Proc. of 4th Eur. Soft. Eng. Conf., Lect. Notes in Comp. Science, Vol. 717, pp. 397-417, Sept. 1993.

Using Formal Methods to Construct a Software Component Library*

Jun-Jang Jeng and Betty H.C. Cheng

Department of Computer Science
Michigan State University
East Lansing, MI 48824, USA

Abstract. Reusing software may greatly increase the productivity of software engineers and improve the quality of developed software. Software component libraries have been suggested as a means for facilitating reuse. Using formal specifications to represent software components facilitates the determination of reusable software because they more precisely characterize the functionality of the software, and the well-defined syntax makes processing amenable to automation. This paper presents an approach, based on formal methods, to the classification and organization of reusable software components. From a set of formal specifications, a two-tiered hierarchy of software components is constructed. The formal specifications represent software that has been implemented and verified for correctness. The hierarchical organization of the software component specifications provides a means for storing, browsing, and retrieving reusable components that is amenable to automation. A prototype browser that provides a graphical framework for the classification and retrieval process is also described.

1 Introduction to Software Reuse

Software reuse has been claimed to be a means for overcoming the software crisis [1, 2, 3, 4]. 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 for constructing software libraries [5, 6]. Unfortunately, software components represented by natural-language may hinder the retrieval process due to the problems of ambiguity, incompleteness, and inconsistency inherent to natural languages. All of the above mentioned problems can be minimized by using formal specifications to represent software components [7, 8, 9, 10, 11, 12, 13].

The major objective of a reuse system is to classify the reusable components and to retrieve them from an existing library [14]. Formal specifications facilitate the

* The work is supported in part by NSF grant CCR-9209873 and a Michigan State University All University Research Initiation Grant.

above tasks because they provide a precise characterization of the purpose of a piece of software and make it easier to determine the reusability of software. We present a classification scheme and algorithms for automatically constructing a hierarchy of software components that provide a means for representing, storing, browsing, and retrieving reusable components.

The hierarchical relationships of the reuse system are based on a *generality* relationship and similarities between software components. The similarities are calculated with respect to a partition of operators into equivalence classes. In order to combine these two concepts into one framework, the component library is structured as a two-tiered hierarchy in two stages. The resulting library structure consists of lower-level and higher-level hierarchies. The lower-level hierarchy is created by a *subsumption test algorithm* that determines whether one component is more general than another. Based on the *generality* relationship, the most general components are placed at the top of the hierarchy and the more detailed or restrictive components at the bottom. The higher-level hierarchy is generated by a classical *hierarchical clustering algorithm* that groups the most similar components together. The end result is a connected hierarchy of software components organized from the most general to the most specific.

The GURU project [5] automatically assembles large components by using information retrieval techniques. The construction of the 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 similar to our hierarchical clustering algorithm. Their indexing scheme is based upon analysis of natural-language documentation obtained from manual pages or comments. The assumption is that natural-language documentation is a rich source of conceptual information. However, natural language is not a rigorous language to specify the behavior of software components. A formal specification language can serve as a contract, and a means of communication among a client, a specifier and an implementer [13]. Because of their mathematical basis, formal specifications are more precise and more concise than natural-language documentation.

The MAPS system [10] applies formal specifications termed *case-like expressions* to specify software modules. MAPS exploits the unification capability to search through reusable modules in the library. However, their library is not hierarchically organized, thus the search space could become very large once the number of software modules in the library increases.

The remainder of this paper is organized as follows. Section 2 describes the notation used in the specification of software components. Section 3 describes the subsumption test algorithm. The hierarchical clustering algorithm is described in Section 4. Section 5 describes searching techniques for reusable components in the two-tiered hierarchy. Section 6 describes the implementation of a browser that handles the construction of a two-tiered hierarchy and the search and retrieval of reusable components. Finally, Section 7 summarizes this work and discusses future investigations.

2 Specification of Software Components

In this project, predicate logic is used to specify software components. Most software is made up of procedural and data abstractions, that is, procedures and user-specified and system-defined data structures [15]. Object-oriented analysis can be used to decompose complex software, which involves defining a set of user-specified data abstractions or *abstract data types* (ADTs) [16, 17, 18, 19, 20, 21]. Thus, in order to apply an object-oriented approach to software reuse, this project focuses on data abstraction, where it is assumed that procedural abstractions are implicitly addressed when discussing the operations that are applicable to the data abstractions. The specification for a software component corresponds to the specification of an abstract data type and a set of methods that operate on that abstract data type. Each method is specified by an *interface*, *type declarations*, a *precondition*, and a *postcondition*. The interface of a method describes the syntactic specification of the method. The typing information describes the types of input and output parameters and internal (local) variables. The precondition describes the condition of the variables prior to the execution of the method whose behavior is described by the postcondition [22]. Currently, program *invariants* are not used in the construction and retrieval processes, with the understanding that the software components being handled are simple enough such that the invariants can easily be derived.

Figure 1 gives the grammar of the specification language. In this grammar, symbols expressed in the roman font represent non-terminals, italicized symbols represent terminals, bold-faced symbols denote keywords, the Kleene star (*) denotes zero or more repetitions of the preceding unit, and parentheses ('()') indicate groupings. The symbol '::' separates an identifier from a description of the *value* denoted by the identifier, and the symbol ':' separates identifier declarations from a description of the *type* associated with the identifier. The boolean operators obey the following decreasing precedence order: negation ($\neg$), conjunction ($\wedge$), disjunction ($\vee$), implication ($\Rightarrow$), and if and only if ($\Leftrightarrow$). Primitive types, including *Bool*, *Int*, and *Real*, are pre-defined and can be referenced by the users.

Figure 2 shows an example specification of the abstract data type *Array*, which has been stored as a software component. Three methods are defined on *Array*: *assign_element*, *sort*, and *last_element*. The lines beginning with *in*, *out*, and *local* describe the types of input parameters, output parameters, and internal (local) variables, respectively (comments are delimited by %). In a software component specification, it is possible to give polymorphic definitions [23], that is, an operation may have more than one meaning. For example, the method *last_element* of *Array* is defined as a polymorphic function that returns the last element of an array, where element can be of any type. The bars || || at the beginning of the definition introduce a generically typed variable E, which indicates that the type of variable E is irrelevant.

3 Lower-Level Hierarchy

The objective of this project is to construct a hierarchical organization of reusable components that will provide a fast means for browsing, retrieving, and searching of software components exploiting the automated reasoning techniques applicable to

$component$ = **type** type_name: ($method$)*
$method$ = **method** method_name
sort: (type_name)* → (type_name)* **is**
in((variable: type_name)*)
local((variable: type_name)*)
out((variable: type_name)*)
{ **pre**: $expression$ }
{ **post**: $expression$ }
$expression$ = true
| false
| ($expression$)
| ¬ $expression$
| $expression$ ∧ $expression$
| $expression$ ∨ $expression$
| $expression$ ⇒ $expression$
| $expression$ ⇔ $expression$
| (∀ variable : type :: $expression$)
| (∃ variable : type :: $expression$)
| predicate_name [($term$ (, $term$)*)]
| $term \stackrel{\text{def}}{=} expression$

$term$ = variable
| function_name [($term$ (, $term$)*)]

Fig. 1. Grammar for software component specifications

predicate logic specifications. The lower-level hierarchy provides a means for a fine-grained, precise determination of reuse, where logical reasoning can be applied to the specifications. The construction of the lower-level hierarchy serves to classify a set of software components according to the subsumption relationship between reusable components, where, in simple terms, component A is said to subsume component B if A is more *general* than B, denoted by $A \sqsupseteq_{comp} B$. A new resolution rule is described that increases the range of candidates, as compared to the number of exact matches, that can be retrieved using automated reasoning techniques.

Component A is *more general* than component B ($A \sqsupseteq_{comp} B$) if, for every method (operation) f in component A, there exists at least one method f' in component B such that f is more general than f', denoted by $f \sqsupseteq_{method} f'$. Method f is said to be more general than another method f' if $pre(f') \sqsupseteq pre(f)$ and $post(f) \sqsupseteq post(f')$, where $pre(f)$ and $post(f)$ represent the pre- and postconditions of f, respectively. The *subsumption* relationship between clauses ($\sqsupseteq$), methods ($\sqsupseteq_{method}$), and components ($A \sqsupseteq_{comp} B$) is further explained in the following section. If component A is more general than component B, then B is said to be a *child* of A and A is a *parent* of B. The pre- and postconditions for a method of a given component are expressed in *disjunctive normal form* (DNF). Therefore, in order to determine which method is more general is to determine which method contains more

type Array:

method ‖E‖ assign_element: Array × E → Array **is**
in(s: Array, e: E)
local()
out(s': Array)
{ **pre**: true }
{ **post**: $len(s') = len(s) + 1 \wedge s'(len(s')) = e$
% len: index of the last element in array with a value
}

method sort: Array → Array **is**
in(s: Array)
local(i, j, min, max: Int)
out(s': Array)
{ **pre**: true }
{ **post**: s' = permutation(s) ∧
($\forall$ i : min ≤ i ≤ max :: ($\forall$ j : min ≤ j < i :: $s'(j) \leq s'(i)$))
}

method ‖E‖ last_element: Array → E **is**
in(s: Array)
local()
out(e: E)
{ **pre**: $len(s) \neq 0$ }
{ **post**: last_element(s) $\stackrel{\text{def}}{=}$ $s(len(s))$
}

Fig. 2. Specification of software component *Array*

general postconditions, that is, weaker requirements. Section 3.2 gives an algorithm that builds the lower-level hierarchy based on the *generality* relationship between components ($\sqsupseteq_{comp}$).

An abstract data type (ADT), is a behavioral notion and may be implemented by many different classes. A class is a program module that implements an abstract data type. A subtype is also an ADT, each of whose objects behave in a way similar to objects of its supertypes. A subclass is an implementation that is derived by inheritance from its superclass. A subtype represents a behavioral relationship. When a subtype is derived from some supertype, the object's behavior with this subtype can be verified according to the objects with its supertype instead of reverifying this subtype. In contrast, a subclass relationship is a purely implementation relationship. In the `C++` language, the subclass relationship is implementation-specific and cannot represent the true supertype-subtype relationship. The *generality* relationship in our system is similar to the *supertype-subtype* relationship.

3.1 Determining generality relationship between two components

Chang and Lee's subsumption test algorithm [24] is used to decide the *subsumption* relationship between clauses, that is, whether clause A subsumes clause B, denoted

by $A \sqsupseteq B$. In this algorithm, the traditional resolution strategy [25] (shown in Figure 3) is exploited to compute the resolvents of two clauses, say C_1 and C_2. Atom L is said to be *congruent* to atom L', denoted by $L \simeq L'$, when both L and

---

C_1: $L, K_1, ..., K_n$
C_2: $\neg L, M_1, ..., M_m$
———————————————
resolvent: $K_1, ..., K_n, M_1, ..., M_m$.

Fig. 3. Resolution Rule

L' are in an equivalence class partition *eq_class* that may be defined by the user or the system (see Section 4.1 for further details). Following the approach of the resolution rule, if a *congruity* relationship exists between L and L', then L and $\neg L'$ can be eliminated in order to obtain a *c-resolvent*, a resolvent with respect to the congruity relationship. As a result, a modified resolution rule given in Figure 4 is derived, where σ is a *substitution* that maps variables to terms.

C_1: $L, K_1, ..., K_n$
C_2: $\neg L', M_1, ..., M_m$ $\quad L\sigma \simeq L'\sigma$
———————————————
resolvent: $K_1\sigma, ..., K_n\sigma, M_1\sigma, ..., M_m\sigma$.

Fig. 4. Modified Resolution Rule

Using the modified resolution rule, the subsumption test algorithm [24] is modified to find the *c-resolvent* of two clauses C_1 and C_2 rather than their resolvent. The modified subsumption test (MST) algorithm can be applied to every pair of methods of the two components being compared in order to determine the *generality* relationship between two components. The MST algorithm between two sets of methods is shown in Figure 5, 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).

3.2 Algorithms for Building the Lower-level Hierarchy

Based on Algorithm 1, the *generality* relationship can be determined between any pair of components in order to build the lower-level hierarchy. The straightforward approach is to construct the lower-level hierarchy by performing a pair-wise comparison between all components. The pair-wise comparison algorithm is shown in

Algorithm 1 *More_General_Component*

Input: *Two sets* $methods_A = \{A_1, A_2, ..., A_m\}$ *and* $methods_B = \{B_1, B_2, ..., B_n\}$.
Output: *The* generality *relationship between components* $Comp_A$ *and* $Comp_B$.
Procedure:

```
begin
  find ← true;
  while methods_A ≠ {} and find = true do
        select some A_i ∈ methods_A;
        methods_A ← methods_A \ A_i;
        set_B ← methods_B;
        find ← false;
        while set_B ≠ {} and find = false do
              select some B_j ∈ set_B;
              set_B ← set_B \ B_j;
              if A_i ⊒_method B_j
              then find ← true;
        endwhile;
  endwhile;
  if find = false
  then return("¬(Comp_A ⊒_comp Comp_B)");
  else return("Comp_A ⊒_comp Comp_B");
end.
```

Fig. 5. Using MST to decide the *generality* relationship between components $Comp_A$ and $Comp_B$.

Figure 6. However, the transitivity property of the *generality* relationship can be exploited in order to reduce the computational complexity of building the lower-level hierarchy. If $A \sqsupseteq B$ and $B \sqsupseteq C$ then the relation $A \sqsupseteq C$ is automatically established without having to compare components A and C. A few definitions are given before presenting the improved algorithm. For some *set of lattices* (SOL) Ψ, the set of top nodes in Ψ is denoted by $Top(\Psi)$ and the set of bottom nodes by $Bottom(\Psi)$. If node α has no parent nodes in the SOL Ψ, then $\alpha \in Top(\Psi)$. Similarly, if α has no children nodes in the SOL Ψ, then $\alpha \in Bottom(\Psi)$. The internal nodes in Ψ are defined as $Internal(\Psi) = \Psi \setminus (Top(\Psi) \cup Bottom(\Psi))$, where '\' represents set subtraction. For some node $\alpha \in \Psi$, the set of parent nodes of α is denoted by $parent(\alpha)$ and the set of children nodes by $child(\alpha)$. The set of the descendants of α, denoted by $descendant(\alpha)$, is defined as follows:

$$\beta \in descendant(\alpha) \Leftrightarrow ((\beta \in child(\alpha)) \vee (\exists \gamma : \gamma \in child(\alpha) : \beta \in descendant(\gamma))$$

The set of the ancestors of α, denoted by $ancestor(\alpha)$, has a similar definition. A parallel algorithm to build the lower-level hierarchy based on recursive comparisons and the generality relationship is given in Figure 7. A pictorial representation of an example construction of the lower-level hierarchy by procedure *Recursive_Comparison* is shown in Figure 8, where dashed lines represent the application of the procedure

Algorithm 2 *Pairwise_Comparison*

Input: *A set of components* $SET = \{C_1, C_2, ..., C_n\}$.
Output: *A hierarchy of components based on the generality relationship.*
Procedure:

```
begin
  while SET ≠ {}
      select some component C_i ∈ SET;
      SET ← SET \ C_i;
      set ← SET;
      while set ≠ {}
          select some C_j ∈ set;
          set ← set \ C_j;
          if C_i ⊒_comp C_j
          /* More_General_Component algorithm will be used to compare C_i and C_j */
          then make C_i a parent of C_j
          else if C_j ⊒_comp C_i
              then make C_j a parent of C_i
              endif
          endif
      endwhile;
  endwhile;
end.
```

Fig. 6. Building the lower-level hierarchy by pair-wise comparison.

Recursive_Comparison, solid lines represent the *generality* relationship, and the dotted lines encapsulate SOLs. Initially, the example contains eight SOLs and each SOL contains only one component. These eight SOLs are merged into one SOL after applying the two procedures *Compare* and *Merge*.

$Compare(\Psi_i, \Psi_j)$ determines the generality relationship between nodes in SOLs Ψ_i and Ψ_j by using a recursive approach. For example, if some node α is more general than some top node β of Ψ, then it is not necessary to compare α with the descendants of β. However, if some top node β is more general than α then the comparison between α and the descendants of β is required. The same reasoning can be applied to the comparison between α and the bottom nodes of the SOL Ψ. The procedure $Merge(\Psi_i, \Psi_j)$ "connects" the newly generated *generality* relationship between SOLs Ψ_i and Ψ_j to form a new SOL. *Recursive_Comparison* can be implemented as a parallel algorithm since the comparisons between the SOLs are independent of each other. Only the nodes in *Top* and *Bottom* sets are compared in the procedure *Compare*.

Applying algorithm $Compare(SOL_A, SOL_B)$ to two SOLs SOL_A and SOL_B is illustrated in Figure 9. For discussion purposes, attention is focused on the top node E in SOL_A and the bottom node F in SOL_B. If $F \sqsupseteq E$, then make node F a parent of node E since all nodes in $ancestor(F) \cup \{F\}$ must subsume the nodes in $descendant(E) \cup \{E\}$. However, if $E \sqsupseteq F$, then node E needs to be compared

Algorithm 3 *Recursive_Comparison*

Input: *A set* $\{\Psi_0, \Psi_1, ..., \Psi_{n-1}\}$, *where* Ψ_i *represents a set of lattices and assume* $n = 2^m$. *Initially,* $\Psi_i = \{C_i\}$ *where* C_i *is a component.*

Output: Ψ_0 *contains a hierarchy of components based on the generality relationship.*

Procedure:

```
begin
  for i := 0 to m-1 do
        d ← 2^i;
        do all Ψ_k where 0 ≤ k ≤ 2^m − 1     /* Parallel execution of all iterations */
            if k mod 2^(i+1) = 0
            then
                Compare(Ψ_k, Ψ_(k+d));
                Ψ_k ← Merge(Ψ_k, Ψ_(k+d));
            endif;
        end_do_all;
  endfor;
end.
```

Fig. 7. Building lower-level hierarchy by recursive comparison.

with the nodes in $ancestor(F)$ and node F needs to be compared with the nodes in $descendant(E)$ in order to obtain complete *generality* relationships. Using the recursive method to build the lower-level hierarchy may reduce the computational time of construction since the comparisons of the internal nodes in the SOL can be eliminated.

4 Higher-Level Hierarchy

After applying the MST, the software components may be grouped into disjoint clusters in a set of graphs (ASG). In order to form a connected hierarchy of software components, a conventional clustering algorithm [26] is applied to the most general components obtained from the MST, that is, the roots of trees and the top elements of the lattices in ASG.

Classification by clustering techniques has been used in many areas of research, including information retrieval and image processing [27]. Typically, the objective of clustering is to form a set of clusters such that the intercluster similarity is low, and the intracluster similarity is high. Applying a clustering algorithm to the most general components of the lower-level hierarchy leads to the generation of the higher-level hierarchy of the component library. The similarity between two components X and X', denoted by $s(X, X')$, is used as the basic criterion to determine clusters. In general, the criterion used to evaluate similarity determines the shape of the resultant clusters.

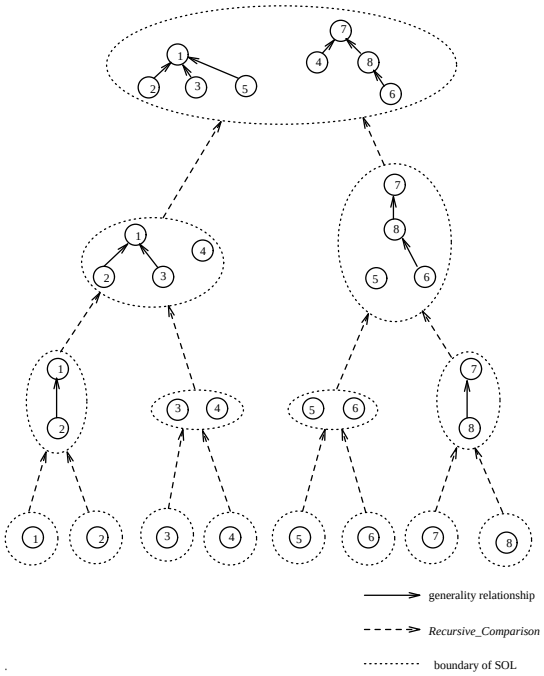

Fig. 8. Example of building hierarchy by recursive comparison.

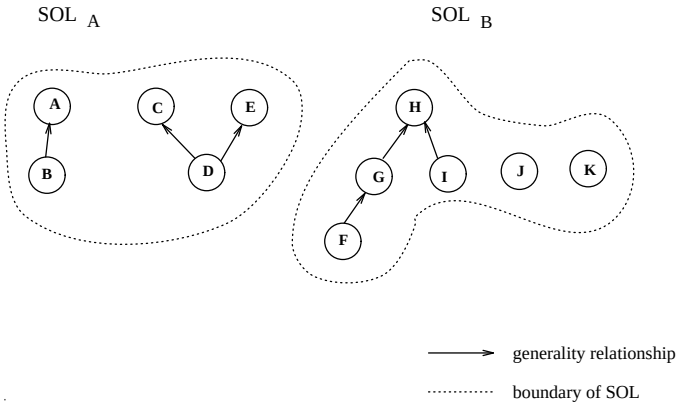

Fig. 9. Example of comparing two SOLs.

4.1 Measure of Similarity between Components

In this section, a simple evaluation method for computing similarity is given. The similarity between a pair of components, X and Y, is denoted by $s(X,Y)$. Similarity is symmetric, thus for any two components, X and Y:

$$s(X,Y) = s(Y,X).$$

In addition, similarity s is said to be normalized if $0 \leq s(X,Y) \leq 1$. Each predicate formula in the library expressed in DNF represents a software component and is regarded as one of the input objects that are to be classified by the hierarchical clustering algorithm.

If $s(X,Y)$ represents the similarity between two software components X and Y, then it is assumed that X and Y are of the following forms.

$$X = x_1 \vee x_2 \vee ... \vee x_m \text{ and}$$
$$Y = y_1 \vee y_2 \vee ... \vee y_n .$$

The disjuncts x_i and y_i are defined in terms of conjuncts, that is,

$$x_i = p_{i_1} \wedge p_{i_2} \wedge ... \wedge p_{i_{u_i}}, \quad 1 \leq i \leq m \quad \text{and}$$
$$y_i = q_{i_1} \wedge q_{i_2} \wedge ... \wedge q_{i_{v_i}}, \quad 1 \leq i \leq n,$$

where u_i and v_i are the number of conjuncts within disjunct x_i and y_i, respectively.

The disjuncts of each object are ordered from left to right in a nondecreasing order according to the number of conjuncts in each disjunct. Given that u_i and v_i represent the number of conjuncts for disjuncts x_i and y_i, respectively, the following inequalities are *true*: $u_{i-1} \leq u_i$ and $v_{i-1} \leq v_i$, for all i. Moreover, each conjunct p_{i_k} is associated with an equivalence class **eq_class**. For example, if $p_{i_k} = greater(a,b)$ and *greater* is in the equivalence class for *comparison*, then $\textbf{eq\_class}(p_{i_k}) = comparison$. The equivalence classes may be specified by the users or be system-defined.

The number of equivalence classes in a software component library is assumed to be a known value, say T. Using the above definitions, a matrix $X_{m\times(T+1)}$ is constructed for every component X. The matrix $X_{m\times(T+1)}$ derived from component X has m rows and $T+1$ columns. $X(i,j)$ represents the entry in row i and column j. Row i represents the i^{th} disjunct of X as follows, where there are u_i conjuncts in disjunct x_i.

$$X(i,0) = u_i, \quad 0 \leq i \leq m-1 \text{ and}$$
$$X(i,j) = l, \quad x_i \textit{ has } l \textit{ terms in eq\_class } j.$$

Similarly, for component Y containing v_i conjuncts in disjunct y_i, the corresponding matrix is defined by

$$Y(i,0) = v_i, \quad 0 \leq i \leq n-1 \text{ and}$$
$$Y(i,j) = l, \quad y_i \textit{ has } l \textit{ terms in eq\_class } j.$$

From the derived matrices $X_{m\times(T+1)}$ and $Y_{n\times(T+1)}$, the *similarity matrix* $s'_{m\times n}$ is

constructed. The following expression defines $s'_{m\times n}$.

$$\begin{aligned} for\ all\ i,j \quad if \quad & X(i,0) = Y(j,0) \\ then \quad & s'(i,j) = \frac{\sum_{t=1}^{T} 2 * min(X(i,t), Y(j,t))}{\sum_{t=1}^{T}(X(i,t) + Y(j,t))} \\ else \quad & s'(i,j) = 0 \end{aligned} \tag{1}$$

where $s'(i,j)$ is the similarity of the i^{th} disjunct of X and the j^{th} disjunct of Y. The similarity between two conjunctive expressions from two software components is calculated according to the minimum number of common occurrences of a given equivalence class. Since the results from the clustering process are purely based on syntactic similarities, only the disjuncts with the same number of conjuncts are selected for comparison. The semantic similarities are used in the construction of the lower-level hierarchy. Assume N is the number of nonzero entries in $s'_{m\times n}$. The similarity between software components X and Y is calculated as follows:

$$s(X,Y) = \frac{\sum_{i=1}^{m}\sum_{j=1}^{n} s'(i,j)}{N}. \tag{2}$$

Here, $s(X,Y)$ is a normalized similarity since $0 \leq s(X,Y) \leq 1$. The following example is presented for clarification purposes.

Example 1. Suppose the similarity of two components X and Y is to be computed, where both specifications are in DNF. Let $X = (C_1 \wedge C_2) \vee (C_2 \wedge C_3 \wedge C_3) \vee (C_3 \wedge C_4 \wedge C_5)$ and $Y = (C_3) \vee (C_2 \wedge C_3) \vee (C_3 \wedge C_3 \wedge C_5) \vee (C_2 \wedge C_5 \wedge C_5)$, where C_i refers to the term that corresponds to the i^{th} equivalence class. There are 5 equivalence classes in this case, so $T = 5$. X has 3 disjuncts and Y has 4 disjuncts. The corresponding matrices for X and Y are shown in Figures 10a and 10b, where the vertical axis represents the disjuncts in each component and the horizontal axis refers to the equivalence classes. From Formula (1), the similarity matrix $s'(X,Y)$ can be computed yielding results shown in Figure 10c, where the vertical axis represents the disjuncts in component X and the horizontal axis refers to the disjuncts in the Y component. From Formula (2), the similarity $s(X,Y) = \frac{2/4+4/6+2/6+4/6+2/6}{5} = \frac{1}{2}$ is obtained. This value is used as input to the clustering algorithm when determining which software components should be merged into one cluster.

4.2 Hierarchical Clustering

Input to a clustering algorithm is a set of components and the similarity values between each pair of components. A finite set of components is denoted by $X = \{x_1, x_2, ..., x_n\}$. Output from the clustering algorithm is a partition $\star = \{G_1, G_2, ..., G_N\}$, where $G_k, k = 1, ..., N$ is a subset of X such that

$$G_1 \cup G_2 \cup ... \cup G_N = X, \quad \forall\, l, k, l \neq k,\ G_l \cap G_k = \emptyset, \tag{3}$$

and $G_1, G_2, ..., G_N$ are the clusters of $\star$.

	0	1	2	3	4	5
1	2	1	1	0	0	0
2	3	0	1	2	0	0
3	3	0	0	1	1	1

(a) Matrix for X

	0	1	2	3	4	5
1	1	0	0	1	0	0
2	2	0	1	1	0	0
3	3	0	0	2	0	1
4	3	0	1	0	0	2

(b) Matrix for Y

	1	2	3	4
1	0	2/4	0	0
2	0	0	4/6	2/6
3	0	0	4/6	3/6

(c) $s'(X,Y)$

Fig. 10. Matrices for components X, Y, and $s'(X,Y)$, respectively

The relationship between the partition of clusters generated from the intermediate stages of refinement, denoted by $\star^i$, $i = 1, ..., K$, is expressed as follows:

$$\star^i = \{G_1^i, ..., G_{N_i}^i\}, \quad \star^j = \{G_1^j, ..., G_{N_j}^j\}, \qquad i = 1, ...K, \; i < j < K+1, \qquad (4)$$

where for all l, $N_l \geq N$, and N is the final number of partitions. $\star^j$ is a refinement of $\star^i$, $i < j$, that is, for any member subset $G_k^i \in \star^i$, there exists $G_l^j \in \star^j$ such that $G_k^i \subseteq G_l^j$. Such groups formed by intermediate partitions yield a hierarchy of clusters. A method for generating such a hierarchy is termed *hierarchical clustering* [26].

In general, hierarchical clustering algorithms are divided into two categories: *divisive* algorithms and *agglomerative* algorithms. A divisive algorithm starts with the set X and divides it into a partition $\star^K = \{G_1^K, ..., G_{N_K}^K\}$, then each cluster G_i^K is subdivided to form a finer partition $\star^{K-1}$, and so on. An agglomerative algorithm initially regards each component as a single cluster: $\star^1 = \{\{x_1\}, \{x_2\}, ..., \{x_n\}\}$. The clusters are merged into a coarser partition $\star^2$, and the merging process continues until the trivial partition $\star^K = \{X\}$ is obtained. Thus an agglomerative clustering algorithm generates a sequence of partitions $\star^1 \rightarrow \star^2 \rightarrow ...\star^K$ that is ordered from a finer partition to a coarser one. This algorithm can be stopped at any partition $\star^l, 1 \leq l \leq K$, if the maximum value of computed similarities is below a specified threshold or if the number of clusters generated for a partition is equal to a user-specified or system-defined value.

In most agglomerative algorithms, only one pair of clusters is merged at a time. Hence if $\star^i = \{G_1^i, ..., G_{N_i}^i\}$ and $\star^{i+1} = \{G_1^{i+1}, ..., G_{N_{i+1}}^{i+1}\}$, then $N_{i+1} = N_i - 1$. That is, $N_i = n - i + 1, i = 1, ..., n$ and $\star^1 = \{\{x_1\}, \{x_2\}, ..., \{x_n\}\}, \star^N = \{X\}$. Figure 11 gives a pictorial representation of the refinement process. Similarity between clusters is used as the criterion for the selection of a pair of clusters in $\star^i$ that are to be merged. A pair of clusters (G_p, G_q) is selected to be merged if it has the maximum value of similarity among all pairs of clusters. Let the current partition be $\star = \{G_1, ..., G_N\}$. The similarity value between two clusters is the maximum value of all similarities calculated between disjuncts from the respective components.

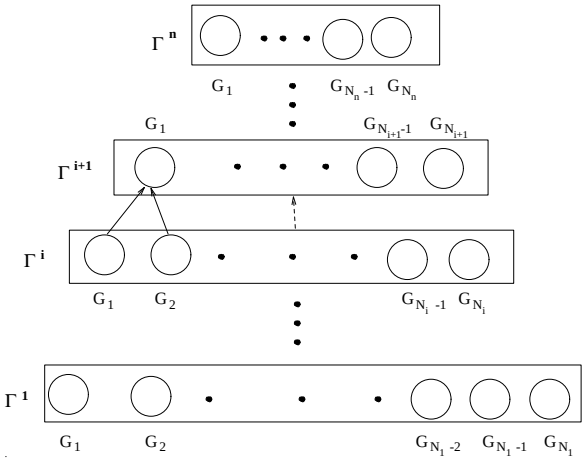

Fig. 11. Refinement of partitions in an agglomerative clustering algorithm

Formally, the *sim* relationship is expressed as

$$sim(G_p, G_q) = \max_{G_p, G_q \in \Gamma_{n-N}} (\max_{X \in G_p, Y \in G_q} s(X, Y)).$$

An agglomerative procedure is given in Figure 12. The similarities between the

Algorithm 4 *Agglomerative Clustering*

Input: *A set of disjoint lattices.*
Output: *A unified cluster.*
Procedure:

1. *Let each root of a tree or the top element of a lattice of the* ASG *be an initial cluster consisting of the single element.*
2. *Find the pair of clusters that has the maximum value of similarity among all pairs of clusters.*
3. *Merge the pair of clusters found in step 2 into a new cluster.*
4. *If there is only one cluster remaining, then stop. Otherwise, update similarity values between clusters; go back to step 2.*

Fig. 12. Agglomerative Hierarchical Clustering Algorithm

new cluster and other clusters are computed as follows: if G_p and G_q are merged

into a new cluster G_r, then

$$\forall i,\ sim(G_r, G_i) = min(sim(G_p, G_i), sim(G_q, G_i)).$$

4.3 Hierarchical Clustering Algorithm

The hierarchical clustering algorithm used is similar to Kruskal's algorithm for finding a minimal spanning tree [28], which always chooses an edge with the least weight in the construction of the spanning tree. In this case, weights are replaced by similarity values for software components and the maximal weight rather than the least weight is sought. After applying the algorithm, a tree-like hierarchical clustering is obtained. Figure 13 contains the detailed description of the hierarchical clustering algorithm where X is the set of predicate

components, $s(X_i, X_j)$ is the similarity between components X_i and X_j, and $sim(G_k, G_l)$ is the similarity between clusters G_k and G_l. The algorithm begins by creating a cluster for each software component to be classified, that is, the most general components found in the lower-level hierarchy, and the first partition contains all of the initial clusters. Next, a pairwise calculation of similarity between the clusters is made. Based on the similarity values, two clusters yielding the greatest value are selected to be merged. After the two clusters are merged, the similarity values between clusters is updated, thus defining the partition for the next iteration of the clustering algorithm. The user may specify an upper bound on the number of iterations (refinements) or stop the clustering algorithm while viewing the clustering process. This flexibility allows the user to incorporate background experience in order to determine when further refinements will fail to yield substantial changes between partitions. The final hierarchically organized library could be of the form given in Figure 14, where filled nodes, termed *real nodes*, represent software components and unfilled nodes are newly generated nodes created by the hierarchical clustering algorithm, called *meta-nodes*. A meta-node acts as a container for the software components from it which it was derived. Dashed lines represent relationships formed by the MST algorithm and the solid lines are formed by the hierarchical clustering algorithm representing similarity relationships.

5 Search for Reusable Candidates

The construction of the hierarchy is performed in two stages, beginning with the lower-level, the results of which are used in the construction of the higher-level. In contrast, the search and retrieval process proceeds from the higher-level hierarchy to the lower-level one, that is, from a coarse-grained search to a fine-grained one for reusable candidates. At the higher-level hierarchy, a query is mapped to some index that indicates the starting nodes within the hierarchy at which the searching algorithm is to begin. After performing the coarse-grained search, the search space may be greatly reduced. The remaining portion of the higher-level hierarchy and the corresponding lower-level is searched using formal reasoning techniques, thus providing an exact determination method. Three possible classes of existing specifications may be retrieved using logic reasoning techniques: an exact match to the new specification, a component more general than the current specification, or a

Algorithm 5 *Hierarchical Agglomerative Algorithm*

Input: *The set* $X = \{x_1, x_2, ..., x_n\}$ *and the similarities* $s(x_i, x_j)$, $1 \leq i, j \leq n$.
Output: *one or more clusters.*
Procedure:

```
begin
    N = n;
    for i = 1,...,N do
        G_i = {x_i}
    endfor;
    Γ_1 = {G_1, G_2, ..., G_N};
    Limit = 1;
    for 1 ≤ i, j ≤ N, i ≠ j do
        sim(G_i, G_j) = s(x_i, x_j)
    endfor; /* Initialization */

    /* If there is more than one cluster then iterate, otherwise stop. */
    while (N > Limit) do
        N = N − 1;
        /* Select the pair of clusters to be merged */
        find a pair of clusters G_p and G_q such that
                sim(G_p, G_q) = max_{G_i,G_j ∈ *_{n−N}, i≠j} sim(G_i, G_j)
                              = max_{G_i,G_j ∈ *_{n−N}, i≠j} max_{x∈G_i, y∈G_j} s(x, y);
        G_r = G_p ∪ G_q;
        Γ_{n−N+1} = (Γ_{n−N} − {G_p, G_q}) ⋃ {G_r}
        /* Update the similarity values */
        for all G_i ∈ Γ_{n−N+1}, G_i ≠ G_r, do
                calculate sim(G_r, G_i) = max_{x∈G_r, y∈G_i} s(x, y)
        endfor;
        Limit = query_user_for_number_of_clusters;
        /* Query user for a limit on the number of generated clusters */
    endwhile;
    return Γ_{n−N+1}
end.
```

Fig. 13. Hierarchical Clustering Algorithm

component more specific than the current specification. At any time, the user may opt to manually browse through the hierarchically organized specifications applying domain-specific knowledge to further the search process. Further search mechanisms in the hierarchy are currently under investigation.

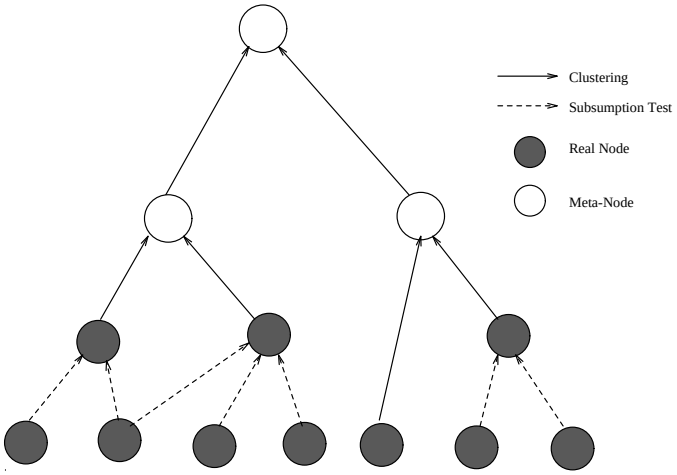

Fig. 14. Two-tiered hierarchy formed by the subsumption test and the clustering algorithms

6 Graphical Browser

A prototype browser that provides a graphical framework for the algorithms has been developed in the Quintus ProWindows language[2] a dialect of Prolog that supports the object-oriented organization of graphical elements [29]. The browser enables a user to graphically traverse the hierarchically organized specifications of software components. By making information about class hierarchies and method specifications more accessible, the browser facilitates the iterative process of developing and accessing reusable specifications.

Figure 15 contains an example application of the *Subsumption* option, where the original set of fifteen components in the library have been organized into four clusters shown in the four subwindows, respectively. The arrows indicate specific-to-general relationships. Figure 16 shows the results of the clustering algorithm, where the components *class1*, *class2*, and *class3* represent newly-created *meta-nodes*. Upon completion of the construction process, the user may choose to rename meta-nodes (e.g. *class1* and *class2*) to more descriptive names.

In searching for a reusable candidate, the user may select the node to begin the search, or the system will select the root node based on the syntactical components of the query. Figure 19a shows the state of the hierarchy before the search, where the user requests the search to begin at node *class2* as shown by the position of the doubly-nested menus containing the search option. Figure 19b shows the result of a search for an exact match among the software components, where the path of the

[2] A product of Quintus Computer Systems, Inc.

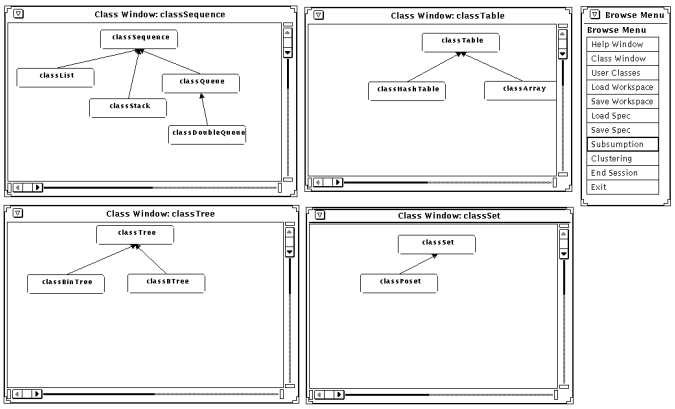

Fig. 15. Sample application of subsumption test algorithm

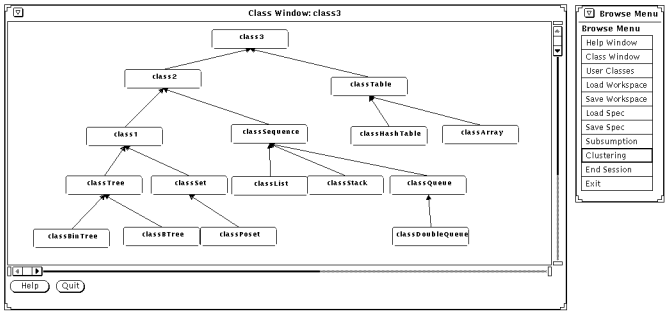

Fig. 16. Sample application of clustering algorithm

search is indicated by the highlighted nodes. In addition, the respective specifications of operations of the highlighted nodes may be displayed.

7 Conclusion

A classification scheme of software components expressed in first-order logic specifications has been presented in this paper. We have also described algorithms for implementing this scheme. The algorithms, implemented in Prolog, are able to construct a two-tiered hierarchical library from formal specifications. Thus, the hierarchy

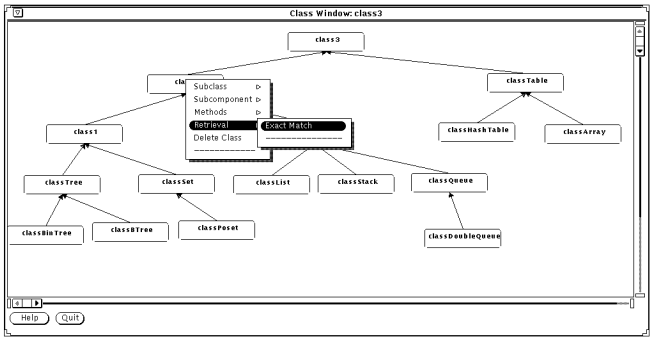

Fig. 19a. Hierarchy of components before invoking search routine

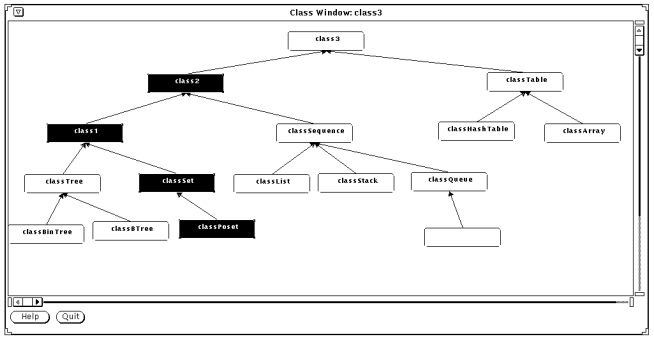

Fig. 19b. Sample search for an exact match among the software components

can help users store, browse, and retrieve existing reusable components. This work, although in its preliminary state of development, is a new approach to reusability, especially for reusing software components based upon formal specifications.

Given the framework that we have built, the system will be extended in several aspects. Efficient techniques are being developed to determine functional similarity between two software components. The abstraction scheme to form *meta-nodes* of software components will also be further investigated. An efficient searching algorithm that includes hashing and reasoning schemes will be developed. Specifications representing the inheritance relationship and the genericity of software components needs to be studied in order to exploit the properties of object-oriented development techniques. Our work provides a framework for a software reuse and retrieval system and we are investigating the integration of this system into a software development environment comprising tools for formal specification editing [30, 31], program visualization from formal specifications [32], and a tool that abstracts formal specifications from program code [33, 34].

References

1. Ted J. Biggerstaff. An Assessment and Analysis of Software Reuse. In Marshall C. Yovits, editor, *Advances in Computers*, volume 34, pages 1–57. 1992.
2. Ted J. Biggerstaff, editor. *Software Reusability Vol. 1: Concepts and Models.* ACM Press, New York, 1989.
3. Ted J. Biggerstaff, editor. *Software Reusability Vol. 2: Applications and Experience.* ACM Press, New York, 1989.
4. Charles W. Krueger. Software Reuse. *ACM Computing Surveys*, 24(2):131–183, June 1992.
5. 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.
6. 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.
7. Betty H.C. Cheng and Jun-Jang Jeng. Formal methods applied to reuse. In *Proceedings of the Fifth Workshop in Software Reuse*, 1992.
8. 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, December 1992.
9. R.L. London. Specifying Reusable Components Using Z: Realistic Sets and Dictionaries. *ACM SIGSOFT Software Engineering Notes*, 14(3):120–127, May 1989.
10. 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.
11. C. Rich and R.C. Waters. Formalizing Reusable Software Components. In *Proc. Workshop on Reusability in Programming*, pages 152–158, Newport, RI, September 1983.
12. B.W. Weide, W.F. Ogden, and S.H. Zweben. Reusable Software Components. *Advances in Computers*, 33:1–65, 1991.
13. Jeannette M. Wing. A Specifier's Introduction to Formal Methods. *IEEE Computer*, 23(9):8–24, September 1990.
14. G. Caldiera and V. Basili. Identifying and Qualifying Reusable Software Components. *IEEE Computer*, 24(2):61–70, Febrary 1991.
15. Barbara Liskov and John Guttag. *Abstraction and Specification in Program Development.* MIT Press and McGraw-Hill, Cambridge, 1986.
16. Rebecca Wirfs-Brock, Brian Wilkerson, and Lauren Wiener. *Designing Object-Oriented Software.* Prentice Hall, Englewood, New Jersey, 1990.
17. James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorensen. *Object-Oriented Modeling and Design.* Prentice Hall, Englewood Cliffs, New Jersey, 1991.
18. Peter Coad and Edward Yourdon. *Object-Oriented Analysis.* Yourdon Press, Prentice Hall, Englewood, New Jersey, 1990.
19. Bertrand Meyer. *Object-Oriented Software Construction.* Prentice Hall, Englewood, New Jersey, 1988.
20. Keith E. Gorlen, Sanford M. Orlow, and Perry S. Plexico. *Data Abstraction and Object-oriented Programming in C++.* John Wiley & Sons, 1990.
21. Ann L. Winblad, Samuel D. Edwards, and David R. King. *Object-Oriented Software.* Addison-Wesley, Publishing Company Inc., 1990.

22. Betty Hsiao-Chih Cheng. *Synthesis of Procedural and Data Abstractions.* PhD thesis, University of Illinois at Urbana-Champaign, 1304 West Springfield, Urbana, Illinois 61801, August 1990. Tech Report UIUCDCS-R-90-1631.
23. Stephen Bear. An Overview of HP-SL. Technical report, Hewlett Packard Laboratories, March 1991.
24. Chin-Liang Chang and Richard Char-Tung Lee. *Symbolic Logic and Mechanical Theorem Proving.* Academic Press, 1973.
25. J.A. Robinson. A Machine Oriented Logic Based on Resolution Principle. *Journal of ACM*, 12(1):227–234, 1965.
26. S. Miyamoto. *Fuzzy Sets in Informational Retrieval and Cluster Analysis.* Kluwer Academai Publishers, 1990.
27. D.H. Ballard and C.M. Brown. *Computer Vision.* Prentice-Hall, Englewood Cliffs, New Jersey, 1982.
28. J.B. Kruskal. On the shortest spanning subtree of a graph and the traveling salesman problem. In *Proc. of the American Mathematical Society*, 1956.
29. Douglas K. Pierce and Betty H.C. Cheng. Intelligent Browser for Formal Specifications of Software Components. Technical Report MSU-CPS-91-14, Department of Computer Science, Michigan State University, 1991.
30. Robert H. Bourdeau and Betty H.C. Cheng. An object-oriented toolkit for constructing specification editors. In *Proceedings of COMPSAC'92: Computer Software and Applications Conference*, pages 239–244, September 1992.
31. Michael R. Laux, Robert H. Bourdeau, and Betty H.C. Cheng. An integrated development environment for formal specifications. In *Proc. of IEEE International Conference on Software Engineering and Knowledge Engineering*, San Francisco, California, July 1993.
32. M. V. LaPolla, J. L. Sharnowski, B. H. C. Cheng, and K. Anderson. Data parallel program visualizations from formal specifications. *Journal of Parallel and Distributed Computing*, May 1993.
33. 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.
34. Gerald C. Gannod and Betty H.C. Cheng. A two-phase approach to reverse engineering using formal methods. In *Proc. of Formal Methods in Programming and Their Applications Conference*, June 1993.

This article was processed using the LaTeX macro package with LLNCS style