannot distinguish between important and unimportant detail, may well rule out that component, when it should have included it in the reply. By focusing a component’s description on the most important functional features of the component, we reduce the chances of this occurrence.

### 3.2 Ordering between Entries

It is quite common, in database applications, that storage keys are used not only to uniquely identify individual entries, but also to define a total ordering among entries; this ordering is used in turn to speed up retrieval algorithms. In our software library, we propose to order entries by the refinement ordering between their specification parts: in other words, we say that \( (c, R) \) is less than \( (c', R') \) if and only if \( R' \) is a refinement of \( R \).

For the sake of illustration, we consider the entries discussed above for procedures \( ins \) (linear search) and \( bns \) (binary search), and we show in Fig. 2 how these entries are ordered by our proposed relation. The entries are:

- \( (ins, R) \), where \( R = \text{Found} \cap \text{Last} \cup \text{Not found} \).
- \( (ins, R') \), where \( R' = \text{Found} \cap \text{Last} \).
- \( (ins, R'') \), where \( R'' = \text{Found}' \cup \text{Not found}' \).
- \( (ins, R'''') \), where \( R''''' = \{(s, s') \mid \forall h : a(s)[h] = x(s) \Rightarrow k(s') \neq 0 \} \).
- \( (bns, B) \), where \( B = \text{Sorted} \cap (\text{Found} \cup \text{Not found}) \).
- \( (bns, B') \), where \( B' = \text{Sorted} \cap \text{Found} \).
- \( (bns, B'') \), where \( B'' = \text{Sorted} \cap (\text{Found}' \cup \text{Not found}') \).
- \( (bns, B''') \), where \( B''' = \text{Sorted} \cap R''' \).

The reader can check the validity of the relations shown in this figure by referring to the definition of the refinement ordering (Definition 1).

### 3.3 Representation Structure

The representation structure that we propose is based on two premises; the first premise is given in the following definition.

**Definition 2.** A program \( p \) on space \( S \) is said to be correct with respect to a specification \( R \) on the same space if and only if the functional abstraction of \( p \) is a refinement of \( R \).

The second premise is given in the following proposition, and stems readily from the definition of correctness as well as the transitivity of the refinement ordering.

**Proposition 3.** If \( p \) is correct with respect to \( R \) and \( R \) is a refinement of \( R' \) then \( p \) is correct with respect to \( R' \).
Using this proposition, we adopt the following measure: instead of storing entries as pairs, we will order specifications by the refinement relationship and attach each program to the most refined specification with respect to which it is correct; hence several specifications may have no attached programs, but they are implicitly matched with all the programs encountered by following arcs upward in the refinement ordering. Fig. 3 shows what becomes of the storage structure of Fig. 2 when we adopt this measure.

3.4 Implementation

To implement this system, we create three directories: a directory that contains specifications; a directory that contains programs; and a directory where the system stores the results of queries (in the form of references to the components that it has retrieved). In the specifications directory (spe), each specification is represented by a file containing a formal definition of the specification (using first order logic) as well as references to other specifications and programs (namely the specifications that are linked to it by the refinement ordering as well as the programs that may be attached to it). In the programs directory (prg), each program is represented by a file that contains the following information: references to specifications to which the program is attached; an indication of the programming language of the program, as well as the text of the program’s source code, or a reference to a file containing the source code; optionally, a section that contains the program’s functional abstraction, written in first order logic.

In order to illustrate the proposed implementation, we consider a set of Pascal compilers, which map Pascal programs into various versions of P-code. We define three possible input spaces:

- **Simple**: the set of all Pascal programs that do not include records, pointers, and user-defined files.
- **Standard**: the set of all syntactically correct Pascal programs that are consistent with the ISO standard.
- **Full**: the set of all the strings that can be composed with Pascal symbols (including correct Pascal programs).

For each set defined above, we represent the predicate defining it with respect to a set containing an arbitrary sequence of characters by its name written in lower case letters (for example, predicate *simple* defines set *Simple*).

Likewise, we define three possible output spaces:

- **Reduced**: the set of all P-code programs that are restricted to eight registers ($r_0$ ... $r_7$) and do not include inc instructions (which are instead achieved by addition of 1 through the ALU).
- **Medium**: the set of all P-code programs that use sixteen registers ($r_0$ ... $r_{15}$) and do not include inc instructions.
- **Complete**: the set of all P-code programs.

From these sets we derive the predicates *reduced*, *medium*, and *complete* in the same manner as above. Further, we define two predicates that are applicable on P-code programs: *peephole*(y) means that program y is peephole optimized; and *globopt*(y) means that program y is globally optimized.

We let our software library contain the following 12 compilers:

- $c_0$: accepting simple Pascal programs, yielding complete P-code;
- $c_1$: accepting simple Pascal programs, yielding reduced P-code;
- $c_2$: accepting simple Pascal programs, yielding peepholed medium P-code;
- $c_3$: accepting standard Pascal programs, yielding complete P-code;
- $c_4$: accepting standard Pascal programs, yielding peepholed medium P-code if the input is in simple Pascal, and reporting an unavailable feature otherwise;
- $c_5$: accepting any string of Pascal terminal symbols, yielding peepholed medium P-code if the input is in simple Pascal, reporting an unavailable feature if the input is in standard Pascal, and an error message otherwise;
- $c_6$: accepting standard Pascal, yielding reduced P-code;
- $c_7$: accepting simple Pascal, yielding peepholed reduced P-code;
- $c_8$: accepting standard Pascal, yielding peepholed medium P-code as output;
Given a user query under the form of a specification, say \( K \), a component \( c \) is considered to be an exact match for \( K \) if and only if \([c]\) is a refinement of \( K \). Because of the storage structure we have adopted, we cannot compare \( K \) against the functional abstraction of \([c]\) (see the discussions of Section 3)—but we can compare \( K \) against the specification to which \( c \) is attached (which, in principle, captures the most relevant functional features of \( c \)).

Our retrieval algorithm proceeds by comparing \( K \) against the specification nodes of the database to identify those nodes that refine \( K \). Given that the network of specifications is structured in a lattice-like fashion, the specification nodes are visited in consecutive layers starting from the top, as long as they refine the search argument. The argument is first matched against the maximal nodes of the lattice to see if it refines them. Whenever it refines a node \( R \), that node is stored in a set of possible answers; whenever \( K \) is found to also refine a descendant of \( R \) then \( R \) is deleted.

Now, the file p4.prg:

```plaintext
%program%  c4.prg
%refined by%  r2.spe
%refined by%  r5.spe
%refined by%  r10.spe
%logical formula%
  (simple(x)& correct(x, y)
  & peephole(y) & medium(y)) |
  (standard(x)&-simple(x)&unavailable(y))
%end%
```

Fig. 4. A database of Pascal compilers.

4 RETRIEVAL ALGORITHMS

We distinguish between two retrieval operations: exact retrieval, when a user submits a query under the form of a specification and seeks to find programs that are correct with respect to this specification; approximate retrieval, when a user submits a specification and seeks to find programs that can be modified to satisfy the specification with minimal effort. We discuss these operations in turn, below.

4.1 Exact Retrieval

4.1.1 Design

Given a user query under the form of a specification, say \( K \), a component \( c \) is considered to be an exact match for \( K \) if and only if \([c]\) is a refinement of \( K \). Because of the storage structure we have adopted, we cannot compare \( K \) against the functional abstraction of \([c]\) (see the discussions of Section 3)—but we can compare \( K \) against the specification to which \( c \) is attached (which, in principle, captures the most relevant functional features of \( c \)).

Our retrieval algorithm proceeds by comparing \( K \) against the specification nodes of the database to identify those nodes that refine \( K \). Given that the network of specifications is structured in a lattice-like fashion, the specification nodes are visited in consecutive layers starting from the top, as long as they refine the search argument. The argument is first matched against the maximal nodes of the lattice to see if it refines them. Whenever it refines a node \( R \), that node is stored in a set of possible answers; whenever \( K \) is found to also refine a descendant of \( R \) then \( R \) is deleted.
from the set and replaced by its descendant. Whenever it is
found that \( K \) refines all the elements of the answer set but
does not refine any of their descendants, the search termi-
nates. The answer set contains the minimal specifications
that refine the search argument. All the programs attached
to these specifications (either directly or by transitivity) are
correct solutions for the search.

4.1.2 Implementation

Matching of the search argument against a current specifi-
cation node is formulated as a first order theorem and
submitted to the Otter theorem prover. During the course of
an exact retrieval, our system generates a number of such
theorems, submits them to Otter, then looks up Otter’s out-
put file to determine whether the theorem has been proven.
It proceeds according to the outcome of the proof.

The general format of such theorems is the following:

\[
\begin{align*}
\text{% checks whether a refines b.} \\
\text{% set parameters of inference,} \\
\text{assign(max_mem,1500).} \\
\text{assign(max_seconds,360).} \\
\text{set(free_all_mem).} \\
\text{% set resolution strategy} \\
\text{set(hyper_res).} \\
\text{formula_list(usable).} \\
\text{% in this section we store domain knowledge.} \\
\text{End_of_list.} \\
\text{% formula_list(sos).} \\
\text{% definition of relations a and b:} \\
\text{% here we formulate the theorem that} \\
\text{% provides that A refines B.} \\
((\text{domclause}\&\text{imageclause}) \leftrightarrow \text{refines}). \quad %1 \\
(\text{all x ((exists y A(x, y)) \leftrightarrow \text{doma}(x))}). \quad %2 \\
(\text{all x ((exists y B(x, y)) \leftrightarrow \text{domb}(x))}). \quad %3 \\
((\text{all x (domb(x) \rightarrow \text{doma}(x))}) \leftrightarrow \text{domclause}).\quad %4 \\
((\text{all x all y ((domb(x)\&A(x, y)) \rightarrow B(x, y)))}) \leftrightarrow \text{imageclause}). \quad %5 \\
% goal clause: does A refine B?
- \text{refines.} \\
\text{end_of_list.}
\end{align*}
\]

In order to justify the definition of this theorem, consider
the following proposition, which we admit without proof: 2
A relation \( A \) refines a relation \( B \) if and only if the following condition holds:

\[
dom(B) \subseteq \dom(A) \land (\forall s, s': s \in \dom(B) \land (s, s') \in A \Rightarrow (s, s') \in B).
\]

If we let \( \text{domclause} \) and \( \text{imageclause} \) represent
(respectively) the first and second conjuncts of this condition,
then it is clear that the file given above is merely the
transcription in first order logic of these conjuncts: Indeed,
line 1 defines predicate \( \text{refines} \) as the conjunction of
predicates \( \text{domclause} \) and \( \text{imageclause} \); lines 2 and 3
define predicates \( \text{doma}(x) \) and \( \text{domb}(x) \), which represent
clauses \( x \in \dom(A) \) and \( x \in \dom(B) \); line 4 defines predicate
\( \text{domclause} \) as the property that the domain of \( B \) is a subset of
the domain of \( A \); and lines 5, 6 define predicate \( \text{image-
clause} \) as the following property:

\( (\forall s, s': s \in \dom(B) \land (s, s') \in A \Rightarrow (s, s') \in B) \).

4.1.3 Illustration

As illustration of exact retrieval, consider the following
query submitted to the library of Pascal compilers:

\[
\text{We are looking for a compiler that takes standard Pascal as input, and produces peepholed medium P-code on output if the input is in simple Pascal.}
\]

Note that the query does not specify what happens if the
input is in standard \( \text{Pascal} \) but not in simple \( \text{Pascal} \); pre-
sumably, this implies that the compiler may do anything in
such cases. On the basis of this query, we generate a file
containing the following information.

\[
\text{(simple(x) & correct(x, y) & peephole(y) & medium(y)) | (standard(x) & -simple(x))}
\]

In formulating theorems to be submitted to Otter, our sys-
tem places this code as the definition of relation \( B \). The
search proceeds as follows (follow on Fig. 4):

1) The search argument is matched against node \( R_{11} \)
   and the system declares a success (\( R_{11} \) refines the
   key);
2) The search argument is matched against node \( R_9 \),
   and the system declares a failure (\( R_9 \) does not refine the
   key);
3) The search argument is matched against node \( R_{10} \),
   yielding success;
4) The search argument is matched against node \( R_5 \),
   yielding success;
5) The search argument is matched against node \( R_4 \),
   yielding success;
6) The search argument is matched against node \( R_2 \),
   yielding failure.
7) The search argument is matched against node \( R_7 \),
   yielding failure.
8) The program terminates and exits. Upon termination,
   the result directory contains specification \text{r4.spe} and
   its associated program \text{p4.prg}.

It may be instructive to consider, for example, the Otter file
in which we record the theorem that is generated by the
comparison of the search key with specification \( R_4 \):

\[
\text{assign(max_mem,1500).} \\
\text{assign(max_seconds,360).} \\
\text{set(hyper_res).} \\
\text{clear(print_given).} \\
\text{clear(print_kept).} \\
\text{clear(print_back_sub).} \\
\text{formula_list(usable).} \\
\text{% axiomatisation of the domain space} \\
\text{(all x (simple(x) \rightarrow standard(x))).} \\
\text{(all x (standard(x) \rightarrow full(x))).} \\
\text{% axiomatisation of the range space} \\
\text{(all y (reduced(y) \rightarrow medium(y))).} \\
\text{(all y (medium(y) \rightarrow complete(y))).} \\
\text{(all y (globopt(y) \rightarrow peephole(y))).} \\
\text{% output conditions do not reduce domain} \\
\text{(all x ((exists y correct(x, y)) \rightarrow (exists y (correct(x, y) \& globopt(y) reduced(y)))).}
\]

\text{End_of_list.}
formulas_list(sos).
% definition of relations a and b
(all x all y (A(x, y) <-> (simple(x) & correct(x, y) & peephole(y) & medium(y))))
| (standard(x) & & simple(x) & unavailable(y))))).
(all x all y (B(x, y) <-> (simple(x) & correct(x, y) & peephole(y) & medium(y))
| (standard(x) & & simple(x)))).
% code for refinement, as given above
% goal clause: does A refine B?
- refines.

end_of_list.

Particularly noteworthy about this description is how little domain knowledge is required. (See for example the clauses listed under formulas_list(usable)). The few clauses that are written under this section are all that is needed to carry out any proof that deals with the predicates of our domain (i.e., simple, standard, full, etc.). In fact, for any particular proof much less than this is required, but we include systematically all the knowledge for now.

4.2 Approximate Retrieval

4.3 Design

Given a user query under the form of a specification, say \( K \), a component \( c \) in a software library \( \sigma \) is considered to be an optimal approximate match if and only if it minimizes (among all the components of \( \sigma \)) the effort it takes to modify \( c \) to satisfy specification \( K \). Because we have no way to predict this modification effort by inspection of \( K \) and \( c \), we consider an alternative criterion: a component \( c \) is considered to be an optimal approximate match if and only if it maximizes (among all the components of \( \sigma \)) the functional information that it has in common with \( K \). Given a query \( K \), a candidate component \( c \) is said to maximize the meet with \( K \) if and only if:

\[
\forall d \in \sigma : K \cap d \sqsupseteq K \cap c \Rightarrow K \cap d = K \cap c.
\]

In other words, no element \( d \) of \( \sigma \) yields a greater meet with \( K \) than \( c \) does. By virtue of the discussions we had in Section 3 about the need to represent components with abstract specifications, we further refine this criterion: a component \( c \) is considered to be an optimal approximate match if and only if it is attached (directly or by transitivity) to a specification that maximizes (among all the components of \( \sigma \)) the functional information that it has in common with \( K \).

In order to make this criterion computable, we consider the lattice operator of meet, which we had interpreted as a measure of common information between two specifications; to illustrate this interpretation, we consider the following example.

Example. Let \( K \) be defined on some space \( S \) by the following formula:

\[
K = \{ (s, s') \mid a(s, s') \land b(s, s') \land c(s, s') \}
\]

and let \( R \) and \( R' \) be defined as follows:

\[
R = \{ (s, s') \mid a(s, s') \land d(s, s') \}
\]

\[
R' = \{ (s, s') \mid a(s, s') \land b(s, s') \}.
\]

Note that neither \( R \) nor \( R' \) is a refinement of \( K \); hence exact retrieval with key \( K \) fails to select \( R \) or \( R' \). Note also that \( R \) and \( R' \) are not comparable by the refinement ordering, hence neither is trivially known to be a better solution than the other. We can observe that \( R' \) has more in common with \( K \) (properties \( a(s, s') \) and \( b(s, s') \)) than does \( R \) (property \( a(s, s') \)), and we will show that the meet of \( K \) and \( R' \) is a refinement of the meet of \( R \) and \( K \). For the sake of simplicity, we assume that all three relations are total. First, we compute \( G = R \cap K \).

\[
G = G \cap K
\]

\[
= \{ \text{definition of } G \}
\]

\[
= \{ \text{formula of meet} \}
\]

\[
(R \circ L) \cap (K \circ L) \cap (R \cup K)
\]

\[
= \{ \text{and } K \text{ and } L \text{ total}, hence } R \circ L = L, K \circ L = L \}
\]

\[
L \cap (R \cup K)
\]

\[
= \{ \text{substitution} \}
\]

\[
\{ (s, s') \mid a(s, s') \land (b(s, s') \land c(s, s') \lor d(s, s')) \}.
\]

Similarly, we compute \( G' = R' \cap K \).

\[
G' = \{ \text{substitution, analogy with development of } G \}
\]

\[
R' \cup K
\]

\[
= \{ \text{because } K \subseteq R' \}
\]

\[
R' = \{ \text{substitution} \}
\]

\[
\{ (s, s') \mid a(s, s') \land b(s, s') \land c(s, s') \}.
\]

Relations \( G \) and \( G' \) have the same domain (both are total); on the other hand, \( G' \) is a subset of \( G \). Hence \( G' \) is a refinement of \( G \).

This example illustrates how, by comparing \( G = K \cap R \) and \( G' = K \cap R' \), we could tell that \( R' \) has more information in common with \( K \) than \( R \) does. Given that the meet of a specification \( R \) in the software library with key \( K \) measures the amount of common information with \( K \), we further refine our criterion of approximate match, to obtain the following definition.

Definition 3. A component \( c \) is considered to be an optimal approximate match for a search key \( K \) if and only if it is attached (directly or by transitivity) to a specification \( R \) that maximizes (among all the components of \( \sigma \)) the meet of \( R \) with \( K \) (i.e., \( R \cap K \)).

For the sake of precision of our approximate retrieval procedure, we need to ensure that we only select components that satisfy the criterion formulated above. On the other hand, for the sake of recall, we need to select all (or as many as possible) of the components that satisfy this criterion. In order to ensure maximum recall, we must identify the minimal (lowest, in the storage structure—see for example Fig. 4) specifications \( R \) that maximize the meet \( K \cap R \), then return all the components that are attached to them (above them in the storage structure).
4.3.1 Implementation

The brute force approach to implement the procedure of approximate retrieval is to scan all the specifications of the storage structure, select those that maximize \( R \cap K \), then select among them those that are minimal. This option is obviously unattractive because it does not take advantage of the built-in structure of the software library. To take advantage of this structuring, consider the following property:

- Either \( R \cap K \) is the same as \( R' \cap K \); this occurs whenever the difference between the functional properties of \( R \) and the functional properties of \( R' \) is totally orthogonal to \( K \). As far as satisfying \( K \) is concerned, \( R \) is as good as \( R' \).
- Or \( R \cap K \) strictly refines \( R' \cap K \); this occurs when \( R \) has more properties in common with \( K \) than does \( R' \).

Hence, if we apply the meet operation with \( K \) to all the nodes of the library, we obtain a collapsed version of its refinement graph. In this collapsed graph, some nodes distinguishable in the original graph (see Fig. 4) have collapsed into a single node and can hence no longer be distinguished (see Fig. 5). To perform approximate retrieval with the argument \( K \), our system starts from the maximal nodes in the refinement graph of the library and computes their meet with \( K \); then it starts prospecting further and further down the graph as long as the nodes it encounters collapse with the maximal nodes. It returns the set of minimal nodes that collapse.

To compute the meet of a specification \( R \) with the search argument \( K \) our system generates the following Otter definition:

\[
\begin{align*}
&\text{% defines meet of current node } R \text{ with arg. } K \\
&\text{% defining } R \\
&\text{(all } x, y \text{ } (r(x, y) \leftrightarrow \text{(definition of } R, \text{ taken from } R.spe)))). \\
&\text{% defining } K \\
&\text{(all } x, y \text{ } (k(x, y) \leftrightarrow \text{(definition of } K, \text{ taken from } key.spe))). \\
&\text{% defining meet} \\
&\text{(all } x \text{ } ((\exists y \ y(x, y) \leftrightarrow domr(x))). \\
&\text{(all } x \text{ } ((\exists y \ y(x, y) \leftrightarrow domk(x))). \\
&\text{(all } x, y \text{ } (\text{meet}(x, y) \leftrightarrow \text{domr}(x) \& \text{domk}(x) \& (r(x, y) \mid k(x, y))).)
\end{align*}
\]

If \( \text{max}(x, y) \) is the relation obtained by taking the meet of \( K \) with a maximal specification \( M \), and given that \( R \) has been reached from specification \( M \), then we generate the following theorem and submit it to Otter:

\[
\begin{align*}
&\text{formula_list(usable).} \\
&\text{(here comes definition of meet,} \\
&\text{taken from file above)} \\
&\text{(here comes definition of max,} \\
&\text{taken from specialized file)} \\
&\text{formula_list(sos).} \\
&\text{% defining collapse} \\
&\text{((all } x, y \text{ } (\text{meet}(x, y) \leftrightarrow \text{max}(x, y)))} \\
&\text{\quad \leftrightarrow \text{collapse).}}
\end{align*}
\]

% does \( R \) collapse with \( M \)
- collapse.

If the proof is established, we include \( R \) in the answer set and delete its immediate ancestor; else we check other specifications in the answer set whose descendants have not all been checked.

4.3.2 Illustration

To illustrate this operation, we consider the following query submitted to the library of Pascal compilers:

We are looking for a compiler that accepts any string of Pascal terminal symbols and returns medium P-code if the input is in standard Pascal, or an error message otherwise.

The search key that this query defines is

\[
K = \{ (x, y) \mid \text{standard}(x) \land \text{correct}(x, y) \land \text{medium}(y) \}
\]

\[
\cup \{ (x, y) \mid \text{full}(x) \land \text{standard}(x) \land y = \text{incorrect} \}.
\]

Execution of the exact retrieval procedure with this search key fails to retrieve correct components, hence it is necessary to envisage approximate retrieval. Execution of the approximate retrieval operation proceeds as follows:

- Specifications \( R_{11} \) and \( R_9 \) are put into the answer set.
- Specification \( R_9 \) collapses with \( R_{11} \); hence \( R_5 \) is included in the answer set.
- Specification \( R_{10} \) does not collapse with \( R_{11} \). Because all the descendants of \( R_{11} \) have been considered, and (at least) one has been included in the answer set, \( R_{11} \) is excluded from the answer set.
- Specification \( R_6 \) collapses with \( R_7 \); hence \( R_6 \) is included.
- Specification \( R_8 \) collapses with \( R_9 \) and is included.
- Specification \( R_7 \) does not collapse with \( R_9 \).
- Specification \( R_3 \) does not collapse with \( R_9 \). Because all the descendants of \( R_9 \) have been considered, and (at least) one has been included in the answer set, \( R_9 \) is excluded from the answer set.

4. This file is a straightforward transcription in first order logic of the relational formula of meet given in the definition of the meet operation; not that \( AL \cap B \) computes the restriction of \( B \) to the domain of \( A \).
5 CONCLUSION

5.1 Summary
In this paper, we have discussed the design and implementation of an automated system for the storage and retrieval of software components in a software library. To this effect, we have defined an ordering between specifications of software components, and shown how this ordering can be used as the basis for organizing the storage structure of the library. Then we have shown how this storage structure affords us means to perform component retrievals in an efficient manner—by inspecting only relevant components in the library and ignoring irrelevant components (if it were not for this structuring, we would have to inspect all the components whenever we perform a retrieval). Also, we have discussed how, whenever no component is found to satisfy a query, we can identify those components that come closest to satisfying it. Finally, we have investigated a prototype that implements this software library, and have used it to illustrate library operations.

5.2 Comparison
Even though it is not the most crucial success factor in software reuse, the issue of component storage and retrieval has received widespread attention to date [2], [11], [26], [23], [33], [39], [40], [44], [45], [49], [53], [57], [58], [60], [63], [66], [67]. Part of the reason, perhaps, is the scientific interest of the problem, and the technical challenge inherent in its solution. Other reasons for the high profile of this issue include the fact that software libraries are useful for other purposes than software reuse [66], as well as the expectation that software reuse libraries of the future will be quite large [16], hence will require adequate storage and retrieval structures. We discuss below a selection of past approaches to this problem, and briefly compare them to our software library structure.

In [36], we present a survey of existing methods of storage and retrieval in software libraries. We classify these methods in six broad families: information retrieval methods, which apply traditional information retrieval technology to the specialized context of storage and retrieval of software components; descriptive methods, which apply controlled vocabulary and faceted classification techniques to software libraries; operational semantics methods, which exploit the operational properties of programs to perform classification and retrieval; denotational semantics methods, which rely on semantic definitions of software assets to represent library entries and queries, and to perform matches; topological methods, which perform retrieval by minimizing some measure of distance between the query and candidate library components; structural methods, which perform asset retrieval on the basis of the asset’s structure rather than its semantic properties. The method we have presented here can be classified as a denotational method, although its approximate retrieval can also be viewed as a topological method. It is more reasonable to view it as a denotational method, hence we compare it to other denotational methods.

Perry and Popovich [43], [44] introduce a prototype of a software library, where software assets (in the form of executable components) are represented by predicates that define their main functional features and interface characteristics. The software library, named Inquire, is based on a specification-based software development environment, named Inscape [43]. Perry and Popovich identify four aspects in the reuse of software systems from building blocks: conceptualization, retrieval, selection, and use; rather than focus on assisting the user with the retrieval aspect, the Inquire prototype aims at helping with conceptualization, thereby producing better selection and use—at the expense of some loss of retrieval precision.

Rittri [53] proposes a method for software component storage and retrieval that applies to modules written in a functional language and is based on signature matching. The matching criterion is defined using polymorphic type systems and provides independence of the order of components in a type. Rittri [54] improves the recall of his method by weakening the matching condition using type isomorphisms. Runciman and Toyn [57] propose a similar solution, which uses polymorphic type systems to define criteria of signature matching in the context of functional programming, and provides independence of the number of arguments; this latter feature is intended to preserve (avoid the loss of) recall in cases when the signatures differ by syntactic details. In [55] Rollins and Wing (1989) present a software library of ML components whose specifications are written in λ-Prolog following a Larch-like [22] two tiered approach; they use λ-Prolog’s inference capability to automate component retrieval.

Moineau and Gaudel [20], [38] introduce a theory of software reusability on the basis of algebraic specifications of software components (more specifically, components that implement abstract data types). Specifications are represented in an OBJ-like language (PLUSS), by describing their signature and axioms which define the interactions between their methods. By varying how two specifications can be considered equivalent, and how some specification can be enriched, they obtain a wide range of matching criteria between specifications. Their criteria make provisions for method renaming and for relaxed equivalence between specifications.

Cheng and Jeng [25] discuss an organization of a software library that is based on formal specifications of components and queries. Their software library is organized into a two-layered hierarchy by means of a clustering algorithm, where the top layer places together related software components; the purpose of the two-level hierarchy is to
apply a two step retrieval process, whereby the first step performs a coarse-grained search to identify a cluster, and the second step performs a fine-grained search on the selected cluster. Chang and Jeng extend their work in [26] by investigating matching criteria that attempt to minimize measures of distance between the query at hand and candidate library components, and they extend it in [27] by defining matching criteria between components, and between methods; these matching criteria make provisions for subtyping, variable renaming, and parameter permutation.

Steigerwald [60] discusses a system that was developed at the Naval Postgraduate School under the name Computer Aided Prototyping System (CAPS). The system is geared towards prototyping real-time embedded systems, and includes an execution support system, a syntax directed editor, a software base with an embedded rewrite system and an engineering database management system with an embedded design management system. Queries are formulated at an abstract level using a special purpose specification language augmented with OBJ and the search mechanism relies on syntactic and semantic criteria. The semantic step of the retrieval procedure uses a method called query by consistency, which relies on an OBJ representation of library components and exploits the formal semantics of OBJ to identify a match between the query and candidate components.

Moorman Zaremski and Wing [65], [66] discuss signature matching as a mechanism for retrieving software components from a software library. Queries and library components are represented by their signatures, and a hierarchy of matching criteria is defined and discussed. The basic matching condition provides for equality between the two signatures, modulo variable renaming and parameter ordering. A number of weaker matching conditions are obtained by relaxing the exact match: By replacing equality with a subtyping relationship, Moorman Zaremski and Wing obtain a generalized match (if the type of the component is more general than the type of the query) or a specialized match (if the type of the query is more general than the type of the component); also, by applying uncurrying and currying transformations to the query and the component, they can match functions that do not have the same number of parameters. All these relaxations can be combined to produce a wide range of matching criteria. Zaremski and Wing extend their work on signature matching by investigating specification matching in [67]. They represent queries and components by (precondition, postcondition) pairs, and define a general matching criterion of which they apply a wide selection of specializations.

Penix and Alexander [41] advocate a formal specification based, domain theory oriented, approach to software component retrieval; they argue that traditional specification matching solutions do not scale up easily, due to the unpredictability of theorem proving, and that a domain theory oriented approach minimizes the impact of theorem proving by creating a domain specific knowledge base. They extend their work in [42] by considering a variety of matching criteria and refining the definition of features. Like Mili et al. [7], [35], [30], Penix and Alexander make a clear distinction between exact retrieval and approximate retrieval (rather than to consider the former as a happy coincidence of the latter), and make separate provisions for them; also, like Moorman Zaremski and Wing [65], [66], [67], Penix and Alexander use a hierarchy of matching criteria and specify components in Larch.

Recognizing that most specification matching algorithms fail to perform satisfactorily in practice, due to the bottleneck of theorem proving, Fischer, Kievernagel, and Snelting [13] propose a stepwise filtering procedure that proceeds in three steps:

1) Signature matching,
2) Model checking,
3) Theorem proving.

The idea of this approach is, of course, to reduce the search space as the retrieval algorithm gets more and more complex (and less and less likely to converge for large spaces): The first steps attempt to reduce the search space without affecting recall; the later steps attempt to improve precision by applying strong matching criteria to the remaining pool of candidates. Fischer et al. apply their approach to a library of components specified in VDM: they find good retrieval precision but poor recall, due primarily to the strong signature matching criterion.

By contrast with all these methods, our method can be characterized by the following premises: Library components are represented by relational specifications, and so are queries submitted to the library; the library has a non-trivial storage structure, which is used to orient retrieval operations, thereby giving a (potentially) small scan ratio; the method distinguishes between two separate retrieval goals, which are exact retrieval and approximate retrieval; the method can be fully automated, and provides the assurance of perfect precision (although at the expense of some loss of recall).

5.3 Assessment
In this section, we wish to briefly evaluate our approach with respect to the three criteria we had set forth earlier, namely precision, recall, and response time.

Precision. By its very design, this technique ensures perfect precision: all the components that are retrieved by an exact retrieval operation are provably correct with respect to the query that is submitted.

Recall. Our technique has less than perfect recall, for the following reason: because we compare the search key against specification nodes rather than directly against components, it is conceivable that a search key K admit a component C as a refinement but is not refined by the specification R to which the component is attached. Then, the exact retrieval will fail to return C, even though C is correct with respect to K (hence is relevant). This is the price we pay for tractability, as it results from letting specification R be arbitrarily abstract.

Response Time. Unlike several other techniques that are based on semantic analysis of library components, our technique defines a semantically meaningful structuring of the software library. This structuring enables us to perform selective retrievals that do not necessarily require inspecting all the components of the library.

Overall, while we do not consider that our method is an integral solution to the problem of storage and retrieval in software libraries, we do feel that it offers a set of features
that make it worthy of investigation. In practice, no one solution offers the right combination of accuracy, tractability and usability: Most existing solutions are either too cumbersome to be usable or too inaccurate to be useful. As in most cases, the Law of Diminishing Returns advocates the use of a wide range of solutions, which, together, offer a reasonable balance between accuracy, tractability, and usability.

5.4 Extensions

In this section, we discuss some extensions that we envision for our software library; the first extension consists in generalizing the definition of refinement to include space refinement (whereas in this paper we compare specifications that are defined on the same space); the second extension consists in selecting which aspects of domain knowledge must be integrated into a theorem before it is submitted to the theorem prover (whereas in the example of this paper we have systematically included all the available domain knowledge); the third extension consists in investigating the storage and retrieval of software modules (whereas in this paper we have focused exclusively on the storage and retrieval of software functions). These extensions are discussed below.

5.4.1 Refining Spaces: Signature Matching

The software library structure that we propose in this paper is critically dependent on the definition of the refinement ordering. Given two specifications \((S, R)\) and \((S', R')\) on the same space, this ordering ranks them by comparing \(R\) and \(R'\); whenever two specifications have different spaces, they are incomparable. Hence, on a macroscopic level, the storage structure is organized into several substructures, one for each space, where components are comparable within a substructure but incomparable across substructures; see Fig. 6.

Clearly, the adequacy of our library structure depends critically on the abundance of ordering relationships between its specifications (abundance of arcs in the graph that represents the storage structure—see Fig. 4, for example). Hence, for example, when our exact retrieval procedure compares a search key \(K\) against a specification \(R\) in the library and finds that \(R\) does not refine \(K\), we conclude that no specification that is lower than \(R\) will refine \(K\)—and exclude from consideration all the specifications that are lower than \(R\). The more arcs there are in the graph of the storage structure, the more nodes can potentially be excluded with a single comparison. By generalizing our definition of refinement to include refinement between spaces, we generate refinement arcs across substructures (see Fig. 6), hence improve the response time of our software library.

The details of this ordering are beyond the scope of this paper. Suffice it to say that it is the combination of the refinement ordering between relations as it is presented in this paper with a refinement ordering between spaces (viewed as data type) similar to that defined in, for example, [66].

5.4.2 Controlling the Theorem Prover

In its current implementation, the software library includes systematically all its domain knowledge whenever it invokes Otter to prove a theorem. This is clearly inefficient, as many theorems may require a small portion of this knowledge, and theorem provers are prone to combinatorial explosion. An important extension of our prototype is to monitor the amount of knowledge that is included in each theorem.

5.4.3 Storing and Retrieving Modules

Zaremski and Wing [66] distinguish between two kinds of software components that one may want to store in a software library: *functions*, which define a mapping between an input data type and an output data type, and are implemented as procedures (in Pascal); *modules*, which define an internal state and operations to modify the state or report on it, and are implemented as classes (in C++), packages (in Ada) or clusters (in CLU). Our approach to dealing with modules is based on a very simple premise: we represent module specifications by means of relations. Consequently, the same storage structure that we have discussed in this paper for functions can be used for modules—with only minor adjustments. It is quite conceivable, in fact, that the same software library contain functions and modules—although they would in all likelihood form distinct substructures (see Fig. 6). The detailed discussion of how to represent modules with relations is beyond the scope of this paper (see for example [5]). We content ourselves in this section with a brief illustrative example.

Given that we must specify a stack of integers, for example, we define a set \(X\), called the input space, by the operations that may be submitted to the stack,

\[
X = \{ \text{init, push()}, \text{pop, top} \};
\]

then we define a set \(Y\), called the output space, by the union of all possible outputs that the module may return,

\[
Y = \text{integer} \cup \{ \text{error} \};
\]

then we let the stack be defined by a relation from the set of sequences of \(X\) to the set \(Y\). Examples of elements of this relation are given below:
closed form (as we have represented function specifications cause module specifications are not usually represented in
dont care. The design of our software library remains unchanged by the introduction of modules; the implementation does change, however, because module specifications are not usually represented in closed form (as we have represented function specifications throughout the paper).

ACKNOWLEDGMENT

This work was completed while Ali Mili and Rym Mili were with the University of Ottawa, Canada. It was partially funded by Canada’s Natural Sciences and Engineering Research Council.

REFERENCES


Rym Mili holds a PhD degree in computer science from the University of Ottawa, Canada, and a Doctorat de Spécialité in computer science from the University of Tunis, Tunisia. She is an assistant professor of computer science at the University of Texas at Dallas. Her research interests are in software reuse, formal specifications, and cleanroom software engineering.

Ali Mili earned the PhD degree in computer science at the University of Illinois and the Doctorat és-Sciences d’Etat at the University of Grenoble, France. He served on the faculty of the Computer Science Department at the University of Tunis from 1984 to 1991, and as department chair from 1987 to 1991; and he worked at the University of Ottawa from 1991 to 1997. On July 1, 1997, he joined West Virginia University, where he holds a faculty position and serves as senior scientist at the Institute for Software Improvement. His research interests are in software engineering, ranging from managerial to technical aspects.

Roland T. Mittermeir received a Mag soc oec degree in business administration from the Wirtschaftsuniversität in Vienna, Austria; a Dipl-ing degree in informatics from the Technische Universität Wien; and a PhD degree from the Wirtschaftsuniversität Vienna, Austria. Mittermeir is professor of informatics at the Universität Klagenfurt, Austria. His academic career started at the Technische Universität Wien. After holding positions at the University of Texas at Austin and the University of Maryland at College Park, he joined the Universität Klagenfurt in 1984, where he was founding chair of the Department of Informatics. His research interests are in various areas of software engineering, notably in software reuse, software re-engineering, software process modeling, and design. He is a member of various professional societies. His www home page is <http://www.ifi.unl_klu.ac.at/cgi_bin/staff_home?roland>.