Specification Matching for Software Reuse: A Foundation*

Jun-Jang Jeng
Department of EE & CS
The George Washington University
Washington, DC 20052
jjeng@seas.gwu.edu

Betty H. C. Cheng
Department of Computer Science
Michigan State University
East Lansing, Michigan 48824
chengb@cps.msu.edu

Abstract

Using formal specifications to represent software components facilitates the determination of reusability because they more precisely characterize the functionality of the software, and the well-defined syntax makes processing amenable to automation. We present specification matching as a method for classification, retrieval, and modification of reusable components. A software component is specified in terms of order-sorted predicate logic. For both components and methods, we consider not only exact match, but also relaxed match and logical match for performing specification matching over a library of reusable software components.

1 Introduction

Software reuse has been claimed to be a means for overcoming the software crisis [1, 2]. However, current techniques to represent and manage software component libraries are not sufficient. Information retrieval methods based on analyses of natural-language documentation have been proposed [3, 4] for constructing software libraries. Unfortunately, software components represented by natural-language may hinder the retrieval process due to the problems of ambiguity, incompleteness, and inconsistency inherent to natural language. All of the above mentioned problems can be minimized by using formal specifications to represent software components [5, 6, 7, 8, 9].

The major objectives of a reuse system are to classify the reusable components, to retrieve them from an existing library, and to modify the retrieved components to satisfy the query specification [10, 11, 12, 13, 14]. This paper presents *specification matching* as a method for recognizing and retrieving a set of reusable components that address the objectives of a reuse system. Specification matching is useful not only to retrieve components, but it can also be used to construct a software component hierarchy to aid the users in browsing a software library in a systematic way [11]. Determining the logical relationship between a query specification and a specification of an existing component can also facilitate the determination of necessary modifications to the existing component to satisfy the query specification [10, 12].

The remainder of this paper is organized as follows. Section 2 gives the foundation of this work: formal specifications based on order-sorted predicate logic. Section 3 presents the definition of *relaxed exact matching*, which is less restrictive than exact matching. Section 4 gives the definition and an example of *logical match*. Section 5 shows an application of the specification matching process that is to retrieve a set of components being logically related to a query specification. Section 6 presents related approaches to software component retrieval. Section 7 closes with a summary and briefly outlines future work.

2 Formal Specification

Order-sorted specification has been proven to be a useful tool for the description of partially defined functions and error handling in specifying abstract data types [15, 16]. Order-sorted predicate logic (OSPL) based on order-sorted specifications can be used to represent typed knowledge, where the sort hierarchy gives the relationships among different types. We use order-sorted predicate logic to specify software components. The relationship between two components, that is, the reusability of one component with respect to another, is based on the sort information and inference rules used for a logical subsumption test.

S is defined to be a *partially ordered set*, or *poset*,

*This work is supported in part by funding from the NSF grants CCR-9209873 and CCR-9407318.

SSR '95, Seattle, WA,USA

if there is a binary relation $\leq$ on $\mathcal{S}$ that is reflexive, transitive, and anti-symmetric; that is, $x \leq y$ and $y \leq x$ implies $x = y$. Every poset has an associated relation $<$, defined by $x < y$ if and only if $x \leq y$ and $x \neq y$, which is transitive and anti-reflexive $\neg(x < x)$. Following the terminology of lattice theory, s is the *minimal element* of $\mathcal{S}$ iff $s \in \mathcal{S}$ and $s \leq t$ for all $t \in \mathcal{S}$. And s is a *lower-bound* of $\mathcal{S}$ iff $s \leq t$ for all $t \in \mathcal{S}$, denoted by $lb(\mathcal{S})$. The greatest lower bound of $\mathcal{S}$ is denoted by $glb(\mathcal{S})$. The meaning of *maximal elements upper-bound* ($ub(\mathcal{S})$), and the least upper bound of $\mathcal{S}$ ($lub(\mathcal{S})$) can be defined similarly.

The members of $\mathcal{S}$ are called *sort symbols*. For $s_1, s_2 \in \mathcal{S}$, if $s_1 \leq s_2$ then we say that s_1 is a *subsort* of s_2. $\mathcal{S}^*$ denotes the set of all finite strings from $\mathcal{S}$, including the empty string ϵ. The equivalence closure of the relation $\leq$ is denoted by $\cong$. The equivalence classes of $\mathcal{S}$ modulo $\cong$ are called connected components of $\mathcal{S}$. The ordering $\leq$ is extended component wise to strings $s_1, s_2, ..., s_n \in \mathcal{S}^*$; so we have $s_1, s_2, ..., s_n \leq s'_1, s'_2, ..., s'_n$ if and only if $s \leq s'$, for $1 \leq i \leq n$.

An order-sorted predicate logic OSPL is a 4-tuple $(\mathcal{S}, \leq, \mathcal{F}, \mathcal{P})$, where $\mathcal{S}$ is a set of sorts, $\leq$ a partial ordering over S, $\mathcal{F}$ a family $\{F_{w,s} | w \in \mathcal{S}^*, s \in \mathcal{S}\}$ of sets of operator symbols, and $\mathcal{P} = \cup_{w \in \mathcal{S}^*} P_w$ a set of predicate symbols. A partially-ordered set $(\mathcal{S}, \leq)$ with a least element $\perp$ and a greatest element $\top$ is called a sort hierarchy. We require there to be no infinitely descending chains in $(\mathcal{S}, \leq)$. For each f, $f \in \mathcal{F}_{s_1,...,s_n,s}$, the *domain sorts* and *range sort* of f are $s_1, ..., s_n$ and s, respectively. For some term t, the sort of t is denoted by $[t]$. The sort of the ith argument of f, denoted $[f]_i$, is s_i provided $1 \leq i \leq n$. Given an OSPL $\Sigma = (\mathcal{S}, \leq, \mathcal{F}, \mathcal{P})$, a family of *variables* over Σ is an $\mathcal{S}$-indexed family V of variables $V = \{V_s | s \in \mathcal{S}\}$, where $V_\perp = \emptyset$.

The family of Σ-terms $\mathcal{T}_\Sigma(V)$ over Σ and V, called the set of well-sorted terms, is the least $\mathcal{S}$-indexed family of sets such that:

1. for a variable, $v \in \mathcal{T}_\Sigma(V)_s$ for every $v \in V_s$. $[v] = s$.
2. for a constant, $f \in \mathcal{T}_\Sigma(V)_{s_0}$ if $f :\rightarrow s_0$ for each $s_0 \in \mathcal{S}$.
3. for an operator with at least one argument, $f(q_1, ..., q_n) \in \mathcal{T}_\Sigma(V)_s$ if $[q_i] \leq [f]_i$ and $q_i \in \mathcal{T}_\Sigma(V)_{s_i}$ for every $i \in \{1, ..., n\}$.

Let $\mathcal{T}_\Sigma = \cup_{s \in \mathcal{S}} \mathcal{T}_\Sigma(V)_s$. A well-sorted ground term is a well-sorted term in which no variable occurs. The set of *expressions* in an OSPL $\Sigma = (S, \leq, \mathcal{F}, \mathcal{P})$, is denoted as $\mathcal{E}_\Sigma$. We can define the membership of $\mathcal{E}_\Sigma$ recursively as:

1. $\perp, \top \in \mathcal{E}_\Sigma$;
   the greatest and least elements are Σ-expressions.
2. if $\alpha \in \mathcal{E}_\Sigma$ then $\neg\alpha \in \mathcal{E}_\Sigma$;
   if α is a Σ-expression, then $\neg\alpha$ is also a Σ-expression.
3. if $\alpha, \beta \in \mathcal{E}_\Sigma$ then $(\alpha \wedge \beta, \alpha \vee \beta, \alpha \rightarrow \beta, \alpha \leftrightarrow \beta \in \mathcal{E}_\Sigma)$;
   if α and β are Σ-expressions, then $\alpha \wedge \beta$, $\alpha \vee \beta$, $\alpha \rightarrow \beta$, and $\alpha \leftrightarrow \beta$ are also Σ-expressions.
4. let $p \in P_{s_1,...,s_n}$, and if $[t_i] = s'_i$ and $s'_i \leq s_i$ for $1 \leq i \leq n$, then $p(t_1, ..., t_n) \in \mathcal{E}_\Sigma$;
   for a predicate $p(t_1, ..., t_n)$, if $p \in P_{s_1,...,s_n}$ and the sort of every argument of $p(t_1, ..., t_n)$ is a subsort of its corresponding sort in $P_{s_1,...,s_n}$, then $p(t_1, ..., t_n)$ is a Σ-expression.
5. if $x \in V$, $s \in S$, and $\alpha \in \mathcal{T}_\Sigma$ then $(\forall x : s :: \alpha) \in \mathcal{E}_\Sigma$;
   for a quantified expression, if the sort of its quantified variable is defined in the sort hierarchy and its body expression is a Σ-expression, then it is a Σ-expression.
6. if $x \in V$, $s \in S$, and $\alpha \in \mathcal{T}_\Sigma$ then $(\exists x : s :: \alpha) \in \mathcal{E}_\Sigma$;
   with conditions similar to those for (5).

The expressions in $\mathcal{E}_\Sigma$ are called Σ-expressions. A variable is free iff it is not bound. A expression is closed iff it does not contain free variables. A Σ-expression without variables is called a Σ-sentence. We assume the partial ordering $\leq_{pred}$ for the predicate set $\mathcal{P}$ is user-defined. The equivalence closure is denoted by $\cong_{pred}$.

In general, a software component can consist of requirements, design knowledge, code segments, or test plans. A component can be used as the vehicle for encapsulation and data hiding, and it also provides the basic unit of reusability. We define a component explicitly to be a user-defined type whose behavior is described by a formal specification. The skeleton of a component specification is shown as follows.

component component_name_identifier
{
    inherit: component_name_identifier*
    (**method**: method_name_identifier)*
};

The key word **inherit** indicates that the current component inherits the properties from the components of previously defined components. The specifications in the **method** section define the behavior of methods in this component. The format of a method specification is:

method method_name $((Var : DomainSort)^*) \rightarrow Var : RangeSort$
    requires pre-expression
    modifies variables
    ensures post-expression

The expressions used to specify a *method* of a given component, including *pre-expression* and *post-expression*, are based on order-sorted predicate logic (OSPL). For each method, the interface specifies both the domain sorts and the range sort. The **requires** clause describes

restrictions on the arguments, which defines how the method may be invoked. We interpret an omitted **requires** clause as equivalent to "**requires** true." The **ensures** clause places constraints on the behavior of the method. The **requires** and **ensures** clauses relate two states, the state when the method is called, which we call *precondition*, and the state when it terminates, which we call a *postcondition*. A **requires** clause only refers to the values in the precondition. An **ensures** clause may refer to values in the pre- and the postconditions. A **modifies** clause describes which variables can be changed. An omitted **modifies** clause is equivalent to the assertion **modifies nothing**, meaning no objects are allowed to change in value during the execution of the method. An example method specification for the component **realSet** is shown in Figure 1, which consists of a function prototype followed by a body specified in terms of pre- and postconditions. Figure 1 asserts that the method **create** is a constructor of this component, the method **insert** adds an element to a real set, the method **delete** deletes an element from a real set, and the method **member** tests whether an element belongs to some set. Here we use $\doteq$ to represent the equality relation among terms. Although equality is not defined in OSPL, the expressions containing $\doteq$ can always be transformed into pure OSPL expressions [17]. Variables decorated with a prime symbol represent the latest value of a given variable. Moreover, the variables in the expressions without quantifiers are assumed to be universally quantified, i.e., each expression in a component specification is a Σ-sentence.

component realSet
{

method create: $() \rightarrow \alpha : realSet$
requires $\neg existing(\alpha)$
ensures $\alpha \doteq \emptyset$

method insert: $(e : real, \alpha : realSet) \rightarrow \alpha' : realSet$
modifies α
ensures $\alpha' \doteq \alpha \cup \{e\} \wedge\ card(\alpha') \doteq card(\alpha) + 1$

method delete: $(e : real, \alpha : realSet) \rightarrow \alpha' : realSet$
requires $\neg empty(\alpha)$
modifies α
ensures $\alpha \doteq \alpha' \cup \{e\} \wedge\ card(\alpha) \doteq card(\alpha') + 1$

method $member(e : real, \alpha : realSet) \rightarrow \beta : bool$
requires $\neg empty(\alpha)$
ensures $\beta \doteq e \in \alpha$

}

Figure 1: Component specification for realSet.

3 Relaxed Exact Match

Specification matching is the process of determining whether a library component *matches* a query at the specification level. It can be assumed that the formal specification is either provided by the specifiers or derivable from code components [18, 19, 20]. As with other retrieval methods, we relax the matching process beyond *exact match*. We introduce two kinds of matches in this paper: *relaxed exact match*, a less restrictive version of exact match and *logical match*. *Relaxed exact match* is defined in this section, and *logical match* is discussed in Section 4. We use the set $Spec\_Match(q, \otimes, C)$ to define specification matching [21]. Let $Spec\_Match(q, \otimes, C) = \{c \in C : c \otimes q\}$, where q is a query specification, $\otimes$ is a boolean operator that defines whether c satisfies the match condition or not, and C is a component library. Let both the query and the library components be specified by an OSPL Σ. Then $\otimes$ is denoted by $\simeq_{\Sigma,comp}$ for relaxed exact match, and $\sqsubseteq_{\Sigma,comp}$ for logical match, where $\sqsubseteq_\Sigma$ denotes logical subsumption over some OSPL Σ.

Let $\Sigma = (\mathcal{S}, \leq, \mathcal{F}, \mathcal{P})$ be an OSPL.

Assume two terms $f_l(s_1, \ldots, s_k), f_q(t_1, \ldots, t_k) \in \mathcal{T}_\Sigma$, then $(f(s_1, \ldots, s_k) \simeq_{\Sigma,term} g(t_1, \ldots, t_k))$ if and only if the following conditions hold:

1. $[f] = [g]$, sorts of terms are the same, and
2. $(s'_1, \ldots, s'_k) = permute(s_1, \ldots, s_k)$ and $(t'_1, \ldots, t'_k) = permute(t_1, \ldots, t_k)$, where permute: Σ-term* $\rightarrow$ Σ-term* is a permutation function,
3. $[s'_i] = [t'_i]$ for $1 \leq i \leq k$, the corresponding arguments have the same sort.

Two terms match exactly if and only if their range sorts match and their domain sorts match after permutation of their arguments. If an operator is not commutative, then the permutation function is an identity function. Two predicates can be defined to be in the same equivalence class via "$\cong_{pred}$". Note that equality is a special case of predicate equivalence. Let $s_1, \ldots, s_k, t_1, \ldots, t_k$ be terms, and p, q be two predicates. We can define the relaxed exact match between p and q. $(p(s_1, \ldots, s_k) \simeq_{\Sigma,pred} q(t_1, \ldots, t_k))$ if and only if both $p \cong_{pred} q$ and $s_i \simeq_{\Sigma,term} t_i$ for $1 \leq i \leq k$.

Based on the previous two definitions, we can now define the relaxed match between expressions. Assume $(s, t \in \mathcal{S})$, $(E, E_1, E_2 \in \mathcal{E}_\Sigma)$, and $(x, y \in V)$. Let $\odot$ represent the predicate connectives: either $\wedge$ or $\vee$. Let Q be either $\forall$ or $\exists$. The relaxed exact match between two Σ-expressions, denoted by "$\simeq_{\Sigma,expr}$", is defined as follows.

1. $(E_1 \simeq_{\Sigma,expr} E_2)$ if $(E_1 \simeq_{\Sigma,pred} E_2)$.
2. $(\neg E_1 \simeq_{\Sigma,expr} \neg E_2)$ if and only if $(E_1 \simeq_{\Sigma,expr} E_2)$.
3. $(E \simeq_{\Sigma,expr} E_1 \odot E_2)$ if and only if $(E \simeq_{\Sigma,expr} E_1)$ or $(E \simeq_{\Sigma,expr} E_2)$.
4. $((E_1 \odot E_2) \simeq_{\Sigma,expr} E)$ if and only if $(E_1 \simeq_{\Sigma,expr} E)$ or $(E_2 \simeq_{\Sigma,expr} E)$.
5. $(Qx : s :: E_1) \simeq_{\Sigma,expr} (Qy : t :: E_2)$ if and only if ([s] = [t]) and $(E_1 \simeq_{\Sigma,expr} E_2)$.

Based on "$\simeq_{\Sigma,expr}$", the relaxed exact match between two methods, denoted by "$\simeq_{\Sigma,method}$", can be derived.

Definition 1 *(Relaxed exact match between methods) Let M_1, M_2 be two methods specified by Σ, then $M_1 \simeq_{\Sigma,method} M_2$ iff $(pre(M_1) \simeq_{\Sigma,expr} pre(M_2))$ and $(post(M_1) \simeq_{\Sigma,expr} post(M_2))$.*

The relaxed exact match between components, denoted by "$\simeq_{\Sigma,comp}$", can finally be defined by using the above definitions. Let *methods(C)* denote the set of the methods of some component C.

Definition 2 *(Relaxed match between components) Let $C_{library}$ be a library component and C_{query} be a query component, then $C_{library} \simeq_{\Sigma,comp} C_{library}$ if and only if there is a one-to-one and onto mapping from methods(C_{query}) to methods($\sigma C_{library}$) over the relation $\simeq_{\Sigma,method}$, where σ is a sequence of variable renamings of user-defined sorts.*

The onto mapping can be relaxed because in practice the users need not specify every method of a query component in order to find a matched library component. To see how relaxed exact match might be useful, consider an example where the library component is the specification in Figure 1. Suppose a user wants to locate a method that deletes an element with sort `real` from a set. The query can be written as:

component *aSet*
{
  method remove: $(P : aSet, q : real) \rightarrow P' : aSet$
    requires $\neg empty(P)$
    modifies P
    ensures $size(P) \doteq size(p') + 1 \wedge P \doteq P' \cup \{q\}$
}

where, *aSet* is a user-defined type. Let $size \simeq_{pred} card$ be previously defined by the specifiers. The query matches the method `delete` of component `realSet` (with the sort renaming [$aSet \mapsto realSet$]) due to the following conditions:

1. $P \simeq_{\Sigma,term} \alpha$ and $q \simeq_{\Sigma,term} e$
   p and q have the same sort.
2. $P' \cup \{q\} \simeq_{\Sigma,term} \alpha \cup \{e\}$
   from (1) and the definition of match between terms.
3. $size(P) \simeq_{\Sigma,pred} card(\alpha)$
   from (2) and the definition of match between expressions.
4. $(size(P) \doteq size(p') + 1 \wedge P \doteq P' \cup \{q\}) \simeq_{\Sigma,expr}$ $(\alpha \doteq \alpha' \cup \{e\} \wedge card(\alpha) \doteq card(\alpha') + 1)$
   from (2), (3) and the definition of match between expressions.
5. (`aSet : remove`) $\simeq_{\Sigma,method}$ (`realSet : delete`)
   from (4) and the definition of match between methods.

4 Logical Match

A relationship to formally define the reusability of one component with respect to another one is described in this section. Based on OSPL, a partial ordering called *generality* between components in the library is formally defined. This relationship facilitates recognition and retrieval of a reusable component from a reusable component library.

Chang and Lee's subsumption test algorithm [22] is used to decide the *subsumption* relationship between clauses, that is, whether clause A subsumes clause B, denoted by $B \sqsubseteq_{clause} A$, where a clause is a disjunction of literals. A set of clauses S is thought of as a conjunction of all clauses in S, where every variable in S is considered to be universally quantified. Figure 2 shows a subsumption test algorithm similar to the traditional subsumption test [22], but it is, instead, based on *order-sorted resolution* [23]. Let A and B be clauses. Let $\theta = \{x_1 \mapsto a_1, ..., x_n \mapsto a_n\}$, a set of variable renamings, where $x_1, ..., x_n$ are variables occurring in B, and $a_1, ..., a_n$ are new distinct constants not occurring in A or B. This algorithm tests whether or not A subsumes B.

The traditional resolution strategy invented by Robinson [24] computes the resolvents of two clauses that are described by *unsorted* predicate logic. Since OSPL is used as the underlying logic of our component specifications, an order-sorted resolution rule is required for the determination of the subsumption relationship between two clauses. Several order-sorted unification methods have been proposed recently [17, 23]. We apply their strategies in order to derive an order-sorted resolution rule as follows:

C_1: $L, K_1, ..., K_n$
C_2: $\neg L', M_1, ..., M_m \qquad L\theta = L'\theta$

---

resolvent: $K_1\theta, ..., K_n\theta, M_1\theta, ..., M_m\theta$.

Input: Two clauses A and B, which are formally specified by an **OSPL** Σ. Let $B = L_1 \vee L_2 \vee ...L_m$. Suppose $\sigma \in \mathcal{SUB}_\Sigma$ and σB is a ground term.
Output: $B \sqsubseteq_\Sigma A$?

1. Let $W = \{\neg\sigma L_1, \ldots, \neg\sigma L_m\}$.
2. Set $k = 0$ and $Z^0 = \{A\}$.
3. If Z^k contains 2, an empty clause, then return $(B \sqsubseteq_\Sigma A)$;
   Otherwise let $Z^{k+1} = \{$resolvents of C_1 and C_2 $\|$ $C_1 \in W$ and $C_2 \in Z^k$ $\}$.
4. If Z^{k+1} is an empty set, then return $(B \not\sqsubseteq_\Sigma A)$;
   Otherwise, set $k = k + 1$ and go to (3).

Figure 2: Subsumption test algorithm based on order-sorted resolution.

The order-sorted resolution rule involves an instantiation of the formulas by a Σ-substitution θ, which is a mapping of variables to terms. Application of this substitution must result in the same atoms for both resolution literals [25]. The Σ-substitution θ can be obtained by the algorithm in Figure 4. If two Σ-terms have a

Input: two Σ-terms x and t, where $x \in A$ and $t \in \sigma B$; and suppose we intend to test whether $(B \sqsubseteq_\Sigma A)$ or not for some **OSPL** Σ.
Onput: Σ-substitution θ between x and t.
Procedure:

1. **if** $x = t$ **then return** $(\emptyset)$;
2. **if** $x \in V(t)$ **then return** ("failed");
3. **if** $[t] \leq [x]$ or $[t] \leq_{pred} [x]$ **then return** $(\{x \mapsto t\})$;
4. **if** $glb([x],[t]) = \emptyset$ **then return** $(\emptyset)$;
5. $\{s_1, .., s_k\} = glb([x],[t])$;
6. let $\{y_1, ..., y_k\} \in V$ such that no y_i is used before and $[y_i] = s_i$ for $1 \leq i \leq k$;
7. **return** $\{\{x \mapsto y_1, t \mapsto y_1\}, ..., \{x \mapsto y_k, t \mapsto y_k\}\})$.

Figure 3: Finding Σ-substitution of variables and terms.

sort relationship, i.e., either $[t] \leq [x]$ or $[t] \leq_{pred} [x]$, then return the Σ-substitution $\{x \mapsto t\}$. Note that the *predicate partial ordering* "$\leq_{pred}$" is defined by the software specifiers. On the other hand, if x and t have no direct sort relationship, then a group that defines their greatest lower bound is determined in order to identify a set of Σ-substitutions.

Since the above algorithm returns a set of substitutions instead of a single one, as in the unsorted case, a recursive process is introduced to the subsumption test algorithm to iterate through the substitutions, which is not discussed in this paper [?]. The procedure of finding a Σ-substitution of variables and terms is applied to the subsumption test algorithm. Therefore, we do not consider the case that the term t could be a variable because each term in σB should be a ground term (see Figure 2), where σ is a set of Σ-substitutions. The more general algorithm for finding a Σ-substitution of variables and terms can be found in [23].

Hence, the subsumption relationship over the clauses of some OSPL Σ can be determined by the subsumption test algorithm based on an order-sorted resolution procedure. The following rules assert several properties of the generality relationship among expressions.

Let s, t be terms, f, f', g, g' be functions, p, p', q, q' be predicates, exp_1, exp_2, exp_3 be expressions, and $\otimes$ denote the predicate connectives. The function *permute* generates all possible permutation of an input list. For example,
permute([1,2,3])={[1,2,3],[1,3,2],[2,1,3],[2,3,1],[3,1,2],[3,2,1]}. The generality relationship "$\sqsubseteq_\Sigma$" is defined by the following rules:

1. $s \sqsubseteq_\Sigma t$ if and only if $[s] \leq [t]$.
2. $(s = t) \sqsubseteq_\Sigma (s' = t')$ if and only if $((s \sqsubseteq_\Sigma s' \wedge t \sqsubseteq_\Sigma t')$ or $(s \sqsubseteq_\Sigma t' \wedge t \sqsubseteq_\Sigma s'))$.
3. $([s_1, \ldots, s_n] \sqsubseteq_\Sigma [s'_1, \ldots, s'_n])$ if and only if $(\exists [t_1, \ldots, t_n] \in permute([s'_1, \ldots, s'_n]) :: (\forall i : 1 \leq i \leq n : s_i \sqsubseteq_\Sigma t_i))$.
4. $f(s_1, \ldots, s_n) \sqsubseteq_\Sigma f'(s'_1, \ldots, s'_n)$ if and only if $([f] \leq [f'] \wedge ([s_1, \ldots, s_n] \sqsubseteq_\Sigma [s'_1, \ldots, s'_n]))$.
5. $p(s_1, \ldots, s_n) \sqsubseteq_\Sigma p'(s'_1, \ldots, s'_n)$ if and only if $(p \leq_{pred} p' \wedge [s_1, \ldots, s_n] \sqsubseteq_\Sigma [s'_1, \ldots, s'_n])$.
6. $(exp_1 \sqsubseteq_\Sigma exp_2)$ if and only if $(\exists \theta \in \mathcal{SUB} :: exp_1 = \theta\ exp_2)$.
7. $((\text{if } exp_1 \text{ then } exp_2 \text{ else } exp_3) \sqsubseteq_\Sigma (\text{if } exp'_1 \text{ then } exp'_2 \text{ else } exp'_3))$ if and only if $(exp'_1 \sqsubseteq_\Sigma exp_1 \wedge exp_2 \sqsubseteq_\Sigma exp'_2 \wedge exp_3 \sqsubseteq_\Sigma exp'_3)$.

As described in Section 2, a method is described by its signature, pre- and postconditions. Suppose some method M is specified by Σ, let $pre(M)$ denote its precondition and $post(M)$ its postcondition. We can use a single predicate called a *characteristic expression* to describe a method. For some method M, the characteristic expression of M, $spec(M) \in \mathcal{E}_\Sigma$ is defined to describe its behavior: $spec(M) = (selected(M) \wedge pre(M)) \rightarrow post(M)$, where a newly introduced predicate "selected(M)" indicates whether the current method M is chosen for execution. The predicate $spec(M)$ says

that if method M is selected and precondition of M is satisfied then the postcondition of M should also be satisfied. Here we assume every method will terminate, where the specification of termination is beyond the scope of this paper. Using the generality relationship between expressions, we give the definition of a logical match between two methods.

Definition 3 *(Logical match between methods)* Let two methods M_1 and M_2 be specified by an OSPL Σ. M_2 is more general than M_1, denoted by $M_1 \sqsubseteq_{\Sigma,method} M_2$, if and only if $spec(M_1) \sqsubseteq_\Sigma spec(M_2)$.

Informally, if method M_2 is more general than method M_1 then it means that if the postcondition of M_2 *subsumes* the postcondition of M_1 and the precondition of M_1 *subsumes* the precondition of M_2.

Based on the generalities among methods, we can obtain the definition of generality between any two components, i.e., the logical match between two components:

Definition 4 *(Logical match between components)* For two components C_{query} and $C_{library}$, $C_{library}$ is more general than C_{query}, denoted by $(C_{query} \sqsubseteq_{\Sigma,comp} C_{library})$, if and only if $(\forall m_q \in methods(\rho C_{query}) :: (\exists m_l \in methods(C_{library}) :: m_q \sqsubseteq_{\Sigma,method} m_l))$, where ρ is a sequence of variable renamings of user-defined sorts.

That is, if component $C_{library}$ is more general than component C_{query} then for every method m_q of C_{query} there is at least one method m_l of $C_{library}$, such that m_l is more general than m_q. Component $\mathcal{C}$ is more specialized than component $\mathcal{D}$ if and only if $\mathcal{D}$ is more general than $\mathcal{C}$. The subsumption test algorithm in Figure 2 can be applied to the characteristic expressions of the two components being compared in order to determine their *generality* relationship. A simple subsumption test algorithm, which is based on the above definitions, between two sets of components is given in Figure 4, where $methods_A$ ($methods_B$) is the set containing the methods of the component $Comp_A$ ($Comp_B$). The cardinality of $methods_A$ ($methods_B$) is m (n).

As an example of how logical match can be used, consider the query specification in Figure 5 that describes the behavior of an integer set. The query component `intSet` matches the library component `realSet`, denoted by `intSet` $\sqsubseteq_{\Sigma,comp}$ `realSet`, with the variable renaming [`intSet` $\mapsto$ `realSet`] because:

1. `intSet : member` $\sqsubseteq_{\Sigma,method}$ `realSet : member`
2. `intSet : add` $\sqsubseteq_{\Sigma,method}$ `realSet : insert`
3. `intSet : remove` $\sqsubseteq_{\Sigma,method}$ `realSet : delete`
4. `intSet : belongs_to` $\sqsubseteq_{\Sigma,method}$ `realSet : member`

Input: Two sets $methods_A = \{A_1, A_2, ..., A_m\}$ and $methods_B = \{B_1, B_2, ..., B_n\}$.
Output: $Comp_B \sqsubseteq_{\Sigma,comp} Comp_A$?

begin
  find $\leftarrow$ true;
  while $methods_A \neq \{\}$ and find = true **do**
    select some $A_i \in methods_A$;
    $methods_A \leftarrow methods_A \setminus A_i$;
    %'\' denotes set substraction
    $set_B \leftarrow methods_B$;
    find $\leftarrow$ false;
    while $set_B \neq \{\}$ and find = false **do**
      select some $B_j \in set_B$;
      $set_B \leftarrow set_B \setminus B_j$;
      if $B_i \sqsubseteq_{\Sigma,method} A_j$
      then find $\leftarrow$ true;
    endwhile;
  endwhile;
  if find = false
  then return("$Comp_B \not\sqsubseteq_{\Sigma,comp} Comp_A$");
  else return("$Comp_B \sqsubseteq_{\Sigma,comp} Comp_A$");
end.

Figure 4: Deciding the *generality* relationship between components $Comp_A$ and $Comp_B$.

Let us only consider the case:
(`intSet : add` $\sqsubseteq_{\Sigma,method}$ `realSet : insert`).
The method `intSet:add` logically matches the method `realSet:insert` due to the following conditions,

1. $a \sqsubseteq_\Sigma e$
   because $int \leq real$
2. $\mathcal{A} \sqsubseteq_\Sigma \alpha$
   under sort renaming $[intSet \mapsto realSet]$
3. $(\mathcal{A} \cup \{a\}) \sqsubseteq_\Sigma (\alpha \cup \{e\})$
   from (1),(2), and subsumption test algorithm
4. $(size(\mathcal{A})) \sqsubseteq_\Sigma (card(\alpha))$
   from (2) and $card \simeq_{pred} size$
5. $(size(\mathcal{A}') \doteq size(\mathcal{A}) + 1 \wedge \mathcal{A}' \doteq \mathcal{A} \cup \{a\}) \sqsubseteq_\Sigma$
   $(\alpha' \doteq \alpha \cup \{e\} \wedge card(\alpha') \doteq card(\alpha) + 1)$
   from (3),(4), and subsumption test algorithm
6. $(\neg full(\mathcal{A})) \sqsubseteq_\Sigma (\neg full(\alpha))$
   from (2) and subsumption test algorithm
7. $spec$(`intSet : add`) $\sqsubseteq_\Sigma$ $spec$(`realSet : insert`)
   from (5),(6), and the definition of characteristic expression.
8. `intSet : add` $\sqsubseteq_{\Sigma,method}$ `realSet : insert`
   from (7) and Definition 3

component intSet
{

method create: $() \rightarrow \mathcal{A} : intSet$
requires $\neg existing(\mathcal{A})$
ensures $\mathcal{A} \doteq \emptyset$

method add: $(\mathcal{A} : intSet, a : int) \rightarrow \mathcal{A}' : intSet$
modifies $\mathcal{A}$
ensures $size(\mathcal{A}') \doteq size(\mathcal{A}) + 1 \wedge \mathcal{A}' \doteq \mathcal{A} \cup \{a\}$

method remove: $(\mathcal{A} : intSet, a : int) \rightarrow \mathcal{A}' : intSet$
requires $\neg empty(\mathcal{A})$
modifies $\mathcal{A}$
ensures $size(\mathcal{A}) \doteq size(\mathcal{A}') + 1 \wedge \mathcal{A} \doteq \mathcal{A}' \cup \{a\}$

method $belongs\_to(a : int, \mathcal{A} : intSet) \rightarrow \alpha : bool$
requires $\neg empty(\mathcal{A})$
ensures $\alpha \doteq a \in \mathcal{A}$
}

Figure 5: Query specification for an integer set.

If we attempt to apply relaxed exact match, we are not able to find the suitable library component **realSet**. But applying a logical match search **realSet** is returned for this query, since **realSet** is *more general* than **intSet**. Using this type of match, the user does not need to make major modifications to a more general component in order to reuse it. That is, only the following modifications are necessary: the operator *size* is implemented by the operator *card* and the sort *int* can be specialized from the sort *real*.

5 Implementation

A prototype system for applying specification matching process to constructing the hierarchical library and retrieving reusable components from that library has been implemented in the Quintus ProWindows language [1], a dialect of Prolog that supports the object-oriented organization of graphical elements. Figure 6 contains an example about the application of the specification matching process to the *construction* of component hierarchy and the *retrieval* of a set of candidate components according to some query specification.

A group of reusable components are classified to form a two-tiered hierarchy. The lower-level hierarchy generated by a subsumption test algorithm represents the *generality* relationships among the components, where the parent component is *more general* then the child component. The higher-level hierarchy is generated by a clustering algorithm that is applied to the most general components of the lower-level hierarchy [11]. The construction of the higher-level is based on *analogy* between component specifications and it is beyond the scope of this paper [12]. The two-tiered hierarchy provides a framework for storing, retrieving, and modifying reusable software components. On the right hand side of Figure 6, a retrieval option is selected from the main menu to execute the retrieval process. In this example, the user's request is to find those components that are more general than the query specification *adt_queue*, which is partially displayed in the left window of Figure 6. Below the search menu, a window displays the results obtained by the retrieval process. The results for this example are in the form of components that are more general than the query specification *adt_queue*. Once the reusable components are retrieved, they can be used as a set of candidates for further modification to fit the query specification [26].

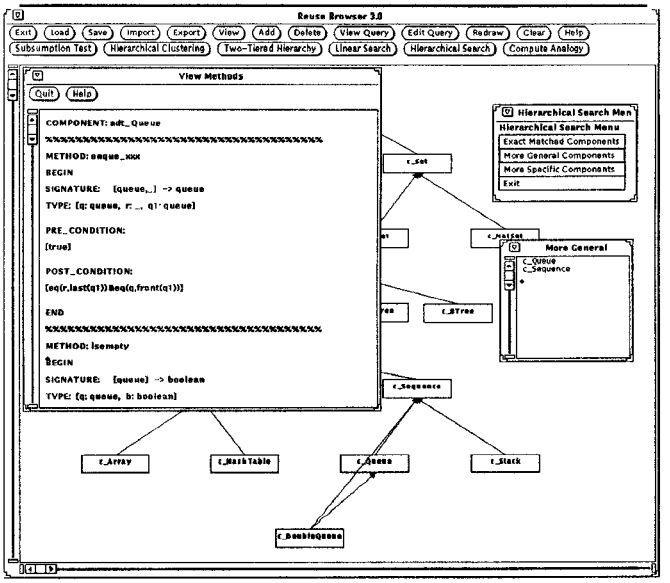

Figure 6: Sample application of construction and retrieval processes

6 Related Work

Zaremski and Wing [21] present *signature matching* as a tool to locate a reusable component. They gave definitions for a variety of matches at both the function and the module levels. A signature is used as a *key* to find a set of reusable components in their retrieval methods. They also identify a set of primitive functional matches

[1] A product of Quintus Computer Systems, Inc

that can be combined in several ways. Our approach to software reuse is to consider matches between specifications of the behavior of software components. We lay foundations of specification matching based on formal specifications, where signature matching can be used as the front end of a retrieval process to reduce the search space of reusable library components.

Perry and Popovich [27] present a system *Inquire* that incorporates both the browser and the predicate-based search mechanism. They use formal specification to describe the components which include operations, data objects, and modules. The formal interface specifications are also used as the medium for retrieval process. Obviously, their approach is closest to ours. However, in this work, we emphasize on the complete automation of retrieval process through well-defined specification languages and specification matching process.

The GURU project [3, 4] automatically assembles large components by using information retrieval techniques. The construction of their library consists of two steps: first, attributes are automatically extracted from natural language documentation by using an indexing scheme; then a hierarchy is automatically generated using a clustering technique. The indexing scheme is based on the analysis of natural language documents obtained from manual pages or comments. The assumption is that natural language documentation is a rich source of conceptual information. In contrast with their work, formal specification is used as a medium to describe and retrieve the reusable components in our work.

Another work on classifying software in order to enable control over the search space is done by Prieto-Díaz [28]. His method classifies software using *facets*, which describe the characteristics of a given software. We view specification matching as a complementary approach to these traditional retrieval techniques. Furthermore, our methods have been extended to the classification of software components by using the logical relationship to construct a component hierarchy. A graphically-oriented prototype environment has been developed to support our formal approach to software reuse [11]. Example applications of our reuse environment include reusing a `C++` class library and the `Motif` widget set.

7 Summary and Future Work

This paper presented the foundations for the use of specification matching as a useful tool for the retrieval of reusable components. Precise definitions have been given for the *relaxed exact match* and *logical match* at both the *component* and *method* levels. We will explore more advanced techniques to facilitate and implement the specification matching process. More domain-specific knowledge will be incorporated into the determination of the subsumption relationship among components in order to locate reusable components. Specification matching will be used to facilitate another important reuse process: modifying the code of retrieved components in order to fit query specifications.

References

[1] Ted J. Biggerstaff, editor. *Software Reusability Vol. 1: Concepts and Models.* ACM Press, New York, 1989.

[2] Charles W. Krueger. Software Reuse. *ACM Computing Surveys*, 24(2):131–183, June 1992.

[3] Y.S. Maarek, D.M. Berry, and G.E. Kaiser. An Information Retrieval Approach for Automatic Constructing Software Libraries. *IEEE Trans. Software Engineering*, 17(8):800–813, August 1991.

[4] R. Helm and Y.S. Maarek. Integrating Information Retrieval and Domain Specific Approaches for Browsing and Retrieval in Object-Oriented Class Libraries. In *Proceedings of OOPSLA'91*, pages 47–61, 1991.

[5] B.W. Weide, W.F. Ogden, and S.H. Zweben. Reusable Software Components. *Advances in Computers*, 33:1–65, 1991.

[6] Jeannette M. Wing. A Specifier's Introduction to Formal Methods. *IEEE Computer*, 23(9):8–24, September 1990.

[7] R.L. London. Specifying Reusable Components Using Z: Realistic Sets and Dictionaries. *ACM SIGSOFT Software Engineering Notes*, 14(3):120–127, May 1989.

[8] G. Caldiera and V. Basili. Identifying and Qualifying Reusable Software Components. *IEEE Computer*, 24(2):61–70, Febrary 1991.

[9] F. Nishida, S. Takamatsu, Y. Fujita, and T. Tani. Semi-Automatic Program Construction from Specification Using Library Modules. *IEEE Transaction on Software Engineering*, 17(9):853–870, 1991.

[10] Jun-Jang Jeng. *Applying Formal Methods to Software Reuse.* PhD thesis, Michigan State University, May 1994.

[11] Jun-Jang Jeng and Betty H.C. Cheng. Using Formal Methods to Construct a Software Component Library. *Lecture Notes in Computer Science*, 717:397–417, September 1993. (Proc. of Fourth European Software Engineering Conference).

[12] Jun-Jang Jeng and Betty H. C. Cheng. Using Analogy to Determine Program Modification Based on Specification Changes. In *Proceedings of 5th International Conference on Tools with Artificial Intelligence, 1993*, pages 113–116.

[13] Jun-Jang Jeng and Betty H.C. Cheng. Using Automated Reasoning to Determine Software Reuse. *International Journal of Software Engineering and Knowledge Engineering*, 2(4):523–546, 1992.

[14] Betty H.C. Cheng and Jun-Jang Jeng. Formal methods applied to reuse. In *Proceedings of the Fifth Workshop in Software Reuse*, 1992.

[15] J.A. Goguen and J. Meseguer. Order-sorted algebra I: equational deduction for multiple inheritance, overloading, exceptions and partial operations. *Theoretical Computer Science*, 105:217–273, 1992.

[16] Uwe Waldmann. Semantics of order-sorted specifications. *Theoretical Computer Science*, 94:1–35, 1992.

[17] C.Beierle, U. Hedtstúck, U. Pletat, P.H. Schmitt, and J. Sickmann. An order-sorted logic for knowledge representation systems. *Artificial Intelligence*, 55:149–191, 1992.

[18] H.P. Haughton and K. Lano. Object revisited. In *Proceedings of IEEE Conference of Software Maintenance*, pages 152–161, Italy, October 1991.

[19] James H. Cross II, Elliot J. Chikofsky, and Jr Charles H. May. Reverse engineering. *Advances in Computers*, 35:200–255, 1992.

[20] Betty H.C. Cheng and Gerald C. Gannod. Constructing formal specifications from program code. In *Proc. of Third International Conference on Tools in Artificial Intelligence*, pages 125–128, November 1991.

[21] Amy Moormann Zaremski and Jeannette M. Wing. Signature Matching: A Key to Reuse. In *Proceedings of the ACM SIGSOFT '93 Symposium on the Foundations of Software Engineering*, pages 182–190, December 1993.

[22] Chin-Liang Chang and Richard Char-Tung Lee. *Symbolic Logic and Mechanical Theorem Proving.* Academic Press, 1973.

[23] Christoph Walther. Many-Sorted Unification. *Journal of Association for Computing Machinery*, 35(1):1–17, January 1988.

[24] J.A. Robinson. A Machine Oriented Logic Based on Resolution Principle. *Journal of ACM*, 12(1):227–234, 1965.

[25] K.H. Blasius and H.J. Burchert, editors. *Deduction Systems in Artificial Intelligence.* Ellis Horwood Series in Artificial Intelligence, 1989.

[26] Jun-Jang Jeng and Betty H.C. Cheng. A Formal Approach to Reusing More General Components. In *Proceedings of The Ninth Knowledge-Based Software Engineering Conference*, pages 90–97, Monterey, California, September 20–23, 1994.

[27] Dewayne E. Perry and Steven S. Popovich. Inquire: Predicate-Based Use and Reuse. In *Proceedings of The Eighth Knowledge-Based Software Engineering Conference*, pages 144–151, September, 1993.

[28] Rubén Prieto-Díaz. Implementing Faceted Classification for Software Reuse. *Communication of ACM*, 34(5), May 1991.