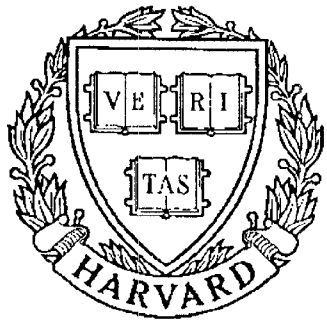
TECHNICAL RESEARCH REPORT

SYSTEMS RESEARCH CENTER



Supported by the National Science Foundation Engineering Research Center Program (NSFD CD 8803012), the University of Maryland, Harvard University, and Industry

Computing Similarity in a Reuse Library System: An AI-based Approach

by E.J. Ostertag, J.A. Hendler, R. Prieto-Díaz and C. Braun

TR 91-6

Computing Similarity in a Reuse Library System: An AI-based approach

Eduardo J. Ostertag
James A. Hendler
Computer Science Department
University of Maryland
College Park, MD 20742

Rubén Prieto-Díaz
Software Productivity Consortium
2214 Rock Hill Rd.
Herndon, VA 22070

Christine Braun
Contel Technology Center
15000 Conference Center Drive
Chantilly, VA 22021

Abstract

This paper presents an AI–based library system for software reuse, called AIRS, that allows a developer to browse a software library in search of components that best meet some stated requirement. A *component* is described by a set of (*feature,term*) pairs. A feature represents a classification criterion, and is defined by a set of related terms. AIRS also allows for the representation of *packages,* that is, logical units that group a set of related components. As with components, packages are described in terms of features. Unlike components, a package description includes a set of *member* components.

Candidate reuse components (and packages) are selected from the library based on the degree of similarity between their descriptions and a given target description. Similarity is quantified by a non-negative magnitude (called *distance*) that represents the expected effort required to obtain the target given a candidate. Distances are computed by functions called *comparators*. Three such functions are presented: the *subsumption,* the *closeness,* and the *package* comparators.

We present a formalization of the concepts on which the AIRS classification approach is based. The functionality of a prototype implementation of the AIRS system is illustrated by application to two different software libraries: a set of Ada packages for data structure manipulation, and a set of C components for use in Command, Control, and Information Systems. Finally, we discuss some of the ideas we are currently exploring to automate the construction of AIRS classification libraries.

1 Introduction

A major problem in software development is the need for greater productivity in the development process. Complex computer programs such as large communications network controllers and command and control systems place a growing demand on the talents of software programmers. This increasing need for complex computer systems has caused a significant gap between the code that can be developed using today's technology and the need for new software.

One aspect of the projected solution to this growing demand for new software is the development of a support technology which allows greater reuse of existing software components [1]. Rather than starting from scratch in new development efforts, an emphasis must be placed on using already available components. This approach avoids the duplication of work and lowers the overall development costs associated with the construction of new software applications.

The ability to reuse existing code requires four steps: *definition, retrieval, adaptation,* and *incorporation*. The definition step describes the component which needs to be constructed (the *target*) in terms of its functionality and relation to the rest of the environment. The retrieval step takes this description, and retrieves from the software library a list of components with similar characteristics (the *candidates*) and selects one of these. The adaptation step generates the target, usually through a modification process. The new component is then used in the application under development. In addition, the incorporation step takes newly constructed components and inserts them back into the software library. As new software components are created and added to the library, they in turn can be used in later development efforts.

This paper presents an AI–based library system for software reuse called AIRS. This system was originally designed to reuse Ada packages [10], but has evolved into a general tool for reuse. It allows a software developer to browse a software library in search of components that best approximate some design specifications. AIRS relies heavily on several AI related data structures, techniques, and algorithms [4]. The internal representation of the software library is constructed using a frame system with multiple inheritance [5], while the procedures used to find reuse candidates are based on A*-like search algorithms [12]. A highly interactive prototype of the system has been implemented using Common Lisp in a Macintosh computer [13].

2 The Retrieval Process

This section presents, by means of an example, a high-level overview of the main steps involved in the process of retrieving components using AIRS. This process selects from the AIRS library a list of candidate components that best approximate the required properties of a target component T, which is described informally as follows.

> Component T is required to print a spreadsheet on some printer. The spreadsheet is stored in computer A, and the printer is connected to computer B. Both computers are connected via a high speed network.

For this example, component T will be separated in two subcomponents: T_1, which transfers a spreadsheet from one computer to another, and T_2, which prints the spreadsheet. These informal descriptions are used as a basis to describe the functionality of each required component in terms of

a predefined set of features.

```
T1 = component                          T2 = component
       Function        = Transfer              Function        = Print
       ObjectType      = SpreadSheet           ObjectType      = SpreadSheet
       SourceType      = Computer-A            DeviceType      = Printer
       DestinationType = Computer-B            ControllerType  = Computer-B
     end component                           end component
```

AIRS selects the best reuse candidate component for each target description (e.g., T_1 and T_2). The candidates are selected based on the degree of similarity between the target and existing library component descriptions. Assume that T_1 and T_2 are not in the software library, but that T_1^* and T_2^* have been selected their best reuse candidates, respectively.

```
T1* = component                         T2* = component
       Function        = Copy                  Function        = Display
       ObjectType      = File                  ObjectType      = SpreadSheet
       SourceType      = Computer-A            DeviceType      = Terminal-V
       DestinationType = Computer-B            ControllerType  = Computer-B
     end component                           end component
```

The selected component T_1^* can "copy" (not "transfer") a "file" (not a "spreadsheet") between computers of different types. Component T_2^* can "display" (not "print") a spreadsheet on a "terminal" (not a "printer"). Each candidate component can be examined by either reading its documentation or obtaining implementation specifics. If it proves to be unsuitable, other candidates can be obtained by using alternative descriptions of the target.

The candidates selected (e.g., T_1^* and T_2^*) could be used directly to construct the required component T. An alternative would be to use a unit that grouped the functionalities of both candidates in a single common environment. This kind of unit is called a *package,* which is also described in terms of features. Unlike components however, package descriptions include a list of member components. Packages are stored in the AIRS library, and their descriptions can be compared for similarity based on their features and/or member sets.

In our example, T_1^* and T_2^* could be grouped together to define a target package — that is, it is hoped by the user that a single package could be found in the library which might help provide both functionalities. Assume there is no package in the library containing exactly T_1^* and T_2^*. AIRS could then suggest an alternative package which contains two functions T_1^+ and T_2^+ that are similar to T_1^* and T_2^*, respectively.

```
T1+ = component                         T2+ = component
       Function        = Copy                  Function        = Display
       ObjectType      = Matrix                ObjectType      = Matrix
       SourceType      = Matrix-File           DeviceType      = Terminal-V
       DestinationType = Matrix-File           ControllerType  = Computer-B
     end component                           end component
```

There are now two alternatives to construct component T. The user could try to group T_1^* and T_2^* into their own package (requiring the overhead of creating joint data structures, definitions, etc.),

or T_1^+ and T_2^+ which belong to the same package. This might be preferable as these components already belong to the same package and therefore presumably share certain properties (e.g., memory management or programming language) that may help construct T by reducing the effort required to merge both T_1^* and T_2^* into a single unit.

The use of such similarity-based reasoning to browse and choose through a library of components and packages is at the heart of the AIRS system. It is enabled by a frame-based knowledge representation and by the use of heuristic similarity computations, based loosely on techniques developed for case-based reasoning systems. In this paper we describe the similarity metrics and show that this AI approach can be used successfully in the design of browsers for software engineering components. First, however, we discuss some related work in software engineering and artificial intelligence.

3 Related Work

In a broad sense, software reuse can be defined as "the use of a software component more than once" [3]. A *software component* may be any product of the software development process — a unit of code, a design specification, a test case, etc. Use of a component more than once can mean anything from informal reuse of a design by its designer to the widespread use of a large software package such as a DBMS. The component can be used unchanged, or it can be modified to fit the new application.

Recent research focuses on exploring new directions aimed at formalizing and standardizing the activities and procedures necessary for reuse. Significant contributions have been reported in the areas of software cataloging and retrieval, program synthesis from reusable components, reuse measurement, and domain analysis. Research in these areas is laying the basis for development of tools and methods to make reuse practical and effective.

Cataloging and retrieval in software libraries has been one of the classical problems in reuse. Different classification approaches have been proposed for organizing software collections into software library systems to facilitate query and retrieval. The approaches reported are essentially of three types: free-text keywords, faceted index, and semantic-net based.

The CATALOG System of Frakes and Nejmeh [8] is an example of a free-text keyword based library system. CATALOG automatically extracts keywords from software documentation and creates an index by associating each item in the library with a set of keywords. It uses proven information retrieval (IR) and indexing technology. The retrieval mechanism takes a user-supplied set of keywords and attempts a match against index keyword sets. Frakes and Nejmeh report good performance when dealing with unambiguous sets of keywords. Two features that make this approach attractive are its simplicity and the fact that it is an automatic process. A limitation of this approach, however, is the lack of semantics associated with keywords. The meaning of a set of keywords can neither be inferred nor can these systems determine whether two different keywords represent the same concept. More recently, Maarek [11] has proposed the concept of lexical affinities among pairs of words as an effective means to extract some semantic information from software documentation. Results from her experimental system (GURU) demonstrate that this approach may be able to substantially improve keyword based library systems.

The faceted index approach proposed by Prieto-Diaz [16] relies on a predefined set of keywords extracted by experts from program descriptions and documentation. These keywords are

arranged by facets into a classification scheme and used as standard descriptors for software components. A thesaurus is derived for each facet to provide vocabulary control and to add a semantic component to retrieval. Keywords can only be used within the context of the facet they belong to and ambiguities are resolved through the thesaurus.

An important component of the faceted index approach is the use of a *conceptual distance graph*. Conceptual distances between items of each facet are used to evaluate their similarity, which is used in turn to evaluate the similarity between required software specifications and available components. Although reported to be very effective in retrieving software components for reuse the faceted approach is labor intensive. Construction of a conceptual graph, moreover, has not been formalized yet. Conceptual distances are assigned based on experience, intuition, and common sense. More recently, Gagliano et al [9], have proposed a method to compute conceptual closeness based on statistical analysis. Frequencies of "perceived similarity" obtained by running experiments with controlled groups of individuals are used to compute a "dissimilarity coefficient". This coefficient is then used to build conceptual distance graphs for a set of users.

An approach that borrows from both faceted and free-text method has been proposed by Embley and Woodfield [7]. They propose a library of abstract data-types (ADT) which are classified using special descriptors. A descriptor defines an ADT using keywords, facets, and list of aliases. The system allows the user to define explicit relations among different ADT's, and provides some built-in relations such as "depends-on", "close-to", and "generalizes" which can be derived automatically from the values of the ADT's, facets, and keywords.

Semantic-net based approaches have been proposed by Wood and Somerville [18] and by Devanbu, Brackman, and Selfridge [6]. These systems provide a structured representation of knowledge with some inferencing capability on semantics. Wood and Somerville propose a library system with a limited number of generic classes represented by simple semantic nets. Each semantic net provides a template-like structure to represent all possible instances of components of that class. By selecting one of these classes, the user can rapidly browse and retrieve components belonging to that class. Devanbu, Brachman,and Selfridge developed LaSSIE, a classification-based software information system. LaSSIE incorporates a large knowledge base, a semantic retrieval algorithm based on logical inference, a powerful user interface with a graphical browser, and a natural language parser. LaSSIE is intended to help programmers find useful information about large software systems like AT&T's System 75 PBX with over one million lines of C code. Semantic-based library systems are very powerful but share a similar disadvantage with the faceted approach. Creating a knowledge structure of any significant size is extremely labor intensive. Another disadvantage is rigidity. They usually support a very narrow application domain.

The AIRS system is essentially a hybridization between the faceted index and semantic network approaches. The domain information inherent in the facets is used largely to reduce the rigidity and the laborious creation of a semantic structure. A hierarchical *frame* system is used to maintain information about which of the objects in the reuse libraries have which features, how these objects are grouped, and how the features are related. Procedural attachment in the frame system is used to make the AIRS browsing system more efficient. In addition the features of the frame system are used to facilitate the integration of new components into the AIRS system, allowing a programmer to bootstrap its knowledge structures from a basic set of existing components.

The frame system used by AIRS is a modification of the system described in [5]. It is implemented in Common Lisp and is a separate module from the rest of the system — it maintains the AIRS knowledge base. Thus, details of the use of the frame system are not integral to an understanding of the functioning of AIRS and therefore in this paper we concentrate instead on the

techniques used in AIRS most relevant to software reuse in a classification library.

One important aspect of AIRS, discussed in some detail in this paper, is the ability to reason heuristically about the similarities between desired components and components residing in the existing knowledge-base (software library). In this respect, AIRS is similar to the case-based reasoning approach currently being explored as a solution to many AI problems[1]. In this approach, memory of previous solutions is used as heuristic knowledge to guide the processing of a novel problem. An important aspect of case-based reasoning systems is the ability to ascertain the information in memory most similar to the current situation. To retrieve relevant memories, these systems typically use a "domain theory" — that is, a knowledge-base of information about the particular situations the system is expected to handle. The domain theory is largely used to control the search in memory for information relevant to the situation at hand.

The AIRS system is similar to a case-based reasoner in that it uses a set of heuristic estimators (the distance computations described below) based on a knowledge-base of information about particular sets of components and their relations to derive an estimate of the similarity between components[2]. In the remainder of this paper, we describe the similarity metrics computed by AIRS and discuss how they are used in the process of software reuse. Examples from two domains, a set of ADA data structure packages and a command and control information system written in C, are also presented to demonstrate the efficacy of our approach. In addition, we discuss some directions we are currently exploring to facilitate the creation and update of AIRS knowledge bases.

4 The AIRS Classification Model

Software objects are described in AIRS based on *terms, features, components,* and *packages*. These descriptions can be compared for similarity using the concept of *distance*. This section presents a formalization of these concepts and their interrelations.

4.1 Features and Components

Features are the basic unit in the AIRS classification system. They are used to characterize the different aspects of a software component. Some examples of features are "functionality", "source language", and "required operating system environment". A feature is defined by a finite set of related values called *terms*. For example, a feature named `source-language` can be defined as the set of terms `{Pascal,Fortran,Ada,C}`. The set of features used to classify a collection of components within a certain domain defines a *feature space*.

A component is modeled by a finite set of (f,t) pairs, where f is a feature in a given feature space and t is a term of f. In other words, a component defines a mapping from features to terms. The notation *"A.f"* represents the term t associated to the feature f of component A. If a feature f is not relevant for describing a component A, then A maps f to a special value called the *null term*

[1] A review of the literature in case-based reasoning is beyond the scope of this paper. The interested reader is directed to the Proceedings of the DARPA workshops in case-based reasoning (1989 and 1990), which contain many papers on the subject.

[2] A departure of AIRS from most case-based systems is the numerical nature and efficient computation of these heuristic estimations.

(denoted by θ). For example, consider two features `source-language` and `project-name`. The description of the standard I/O function "printf" would map the feature `source-language` to the term `C`, but `project-name` would be mapped to θ because this function does not belong to any specific software project.

To compare components based on their descriptions, AIRS quantifies their degree of similarity by computing a *distance* between their corresponding descriptions based on two criteria: the *closeness* and *subsumption* relations. These are described in detail below.

4.2 The Closeness Relation

The *closeness relation* is intended to capture the idea that a component can be constructed by modifying another component. If a component *B* can be constructed by modifying certain portions of a component *A*, we consider *A* to be a suitable reuse candidate for *B* if the extent of these modifications is small. In general, it is impossible to determine if a component has been constructed by modifying another one, much less to determine the nature and size of such a modification. However, it has been argued [14][16] that a good heuristic for determining the differences between two components is to consider the differences between their corresponding terms.

Distance between Feature Terms

To estimate the difference between terms of a feature, we arrange the terms in a weighted directed graph called a *feature graph,* where each node corresponds to a particular term. The weight w associated to an arc connecting a node t_1 to a node t_2 is a non-negative magnitude that represent the expected effort required to obtain the target term t_2 given the candidate term t_1.

A feature graph also includes an additional node for the null term θ, and all other terms in the graph are connected to θ with an associated weight of zero. The rational behind this decision is that when the null term is used as a target we want it to represent the "don't care" value. On the other hand, the distance from the null term to a term t in the graph may be greater than zero. This distance is called the *creation distance* of term t, meaning the expected effort to obtain t from scratch.

Some pairs of nodes in a feature graph may not be connected by an arc, meaning there is no known method to directly obtain one from the other. Yet, to compare components we need a method to estimate distances between all possible pairs of terms. We define the distance from two different terms t_1 and t_2 as the weight of the shortest path from t_1 to t_2 in the graph. If no such path exists, the distance is set to be infinity (denoted by ∞). If t_1 and t_2 are the same, the distance is zero.

For example, consider the feature language = {assembler,pascal,common-lisp} and its associated graph, shown in Figure 1. According to this graph, the distance from assembler to pascal is 5, while the distance from `pascal` to `common-lisp` is 10, corresponding to the intuition that transforming an assembler source file to pascal might be easier than recoding pascal to common-lisp. The creation distance for `assembler` is 30, given by the arc connecting θ to `assembler`. The creation distance for `pascal` and `common-lisp` is 35 and 45 respectively, given by the weight of the shortest path from θ. Similarly, the distance from `pascal` to `assembler` is 30, since the path `pascal` → θ → `assembler` yields the smallest distance from pascal to assembler.

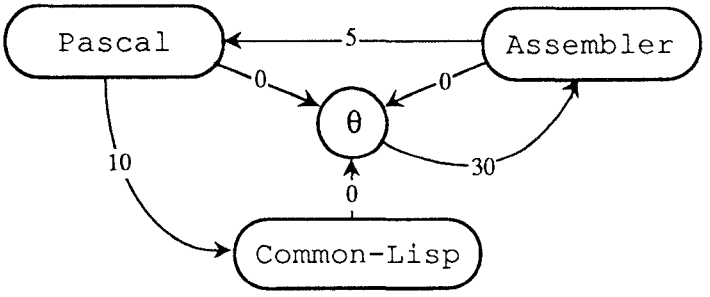


Figure 1: Sample feature graph

Closeness Distance between Components

We say that a component A is *close* to a component B if the overall difference between their corresponding terms is small. This difference is quantified by a non-negative magnitude called *closeness distance,* which represents the expected effort required to obtain component B given component A. This distance is computed as the sum of the efforts required to obtain each feature term of B given the corresponding feature term of A. That is, the closeness distance from target component A to a candidate component B is given by the following expression.

$$d_C(A,B) = \sum_{f \in \Psi} d_f(A.f, B.f)$$

where Ψ is a feature space, and $d_f(t_1,t_2)$ is the distance from term t_1 to term t_2 as defined by the feature graph of f. Function d_C is called *closeness comparator,* and the functions d_f are called *feature comparators.*

For example, consider the following two component descriptions: `pop` removes an element from the head of a structure, while `append` inserts a set of elements at the tail of a structure.

```
pop = component                    append = component
          operation  = remove                  operation  = insert
          object     = element                 object     = element-set
          position   = head                    position   = tail
      end component                      end component
```

The closeness distance from `pop` to `append` is given by the following expression which computes the sum of the distances between their corresponding feature terms.

```
closeness-distance(pop,append) = operation-comparator(remove,insert)     +
                                 object-comparator(element,element-set) +
                                 position-comparator(head,tail)
```

where `operation-comparator`, `object-comparator`, and `position-comparator` are the feature comparator functions of features `operation`, `object`, and `position` respectively. These functions are computed using their corresponding feature graphs.

4.3 The Subsumption Relation

The *subsumption relation* is intended to capture the idea that certain components can be built by composing several other components. If the functionality of a component A is partially provided by a component B, then B (the *subsumer*) is considered to be a suitable reuse candidate to construct A (the *subsumee*). As opposed to the closeness relation, the subsumer is used to construct the subsumee as a subfunction without modification. For example, consider the abstract data types *stacks* and *lists*. The stack operation *push* is subsumed by the list operation *cons*, because push can be constructed using cons as a subfunction.

We represent the subsumption relation between components using a weighted directed acyclic graph (DAG) called *subsumer graph* (there is one subsumer graph for each software library.) Each component is represented by a node in the graph, and an arc indicates that the source node subsumes the destination node. The weight w of an arc is a non-negative magnitude that represents the expected effort required to obtain the *subsumee* given the *subsumer*.

In general, we define the *subsumer distance* from a component c_1 to a component c_2 as the weight of the shortest path from c_1 to c_2 in the subsumer graph. If this path does not exist, the distance is infinity, meaning c_2 is not subsumed by c_1. If c_1 and c_2 are the same, the distance is zero.

For example, consider the subsumer graph shown in Figure 2. According to this graph, component `append-element` can be used to implement (subsumes) component `push` with an expected effort (subsumer distance) of 7. Component `append-list` also subsumes `push` with a subsumer distance of 9.

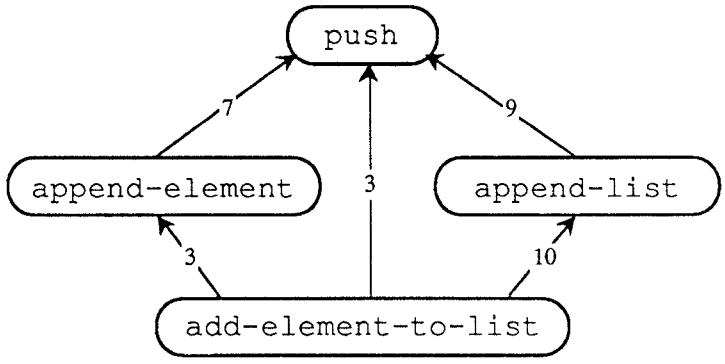


Figure 2: Sample subsumer graph

We denote by $d_S(c_1,c_2)$ the subsumer distance from component c_1 to component c_2 as defined by the subsumer graph. This function is called the *subsumer comparator*.

4.4 Packages

Packages are like components in the sense that they can be described in terms of features, but while a component represents a particular object, a package represents a collection of components which are tightly coupled, that is, each component in the collection is defined to be used in conjunction with the others. This set of components are called *members* of the package. For example, consider an abstract data type *list*, which has been implemented in Pascal using arrays of fixed size to store the elements. This software object could be described in AIRS as follows.

```
list = package
          source-language = Pascal
          maximum-size    = Bounded
          member-set      = {Head,Tail,Cons,Append,Length}
       end  package
```

The names `head`, `tail`, `cons`, `append`, and `length` are not terms, but the names of components that must be defined separately as explained in Section 4.1. As with components, the description of a package defines a mapping from features to terms. Unlike components, a package is defined using a special feature named "member-set", which is mapped to the set of member components of the package.

Distance between Packages

A candidate package p_1 is said to be similar to a target package p_2 if the overall difference between their features and member sets is small. This difference is quantified by a non-negative magnitude called *package distance,* which represents the expected effort required to obtain p_2 given p_1. This distance is computed as the sum of the distances between their corresponding feature terms (as defined by their feature graphs), plus the distance between their member sets.

The distance from a member sets m_1 to a member set m_2 represents the expected effort required to obtain each component in m_2 given the components in m_1. This distance is computed by mapping each component in m_2 to its best reuse candidate component in m_1 as defined by either the closeness of subsumption relation, and then summing up the distances between these pairs of components.

In summary, the distance from a candidate package p_1 to a target package p_2 is defined by the following function called *package comparator.*

$$d_p(p_1,p_2) = \sum_{f\in\Psi} d_f(p_1.f, p_2.f) + \sum_{c_2\in m_2} \min_{c_1\in m_1} d(c_1,c_2)$$

where Ψ denotes the feature space, d_f is the feature comparator of feature f, m_1 and m_2 are the member sets of p_1 and p_2 respectively, and $d(c_1,c_2)$ is a user-selected component comparator function, namely the subsumer comparator (d_S) or the closeness comparator (d_C).

5 Examples of the use of AIRS

A prototype of the AIRS classification system has been implemented using Common Lisp in a Macintosh computer. This prototype has been used to classify two different software libraries. The first is the EVB GRACE library (part I) developed by EVB corporation, which contains a collection of Ada packages that implement data structures such as *stacks* and un*directed graphs*. The second is the Contel CCIS software library developed at Contel Technology Center, which contains a collection of C modules for implementing the basic functionalities of Command, Control, and Information Systems. The GRACE application demonstrates AIRS' applicability to a highly-structured, dense collection. The CCIS application shows that AIRS also works with a

more loosely-structured sparse collection more representative of those arising in industry.

In general, to create a software library for reuse it is necessary to perform a *Domain Analysis,* defined by Prieto-Díaz [15] as the process of identifying, collecting, organizing, analyzing, and representing a domain model and software architecture from the study of existing systems, underlying theory, emerging technology, and development histories within the domain of interest. This analysis is important for software classification because it provides (1) a software architecture describing the different relations among the components of the architecture, and (2) a dictionary where the components are defined. These two elements are the basis for software classification processes.

In the particular case of an AIRS software library, the process of domain analysis includes defining the features and terms to describe both packages and components. The programmer also has to define the necessary feature and subsumer graphs. This allows AIRS to select from its library candidate packages and components for reuse based on their degree of similarity to a user-supplied target description (as described above).

5.1 Classification of the GRACE package

The EVB GRACE library (part I) is a collection of generic Ada packages that implement the following data structures: binary search trees, circular doubly linked lists, doubly linked lists, singly linked lists, queues, stacks, and undirected graphs. This library contains several Ada packages that implement the same conceptual data structure, but differ on their *functional* behaviour, as well as *time* and *space* characteristics. These differences are classified based on a subset of Booch's taxonomy [2]. The features used are *control, manager, allocation* and *iterator.* For example, the following is a description of an Ada package which implements an unbounded size binary tree using a user-supplied memory management system, and which provides both concurrent access to its elements and a function for traversing the structure.

```
binary-tree = package
                 allocation   = Unbounded
                 manager      = User
                 control      = Concurrent
                 iterator     = Supplied
                 member-set = {depth-first, insert, retrieve, ...}
              end package
```

Member components such as `depth-first`, `insert`, and `retrieve` are classified using the following set of features.

- **optype** = `{insert,remove,create,select,traverse}`: kind of action performed by the operation either to the data structure or to its elements. The term `create` is used to describe operations that initialize structures. The term `select` is used for operations that read or modify the state of an element or the structure.

- **count** = `{zero,one,two,several,all}`: number of elements produced, consumed, or selected by the operation. The term `several` is used to represent an unknown number of elements between zero and the size of the structure.

- **type** = {element, link, structure}: kind of element produced, consumed, or selected by the operation. The term structure is used when the operation acts on both elements and links at the same time.

- **position** = {first, second, rest, last, root, keyed}: position of an element or set of elements relative to which the operation is applied. The term keyed is used when the position of an element is implicitly defined by the feature key.

- **key** = {pointer, index, field-value, fixed}: key type used by the operation to determine the position of an element or set of elements. The term fixed is used when the position of an element is explicitly defined by the feature position.

- **direction** = {left, right, breadth-first, depth-first, self}: direction of application of the operation relative to the position of an element. The term self is used when the operation acts directly on an element whose position is defined either by the position or key features.

Whenever a feature is not relevant for the description of a component, it is mapped to the null term. For readability, this term has been given different names such as no-key and no-type. For example, the following is the description of an "append" operation of a "singly linked list" data structure.

```
append = component
             optype    = Insert
             count     = One
             type      = Element
             direction = Right
             position  = Last
             key       = No-key
         end component
```

This component description characterizes an operation which "*inserts one element* to the *right* of the *last* position of a structure. No *key* value is required to perform this action."

The last phase in the design of the AIRS classification scheme for EVB GRACE library was the definition of the feature and subsumer graphs. The construction of these graphs requires access both to the documentation and implementation of the components and packages being classified. In this case, we only had access to the documentation of the library, so to obtain these graphs we had to complement this information with our own experience and intuition. The process of constructing feature and subsumer graphs is a knowledge intensive operation which must be performed by an expert analyst. We are currently exploring an approach to create these graphs in terms of a semi-automatic, user-guided process (see section 6).

Using the AIRS system to retrieve reusable EVB GRACE components

The AIRS browser [13] is a highly interactive tool which allows to search for software components that satisfy some stated requirements. This section presents a sample user/system

interaction to help get a better understanding of how AIRS is used and the kind of information it provides. Assume a software developer is interested in finding a data structure Ada package capable of performing the following operations.

- OPER1: insert an element at the front of a structure.
- OPER2: select an element based on a given key value.
- OPER3: remove an element given a pointer to the element.

To accomplish this task, the developer uses AIRS to perform the following steps: (1) describe each target operation (i.e., OPER1, OPER2, and OPER3) in terms of the set of features explained in the previous section. (2) for each target select a candidate operation from the library. (3) describe the target (desired) package using the selected candidates and a set of package features. (4) retrieve the best reuse candidate for the target package. What follows is a description of how these steps are performed using the AIRS library browser.

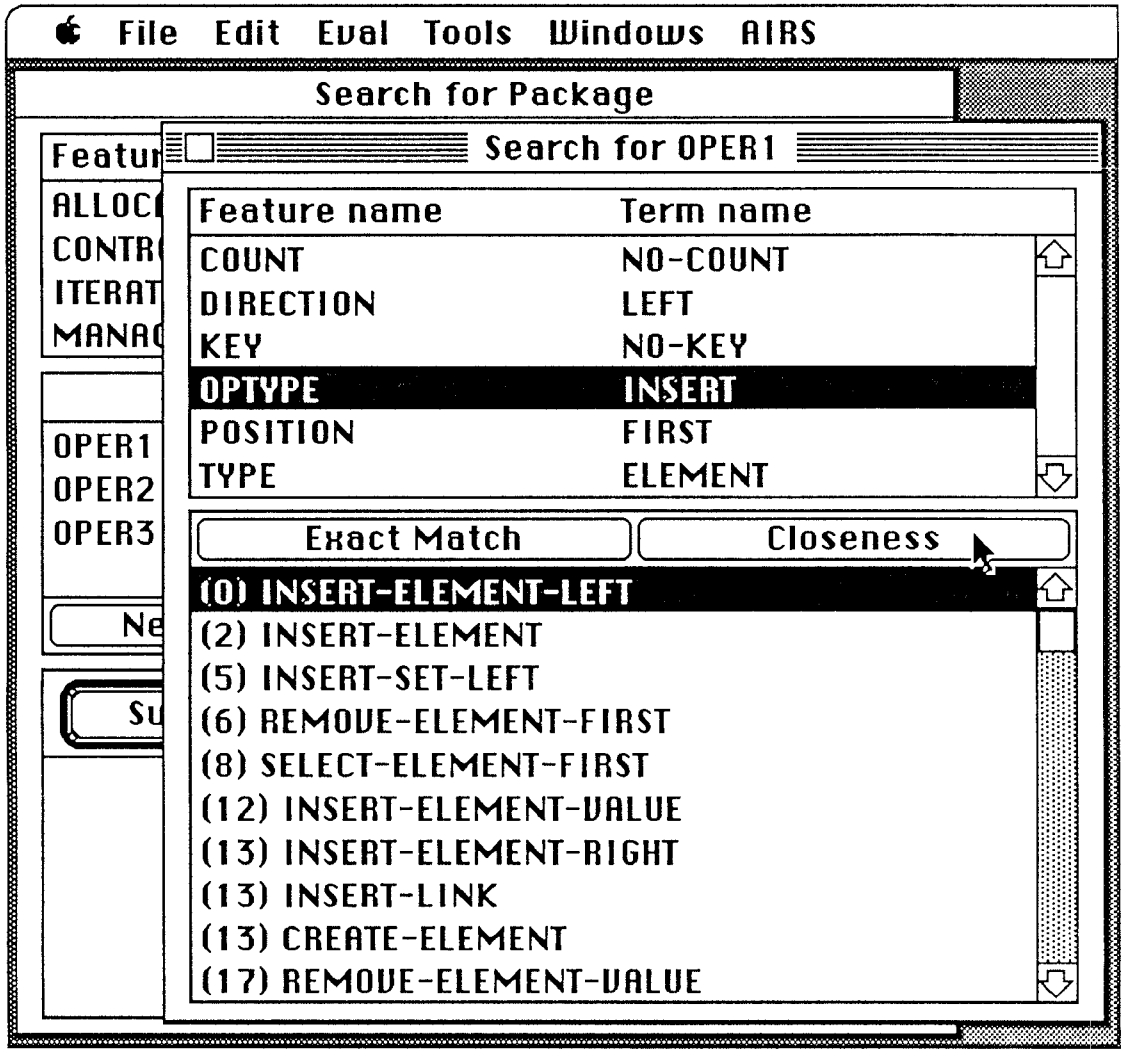


Figure 3: Describing an operation using the feature terms.

The front window of Figure 3 (Search for OPER1) is used to define the feature terms of OPER1 (Insert, No-Count, No-Key, First, Left, and Element). This window can also be used to obtain the list of component names which match exactly the given terms[3], or to obtain a list of reuse candidates based on the closeness relation. In this case, the closeness relation option was selected producing a list of class names and their associated closeness distance (shown in parenthesis to left of the name). Each name in this list can be selected (like INSERT-ELEMENT-LEFT) to obtain detailed information about the class (Figure 4). Note the distance from INSERT-ELEMENT-LEFT to OPER1 is zero, meaning the description of OPER1 matches exactly the descriptions of the components of this class.

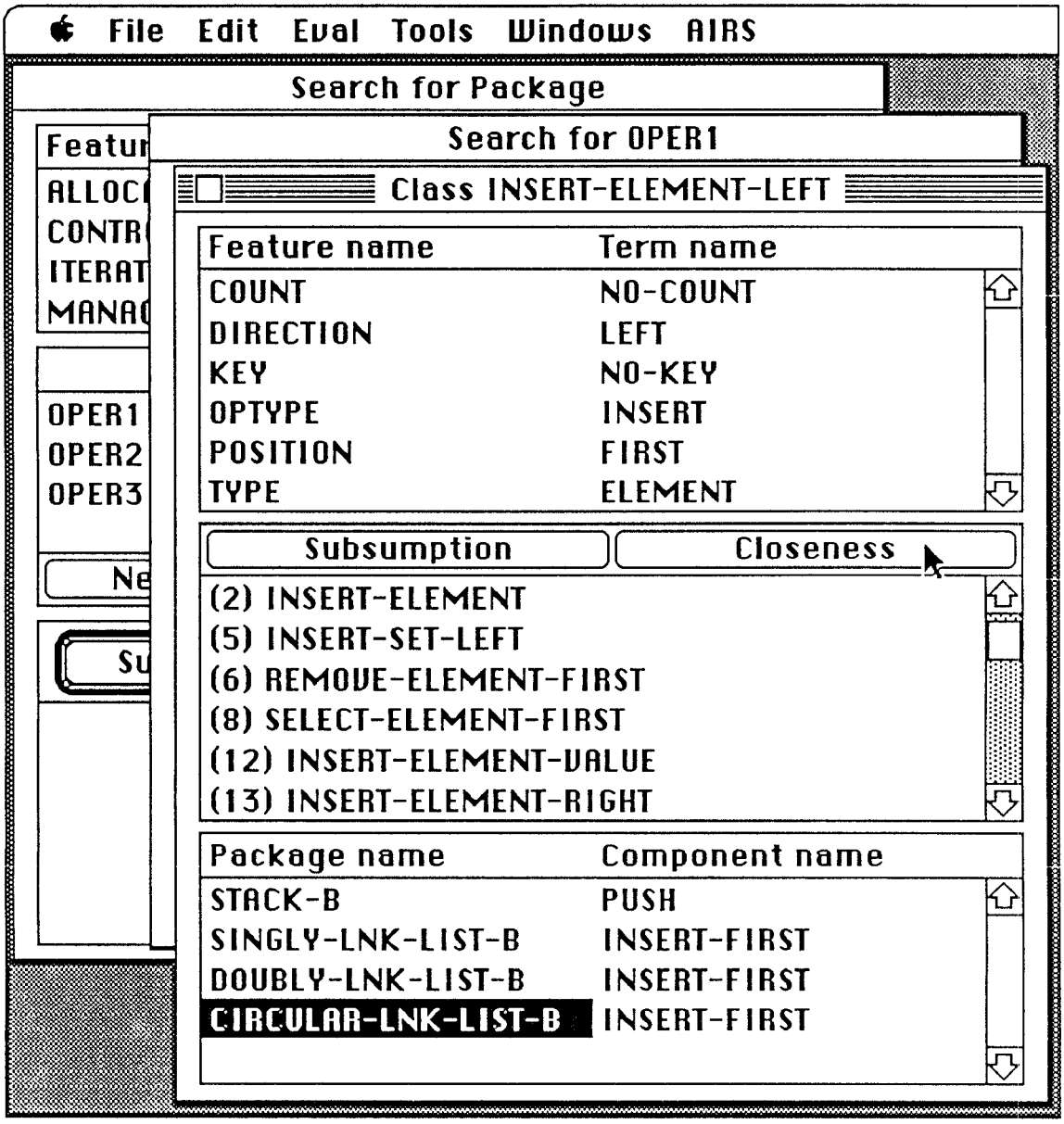


Figure 4: Information of a class of components in the AIRS library.

[3] The set of components described using the same set of feature terms is called a *class*.

The topmost window of Figure 4 gives detailed information about the class INSERT-ELEMENT-LEFT, which includes the list of components that belong to the class and the set of features they share. This window can also be used to obtain a list of reuse candidates based on either the subsumption or the closeness relations (in this case closeness was used). The list of components of the class (bottom) is divided in two columns: name of a component (right), and the name of the package (left) that contains the component. If a component name is selected, its Ada code or documentation can be obtained (not shown). If a package name like CIRCULAR-LNK-LIST-B is selected, its features and member set can be obtained (Figure 5).

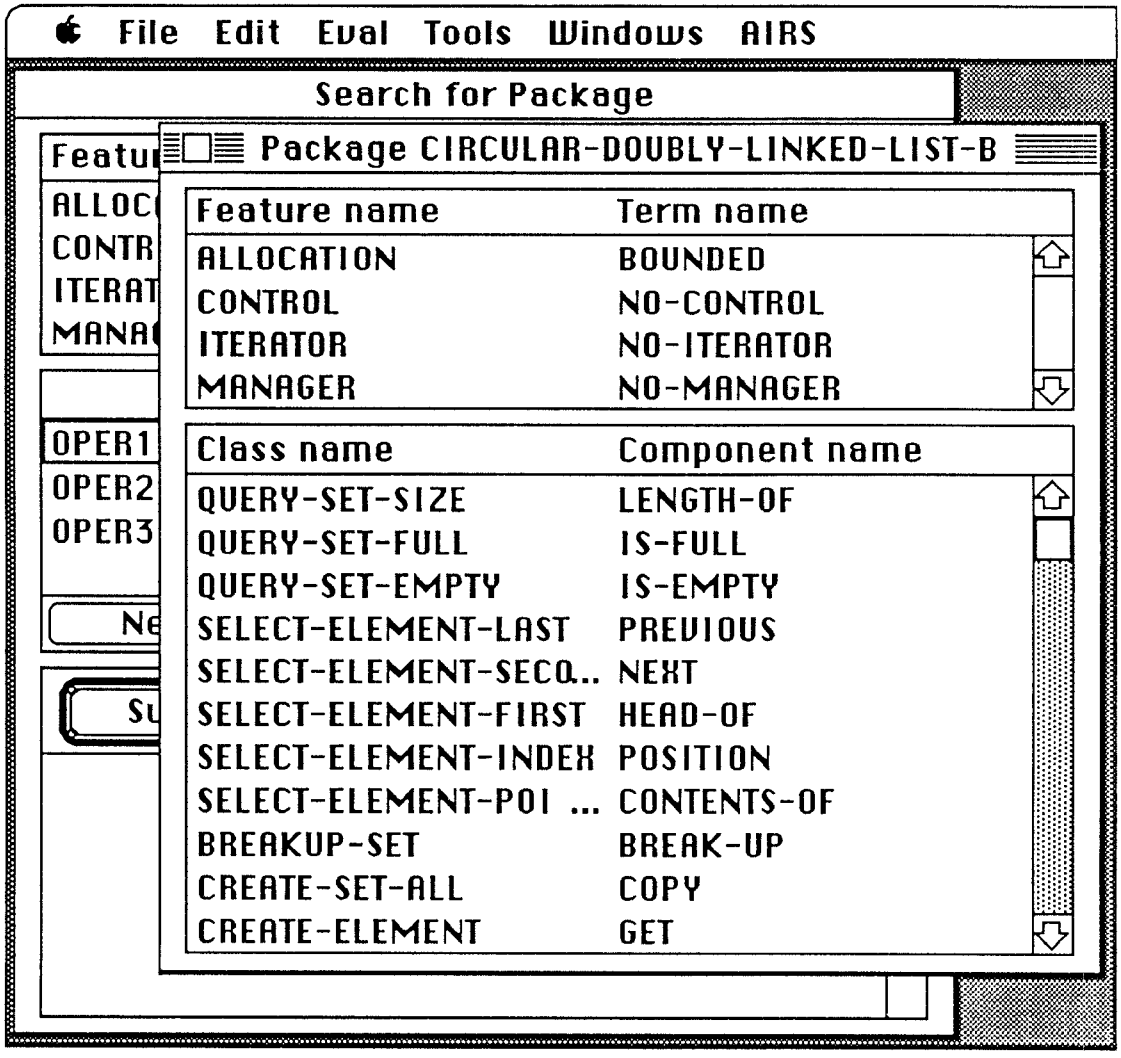


Figure 5: Information of a package in the AIRS library.

The front window of Figure 5 shows information associated with the Ada package CIRCULAR-LINKED-LIST-B, which includes its feature terms (No-Control, No-Manager, Bounded, and No-Iterator), and a list of its member components. This list shows the name of each component (right) and the name of the class (left) to which it belongs. If a component name is selected, its Ada code or documentation can be obtained (not shown). If a class name is selected, the user obtains a window like the one shown in Figure 4.

The window shown in Figure 6 is used to define both the features of the required package (BOUNDED) and its member set (OPER1, OPER2, and OPER3). Once each of the operations in its member set have been defined (Figure 3), the user can obtain a list of candidate packages based on its features, its member set, or both. In this case, the list of packages (bottom) was obtained using both criteria. Features with an associated term named "Any" are not considered in the computation of package distances. The resulting list shows that the best alternative for implementing the required package is using a bounded SINGLY-LINKED-LIST data structure.

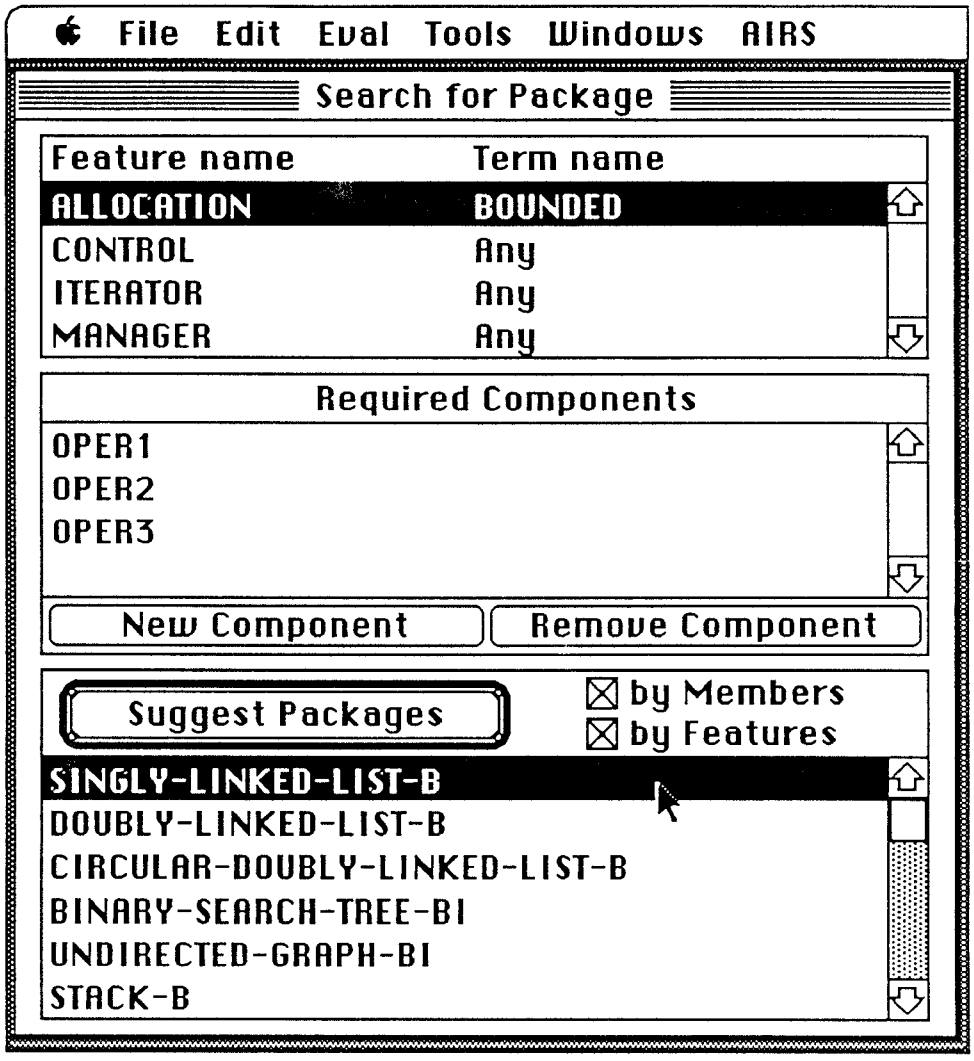


Figure 6: Obtaining a candidate reuse package.

5.2 Classification of the Contel CCIS library

The construction of a reuse library for the EVB GRACE collection shows the applicability of AIRS to a highly-structured, dense collection of software components. To test the effectiveness of AIRS on a software collection more representative of those arising in industry, we decided to use this technique on the Contel CCIS library developed at Contel Technology Center (CTC). This library contains a collection of C packages for implementing the basic functionalities of a Command and

Control Information System.

To improve reusability of the components of the CCIS library, CTC developed an interactive reuse browser based on Prieto-Díaz's faceted scheme [16]. In this system, both source files[4] and functions were classified according to three facets: **area**, **function**, and **object**. Conceptual distance graphs were not supported, so components could not be retrieved based on their degree of similarity but solely on partial matches among facet terms. A thesaurus allowed a user to retrieve components described using term synonyms.

To test the AIRS system we built an AIRS knowledge-base for the CCIS library using as a basis both the documentation of the library and the CTC taxonomy (defined by the facets, terms, and synonyms used to construct the CTC browser). The resulting AIRS browser took one person less than a week to build, and provided all the functionality of the CTC browser plus those unique to AIRS, namely: (1) to represent packages, (2) to retrieve components based on the closeness and subsumption relations, and (3) to retrieve packages based on the package relation.

Description of the CCIS library and the CTC taxonomy

The following is a description of the different modules that compose the CCIS library.

- **general** (GEN): collection of general purpose functions that do not belong to any specific module. These functions are typically extensions to the ones contained in the standard C library.

- **memory file** (MF): collection of functions that implement sequential files allocated in main memory (RAM). These files are created and exist only during the execution of a program.

- **set structure** (SET): collection of functions that implement unbounded sets of elements. The elements of a particular set must be of the same type.

- **database interface** (IDB): collection of functions that provide a simplified interface to the most commonly used operations of a relational database system.

- **database file** (DBF): collection of functions that implement database files. These files are flat structures stored in a relational database processor.

- **mail service** (MS): collection of functions that implement the basic functionalities of an electronic mail system.

- **man-machine interface** (MMI): collection of functions that implement user/system interfaces based on windows, predefined keys, and menus.

- **free text file** (FTF): collection of functions that implement text files. These files are regular text files but are stored on a relational database processor.

- **parametric database display** (PPD): collection of parametric functions to retrieve and display information contained in a relational database.

[4] A source file name is used to represent the collection of C functions contained in the file.

Some of the C source files that compose these modules and all their functions were classified at CTC according to three facets: **area**, **function**, and **object**. The following is a description of these facets and their corresponding terms.

- **area**: general problem area the components belongs. Each of the modules previously defined presents a solution to well defined problem, therefore module names are used as synonyms of problem areas. The terms of this feature are the following.

• database-interface	• database	• free-text-file
• general	• mail-service	• memory-file
• man-machine-interface	• parametric-database-display	• set

- **function**: to discriminate among the different operations that belong to a same problem area in terms of its functionality. The list of terms of this facet follows. The functionality represented by each term is that which is normally associated with its name.

• add	• assign	• clear	• close	• convert	• copy	• count
• create	• delete	• display	• enable	• execute	• find	• goto
• intersect	• log	• map	• measure	• modify	• open	• parse
• process	• query	• read	• rename	• replace	• retrieve	• search
• suspend	• terminate	• test	• transfer	• union	• write	

- **object**: describe the kind of input used by the operation or the kind of output generated by the operation, which ever is more relevant for the operation. The terms of this facet follows. The object represented by each term is that which is normally associated with its name.

• SQL-command	• address	• code	• column-type	• column
• control-variable	• descriptor	• directory	• element	• event
• file	• function-key	• group	• interface	• keyboard
• menu	• name	• offset	• owner	• pdd-page
• pdd-descriptor	• ppd-table	• permission	• pointer	• printer
• queue-entry	• queue	• record	• set	• string
• subset	• substring	• list	• text	• tuple

Each of the terms of a facet has an associated list of synonyms names which can be used to query the CTC library. This thesaurus facility allows users of the CTC browser to describe the required components using the terminology they are most familiar with.

Construction of the AIRS browser

Component descriptions used in the CTC browser were mapped almost identically to the AIRS library, with the exception of the `area` facet. Since each term in this facet represents a module in the CCIS library, they were mapped to packages in the AIRS library, so instead of describing components using the `area` facet, they became members of a package. For example, consider the

CTC description of the CCIS component `strstr` which returns the location of a string within a string: `[area=general,function=find,object=substring]`. This description was mapped to the AIRS system as follows.

```
general = package                          strstr = component
            member-set = {strstr, ...}             function = find
          end package                              object   = substring
                                                 end component
```

The next step in the construction of the AIRS library requires the definition of feature and subsumer graphs. Based on our intuition and experience, each term *t* of a feature *f* was assigned distances to a small set of terms in *f* that are considered similar to *t*. These assignments defined *groups* of closely related terms. For example, figure 7 shows the groups associated with the terms `copy`, `assign`, and `add`.

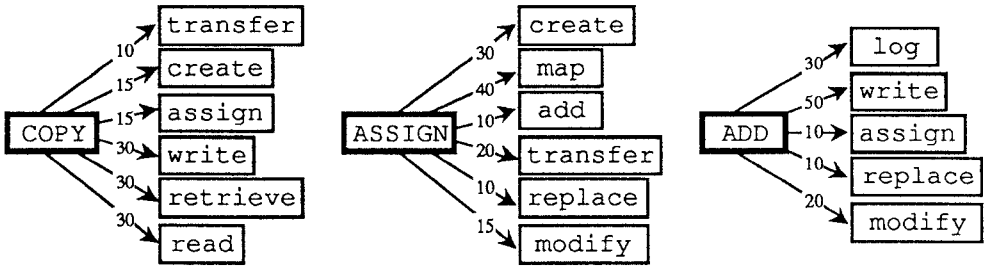


Figure 7: Defining groups of closely related terms.

When these groups are integrated in the AIRS library, they define a combined feature graph which allows the user to compare terms that are not related by a group. Figure 8 shows a partial feature graph defined by the three groups of figure 7. Based on this graph, for example, we could estimate the degree of similarity between `copy` and `log` as the weight of the path `copy` → `assign` → `add` → `log` for a weight of 55.

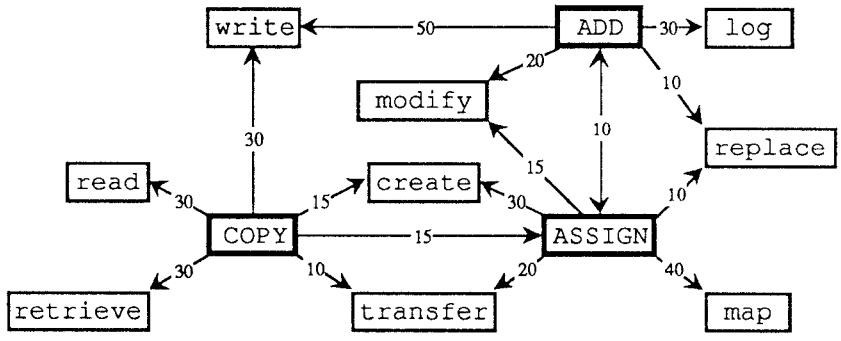


Figure 8: Part of the feature graph for feature `function`.

To handle synonyms, we set the distances between a term and its synonyms to zero (not shown in Figure 8). This has the effect that the closeness distance from a candidate component *A* described using terms which are synonyms to the ones used to describe a target component *B* will be zero,

therefore making *A* the best reuse candidate for *B*. That is, *A* and *B* become synonym descriptions.

To complete the definition of components, their documentation and source code was included in the AIRS library. This allows users of the AIRS browser to obtain specific implementation details of components (see Figure 9).

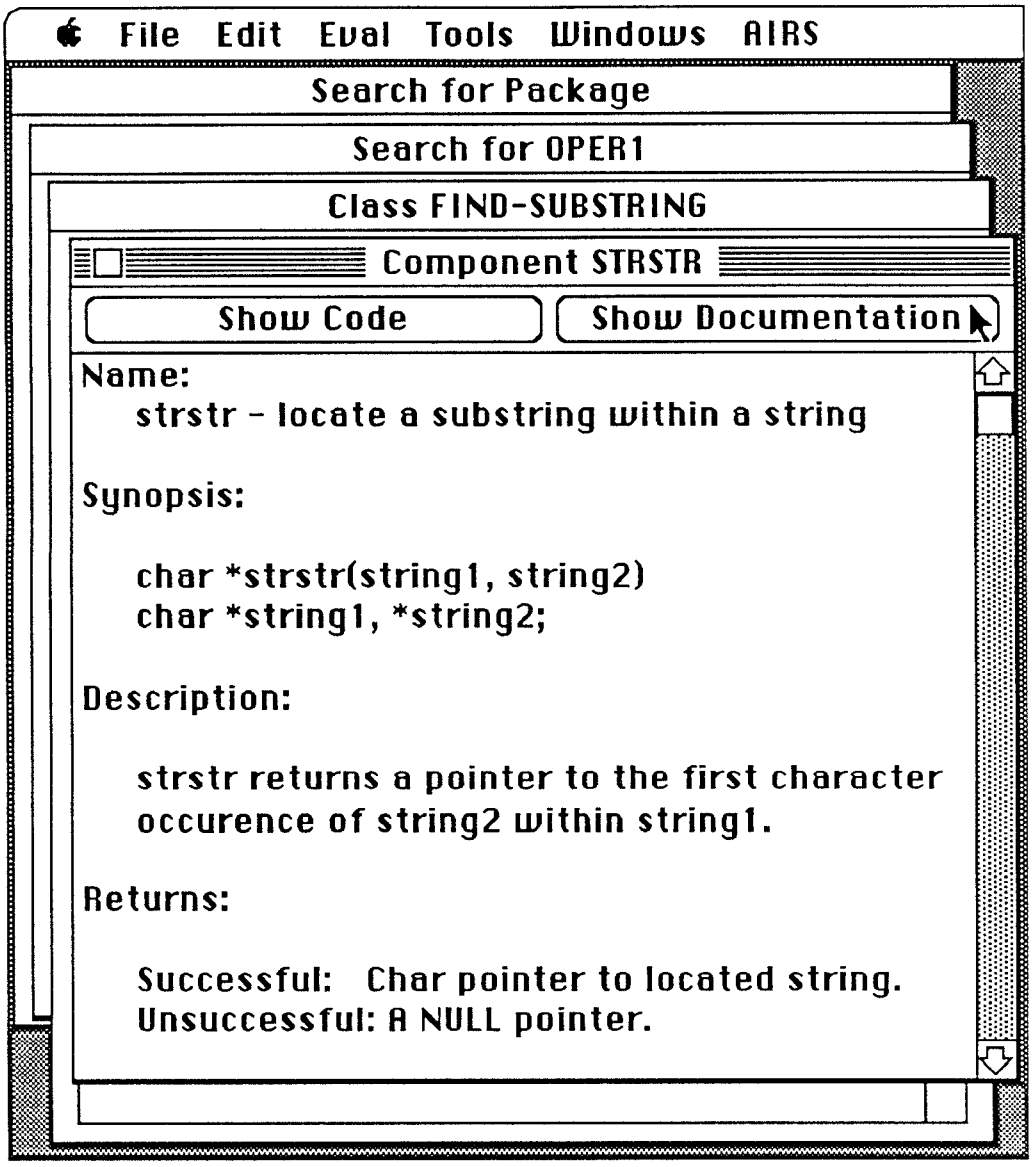


Figure 9: Documentation of the CCIS component `strstr`.

6 Future Research Directions

The classification model described in the previous sections show how the AIRS approach can be used to retrieve components for reuse, but little has been said of how to construct a classification library for a given domain. Currently, this is a knowledge intensive operation which must be performed by an expert analyst. In this section we present an overview of some of the methods we are currently exploring to further automate this construction.

Semi-Automatic Classification

A method is needed to classify components and packages in terms of a feature space. In general, this involves analysis of the different parts of a component (e.g., source code, documentation, etc.), and the use of heuristics to extract features based on this analysis. We are currently examining a heuristic method called *feature aggregation,* which extracts information from the source code implementation of a component *A* to predict its features. The source code of *A* is scanned to obtain the list of component names used as subfunctions in its implementation. For example, consider the following implementation of the function "minimum of a set of numbers".

```
min(list-of-numbers) = first-element(sort-ascending(list-of-numbers))
```

Function "min" is implemented by sorting in increasing order of magnitude the list of numbers it receives as argument, and then returning the first element of the sorted list. This implementation uses a set of two subfunctions to accomplish its task, namely `first-element` and `sort-ascending`. This set is called *reference set.* Feature aggregation is defined as the method for predicting the features of a yet unclassified component *A* with a reference set *B,* by using the feature descriptions of those components in *B* which are stored in the AIRS library.

Refining Subsumer and Feature Graphs

A method is needed to test whether the reuse candidates proposed by the system are truly the best ones available in the software library. For example, if we classify a new component *A* known to be similar to a previously classified component *B*, we would expect the library system to propose *B* as a reuse candidate for *A*. Failure to do this could arise due to errors in classification of components *A* or *B*, or because of errors in the definitions of feature and subsumer graphs. If both *A* and *B* are classified correctly, then we need to adjust both the links and weights of the graphs used to compute their degree of similarity (i.e., distance).

We are currently examining a semi-automatic refinement process which will modify the links and weights of these graphs based on feedback provided by the users of the software library. This process works as follows: Let *T* be the description of a test component, and *E* the description of a component in the software library known to be *T's* best reuse candidate. The AIRS system is used to retrieve a proposed candidate *P* for *T*. If the distance from the proposed candidate (*P*) to the expected candidate (*E*) is greater than a certain value *K,* the descriptions of *T, E,* and *P* are used to adjust the feature and subsumer graphs. A value K of zero forces *P* to be equal to *E*. In general, *K's* value may need to be greater than zero to avoid undoing adjustments of the graphs done for other test components.

Expanding the Feature Space

One obvious requirement for any library system is the ability to incorporate components which have little or no relation to the ones currently stored in the library [17]. Classification of these new components may include adding new terms to an existing feature, as well as adding new features to the feature space. A restriction currently imposed by AIRS is that components must be described in

terms of all features in the feature space, therefore adding new features would involve modifying all the components already stored in the library.

We are currently examining a method to automate the process of expanding a feature space. This method uses a new concept called *expansion rules*. For example, consider a feature space Ψ (defined by two features f_1 and f_2) which needs to be expanded by adding a new feature f_3. A rule for this expansion would be defined as follows.

```
rule-name = component
                f1 =  t1
                f2 =  t2
                f3 → t3
            end component
```

The description of a rule is similar to that of a component in the sense it defines mappings from features to terms (denoted with a =), therefore we can compute distances from components to rules using the closeness relation. Unlike components, a rule is defined using one or more *projections* (denoted with a $\rightarrow$). To determine the classification of a component C in Ψ in terms of the new feature f_3, we find the rule R whose distance from C is smallest, and then use $R's$ projections to expand $C's$ definition. In this case, C would map f_3 to t_3.

7 Conclusions

We have presented AIRS, an AI-based classification system for software reuse. We argue that such a system is an important aspect of general problem of software reuse and that it has the potential to reduce the cost of software development. This system uses a classification model to describe software objects based on three different concepts: *features, components,* and *packages*. These descriptions plus the concepts of *subsumption* and *closeness* are used to select candidate components from the library based on their degree of similarity to a given target component. Similarity is quantified by a non-negative magnitude (distance) representing the expected effort required to obtain the target given a candidate. We have demonstrated AIRS using two different software libraries, and presented an overview of our current research on methods to automate the construction of AIRS software libraries.

Acknowledgements

Funding for this work was provided by Contel Corporation in a grant awarded through the University of Maryland Foundation and the UM Systems Research Center. Additional support was provided to Eduardo Ostertag by Orden S.A., Chile. Yu Chung Wong, A. Vinciguera, and J. Mogilensky participated in the first design of AIRS. We are also grateful to Pablo Straub whose comments helped formalize the concepts underlying the system.

References

[1] T.J. Biggerstaff and C. Richter. Reusability framework, assessment, and directions. *IEEE Software,* 4(2), March, 1987. Also in T.J. Biggerstaff and A.J. Perlis, eds., *Software Reusability,* Volume I, ACM Press, 1990.

[2] G. Booch. *Software Components with Ada: Structures, Tools, and Subsystems.* The Benjamin/Cummins Company, Inc., Menlo Park, California, 1987.

[3] C.L. Braun and A.B. Salisbury. Software Reuse in Command and Control Systems. Technical Report, Contel Technology Center, Chantilly, Virginia, 1990.

[4] E. Charniak and D. McDermott. *Introduction to Artificial Intelligence.* Addison-Wesley, Reading, Massachusetts, 1985.

[5] E. Charniak, C.K. Riesbeck, D.V. McDermott, and J.R. Meehan. *Artificial Intelligence Programming.* Lawrence Erlbaum Associates, Inc., Hillsdale, New Jersey, 1987.

[6] P.T. Devanbu, R.J. Brachman and G.S. Peter. LaSSIE - A Classification-Based Software Information System. In *Proceedings of the 12th ICSE,* Nice, France, March, 1990.

[7] D.W. Embley and S.N. Woodfield. A Knowledge Structure for Reusing Abstract Data Types. In *Proceedings of the Ninth Annual Software Engineering Conference,* Monterey, California, 1987.

[8] W.B. Frakes and B.A. Nejmeh. An Information System for Software Re-Use. In *Proceedings of the Tenth Minnowbrook Workshop on Software Re-Use,* pp. 142-151, 1987.

[9] R.A. Gagliano et. al. Issues in Reusable Ada Library Tools. In *Proceedings of the 6th EFISS Symposium,* Atlanta, Georgia, 1989.

[10] J.A. Hendler, Y.C. Wong, A. Vinciguerra, and J. Mogilensky. AIRS: an AI-based Ada Reuse Tool. *Proceedings of the AIDA Conference,* October, 1987.

[11] Y.S. Maarek and D.M. Berry. The Use of Lexical Affinities in Requirements Extraction. In *Proceedings of the 5th International Workshop on Software Specification and Design,* Pittsburg, Pennsylvania, pp. 196-202, May, 1989.

[12] N.J. Nilsson. *Principles of Artificial Intelligence.* Tioga, Palo Alto, California, 1980.

[13] E.J. Ostertag and J.A. Hendler. An AI-based reuse system. Technical Report CS-TR-2197, UMIACS-TR-89-16, University of Maryland, Dept. of Computer Science, February 1989.

[14] R. Prieto-Díaz. *A Software Classification Scheme.* Ph.D. dissertation, Dept. of Information

and Computer Science, University of California at Irvine, 1985.

[15] R. Prieto-Díaz. Domain Analysis for Reusability. In *Proceedings of COMPSAC '87,* Tokyo, Japan, pp. 23-29, October, 1987.

[16] R. Prieto-Díaz. Classifying of Reusable Modules. In T.J. Biggerstaff and A.J. Perlis, eds., *Software Reusability,* Volume I, ACM Press, 1990.

[17] P.A. Straub and E.J. Ostertag. Semantics of the Extensible Description Formalism. Technical Report CS-TR-2561, UMIACS-TR-90-137, University of Maryland, Dept. of Computer Science, November, 1990.

[18] M. Wood and I. Somerville. An Information System for Software Components. ACM SIGIR Forum, 22:3, Spring/Summer 1988.