The SoRReL Papers

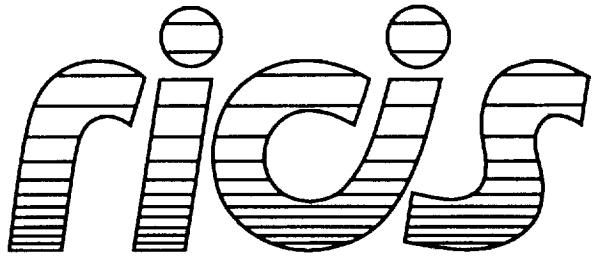
Recent Publications of the Software Reuse Repository Lab

David Eichmann (ed.)
West Virginia University

May 20, 1992

Cooperative Agreement NCC 9-16
Research Activity No. SE.43

NASA Johnson Space Center
Information Systems Directorate
Information Technology Division



Research Institute for Computing and Information Systems
University of Houston-Clear Lake

TECHNICAL REPORT

The RICIS Concept

The University of Houston-Clear Lake established the Research Institute for Computing and Information Systems (RICIS) in 1986 to encourage the NASA Johnson Space Center (JSC) and local industry to actively support research in the computing and information sciences. As part of this endeavor, UHCL proposed a partnership with JSC to jointly define and manage an integrated program of research in advanced data processing technology needed for JSC's main missions, including administrative, engineering and science responsibilities. JSC agreed and entered into a continuing cooperative agreement with UHCL beginning in May 1986, to jointly plan and execute such research through RICIS. Additionally, under Cooperative Agreement NCC 9-16, computing and educational facilities are shared by the two institutions to conduct the research.

The UHCL/RICIS mission is to conduct, coordinate, and disseminate research and professional level education in computing and information systems to serve the needs of the government, industry, community and academia. RICIS combines resources of UHCL and its gateway affiliates to research and develop materials, prototypes and publications on topics of mutual interest to its sponsors and researchers. Within UHCL, the mission is being implemented through interdisciplinary involvement of faculty and students from each of the four schools: Business and Public Administration, Education, Human Sciences and Humanities, and Natural and Applied Sciences. RICIS also collaborates with industry in a companion program. This program is focused on serving the research and advanced development needs of industry.

Moreover, UHCL established relationships with other universities and research organizations, having common research interests, to provide additional sources of expertise to conduct needed research. For example, UHCL has entered into a special partnership with Texas A&M University to help oversee RICIS research and education programs, while other research organizations are involved via the "gateway" concept.

A major role of RICIS then is to find the best match of sponsors, researchers and research objectives to advance knowledge in the computing and information sciences. RICIS, working jointly with its sponsors, advises on research needs, recommends principals for conducting the research, provides technical and administrative support to coordinate the research and integrates technical results into the goals of UHCL, NASA/JSC and industry.

The SoRReL Papers

Recent Publications of the Software Reuse Repository Lab

RICIS Preface

This research was conducted under auspices of the Research Institute for Computing and Information Systems by Dr. David Eichmann of West Virginia University. Dr. E. T. Dickerson served as RICIS research coordinator.

Funding was provided by the Information Technology Division, Information Systems Directorate, NASA/JSC through Cooperative Agreement NCC 9-16 between NASA Johnson Space Center and the University of Houston-Clear Lake. The NASA technical monitor for this activity was Ernest M. Fridge, III of the Information Technology Division, Information Systems Directorate, NASA/JSC.

The views and conclusions contained in this report are those of the author and should not be interpreted as representative of the official policies, either express or implied, of UHCL, RICIS, NASA or the United States Government.

SoRReL

West Virginia University

Software Reuse Repository Lab

Department of Statistics and Computer Science
West Virginia University
Morgantown, WV 26506
(304) 293-3607 email: sorrel@cs.wvu.wvnet.edu

The SoRReL Papers

Recent Publications of the Software Reuse Repository Lab

David Eichmann (ed.)

May 20, 1992

Contents

Design of a Lattice–Based Faceted Classification System

David Eichmann John Atkins

Dept. of Statistics and Computer Science
West Virginia University
Morgantown, WV 26506

Abstract We describe a software reuse architecture supporting component retrieval by facet classes. The facets are organized into a lattice of facet value sets and facet n–tuples. The query mechanism supports both precise retrieval and flexible browsing.

1. Introduction

There are many obstacles in the path to development of a practical and useful software reuse environment. Retrieval of "suitable" reuse candidates from a collection of possibly thousands of components is a particularly significant obstacle. We describe the design of a component classification scheme and its associated query mechanism. The classification scheme is based upon a lattice of facet values and facet tuples. The query mechanism uses type inference rules to locate and retrieve those components whose classifications in the lattice are subtypes of the query specification.

1.1. Software Reuse

Reuse has long been an accepted principle in many scientific disciplines. Engineers make design decisions on the availability of components that facilitate product development, biologists use established laboratory instruments and chemists use standardized measuring devices to record experimental results. It would be unthinkable for an engineer to "design and develop" the transistor every time that a transistor is required in an electrical instrument. Computer scientists, however, are guilty of a comparable practice in their discipline: software reuse is not widely practiced in the computer science field. Generally, the reasons are:

1. Development standards have not been established for software;
2. There is a pervasive belief that if it is "not developed here", it can't be used by "us";
3. Software is all too often developed with respect to a specific requirement with no consideration given to reuse in other environments;
4. Many languages encourage constructs that are not conducive to reuse;
5. Software Engineering principles are not widely practiced and consequently, requirements and design documents often are not available with the code; and
6. No widely accepted methodology has been developed to facilitate the identification and access of reusable components.

Regardless of the reasons for not developing software for eventual reuse, the spiraling cost of new software development is mandating an increased interest in software reuse. It has been estimated that in 1990 alone, the output of source code will be 15.3 billion lines of code [11]. With the minimal effort to reuse existing software, it is natural to ask what percentage of this enormous number of lines of code will represent duplication of effort. It has been estimated that only 30 to 40% of this code will represent novel applications while 60 to 70% of the code will apply to generic computer tasks such as data entry, storage, sorting, searching, etc.

Although there are no definitive answers as yet to the software reuse problem, there is substantial ongoing research on the problem. One area of research is to identify characteristics of software components that enhance the reuse potential of the component in terms of its bindings to other modules [3]. Another area of research is to identify techniques that can be used to translate a software component that has marginal reuse potential to one that can be easily incorporated into a larger system. A third research area relative to software reuse that has been extensively studied is that of identifying metrics that measure software complexity. An example of this is

This work was supported in part by a grant from MountainNet Inc. as part of the AdaNet project under NASA cooperative agreement NCC9–16.

Second International Conference on Software Engineering and Knowledge Engineering, Skokie, IL, June 21-23, 1990, pages 90-97.

McCabe's Complexity metric. A very recent area of research in software reuse is that of the problem of classifying software in order to identify and access the software [4], [12]. The most promising classification method for software reuse is the Faceted Classification System. This methodology has been studied extensively by Prieto–Diaz and forms the basis for the methodology presented in this paper.

1.2. Faceted Classification

The faceted classification methodology, as studied by Prieto–Diaz, begins by using Domain Analysis "to derive faceted classification schemes of domain specific objects" [13]. This process relies on a library notion known as Literary Warrant. Literary Warrant collects a representative sample of titles which are to be classified and extracts descriptive terms to serve as a grouping mechanism for the titles. From this process, the classifier not only derives terms for grouping but also identifies a vocabulary that serves as values within the groups.

From the software perspective, the groupings or facets become a taxonomy for the software. Using Literary Warrant, Prieto–Diaz has identified six facets that can be used as a taxonomy [14]. These facets are: Function, Object, Medium, System Type, Functional Area and Setting. Every software component is classified by assigning a value for each facet for that component. For example, a software component in a Relational Database Management System that parses expressions might be classified with the tuple

(parse, expression, stack, interpreter, DBMS,).

Thus, the Function facet value for this component is "parse", the Object facet value is "expression", etc. Note that no value has been assigned for the Setting facet as this software component does not seem to have an appropriate value for the Setting facet.

The software reuser locates software components in a faceted reuse system by specifying facet values that are descriptive of the software desired. For example, if we are using Prieto–Diaz's facets, suppose that we wish to find a software component to format text. We might query the system by constructing the tuple

(format, text, file, file handler, word processor, *).

Note that the asterisk for the value for the Setting facet acts as a wild card in the query which indicates that there is no constraint on that facet. If the query results in one or more "hits", then the reuser chooses from the hits the particular software component that best fits the desired need. The problem arises if no hits are obtained or if the software that is identified is not appropriate to the needs of the reuser. One solution is to weaken the query by relaxing one or more constraints by replacing a facet value with a wild card. For example, if the Functional Area facet has the least significance to the required need, the reuser could again pose the query with the tuple

(format, text, file, file handler, *, *).

This process of weakening the query continues until a suitable component is retrieved.

An alternative method to continue the search after an initial query is known as the method of "conceptual closeness." In this method, pairs of facet values for the same facet have numeric values associated with them that in a sense measures their "degree of sameness." For example, the two facet values "delete" and "remove" would be very close in meaning and hence would have a metric value close to 0 indicating their semantic closeness. However, the two values "add" and "format" for Function have little in common and hence would have a closeness value nearer to 1. In this method, the system assumes the responsibility for continued searches by modifying the query by replacing facet values with values that are "close" in meaning as determined by the closeness metric. For example, if the facet value "editor" is closer to "word processor" in terms of the metric than any other value in any facet, then the system poses the query with the modified tuple

(format, text, file, file handler, editor, *)

and continues in this manner until a hit is obtained.

Although this appears to be a reasonable solution to the problem of continued searches, the difficulty lies in the need to assign meaningful closeness values to pairs of facet values. With a large collection of values, this is a daunting task. However, one solution is suggested by adapting the work of Kruskal [8] to the conceptual closeness problem. In this method, a metric is assigned to pairs of values based on user acceptance of modified queries. The method requires the use of a two dimensional matrix for each facet indexed by the facet values themselves. For example, if an original query tuple consisting of

(format, text, file, file handler, word processor, *)

failed to achieve a hit and the user later accepted a component with the query tuple

(format, text, file, file handler, editor, *),

the matrix corresponding to the Functional Area facet would have one added to the two matrix cells corresponding to the entries for "word processor" and "edi-

tor". Now if N is half of the total of the cell values in the matrix, then the distance between "word processor" and "editor" is defined to be 1 – (cell value)/N where the cell value is the value in either of the entries corresponding to the pair "word processor" and "editor". It is clear that this method requires a large and patient user group in order to establish viable metric values.

1.3. Lattices

The faceted classification model that we shall describe in the next section is based on the mathematical notion of a lattice. The definition of a lattice requires the concept of a partial ordering on a set. Thus, a partial ordering $<$ on a set A is a relation defined on A that satisfies three conditions, namely:

a. Reflexive: for all x in A, $x < x$;
b. Antisymmetric: for all x, y in A, if $x < y$ and $y < x$, then $x = y$;
c. Transitive: for all x, y and z in A, if $x < y$ and $y < z$, then $x < z$.

For example, the arithmetic comparison "less than or equal" is a partial ordering on the Natural numbers. Another example is the subset relation defined on the power set of a set. It should be noted that a partial ordering on a set does not guarantee that any two objects in the set can be compared using the partial ordering. For example, two arbitrary elements in the power set are not comparable in the sense that one need be a subset of the other.

A lattice is a set A on which is defined two binary operations, $\wedge$ (meet) and $\vee$ (join), which satisfy the following:

a. Idempotent: for any in A, $x \wedge x = x$ and $x \vee x = x$;
b. Commutative: for any x and y in A, $x \wedge y = y \wedge x$ and $x \vee y = y \vee x$;
c. Associative: for any x, y and z in A, $x \wedge (y \wedge z) = (x \wedge y) \wedge z$ and $x \vee (y \vee z) = (x \vee y) \vee z$;
d. Absorption Law: for any x and y in A, if $x < y$, then $x \vee y = y$ and $x \wedge y = x$.

Additionally, if for any x, y and z in A, $x \wedge (y \vee z) = (x \wedge y) \vee (x \wedge z)$ and $x \vee (y \wedge z) = (x \vee y) \wedge (x \vee z)$, we say that the lattice is distributive. For example, the power set with intersection as the meet and union as the join forms a distributive lattice using the subset partial order.

Let $<$ be a partial ordering on a set A. If X is a subset of A, we say that an element a in A is a lower bound of X if $a < x$ for every x in X. A Greatest Lower Bound (GLB) of X is a lower bound b of X with the property that if a is any other lower bound of X, then $a < b$. It is clear that if a GLB exists for a subset X of A, then it must be unique. For example, any subset of elements in the power set has a GLB consisting of the intersection of all elements in the subset. In a lattice, any two elements have a GLB which is just the meet of the two elements, i.e. if x and y are in a lattice A, then $x \wedge y < x$ and $x \wedge y < y$ and if z is any lower bound of both x and y, then $z < x \wedge y$.

There is a dual to lower bounds which is the notion of upper bounds. An element a is an upper bound for a set X if $x < a$ for all x in X. A Least Upper Bound (LUB) of a set X is an upper bound b such that if a is any other upper bound, then $b < a$. For the example of the power set, a LUB for a set X is the union of all the elements in the subset. In a lattice, any two elements also have a least upper bound which is just the join of the two elements. Thus, for any two elements x and y in A, $x < x \vee y$ and $y < x \vee y$ and if z is any upper bound of both x and y, then $x \vee y < z$.

We note that if A is a set with a partial ordering $<$ such that any two elements have a GLB and a LUB, then the set is a lattice where the meet of any two elements is the GLB of the elements and the join of any two elements is just the LUB of the elements.

1.4. Subtypes and Inheritance

The popularity of the Smalltalk programming language [9], with its object orientation and built–in type inheritance, has resulted in a flurry of research in object–oriented database systems. An object–oriented database system is one that is organized around objects and which communicates through message–passing. Operations (termed methods) are associated with each object in a database; some of these operations are bound to specific types of messages for that object. Most message–passing systems are not strongly typed, but rather perform run–time type checking. This is done primarily to support rapid prototyping of applications. Deferring the binding of an object or message to a type until run–time reduces the amount of effort needed to begin exercising an application, but it also requires a run–time system that can handle the errors that may arise.

The object classes in an object–oriented database are organized into a partial ordering. Object classes *inherit* attributes and methods from their ancestors in the ordering. Single inheritance schemes restrict a given object class to at most one immediate ancestor in the partial ordering. Multiple inheritance schemes allow a given object class to have any number of immediate ancestors in the partial ordering. Cardelli [5] formalizes some of the semantics of multiple inheritance.

Object–oriented database systems have a number of design goals, some concerning typing, but others concerning peripheral issues (such as rapid prototyping). The type semantics of object–oriented systems (including inheritance and subtyping) is present in other systems which are not based upon message–passing (e.g., Morpheus [7], Galileo [2]). Such systems are strongly typed, and hence, as Cardelli and Wegner [6] argue, can produce more efficient and reliable applications.

Horn [10] introduces the notion of *conformance*, allowing one type instance to be treated as if it were an instance of another type. In a limited sense, this is what happens with inheritance, but conformance is more general. Inheritance requires that this treatment only be allowed when moving up the type hierarchy or lattice. Inheritance uses a partial ordering of types (by subtype), plus an implicit definition of existence dependencies between a given type and its ancestors. Conformance can hold for arbitrary types, independent of any type ordering scheme. Such a notion is clearly superior to hierarchies or lattices for type–related query languages, where intermediate results (derived from existing types, but not part of the database schema) need to be manipulated.

Inheritance–based systems are, in some sense, *navigational*. A user querying an object–oriented database must be aware of the inheritance structure of that specific database, just as a user querying a network database must be aware of database structure. Because of their non–navigational characteristics conformance–based models promise to gain prominence over inheritance–based models, just as relational models have over network models.

2. The Reuse Type Lattice

Figure 1 shows the general structure of the reuse type lattice. At the top is $\top$, the special universal type. Any value conforms to the universal type. At the bottom is $\bot$, the void type. These two special types ensure that any two types in the lattice have a least upper bound and a greatest lower bound, respectively. Between the universal and void types appear the upper and lower bounds for the two type constructors facet and tuple. $Facet_0$ characterizes the notion of the empty facet type; it contains no values, but is still a facet. Likewise, Facet characterizes the notion of the set of all possible facet values. The dotted line between them indicates that an arbitrary number of types may appear here in the lattice. For example, figure 2 shows the sublattice for facet sets for the examples in section 1.2.

The tuple sublattice has a similar structure. At the top is the empty tuple type {}, characterizing a tuple w

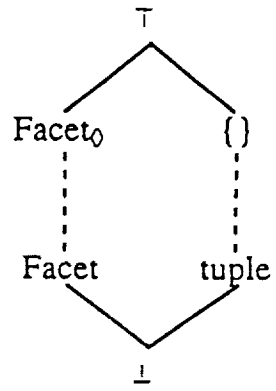


Figure 1. The reuse type lattice

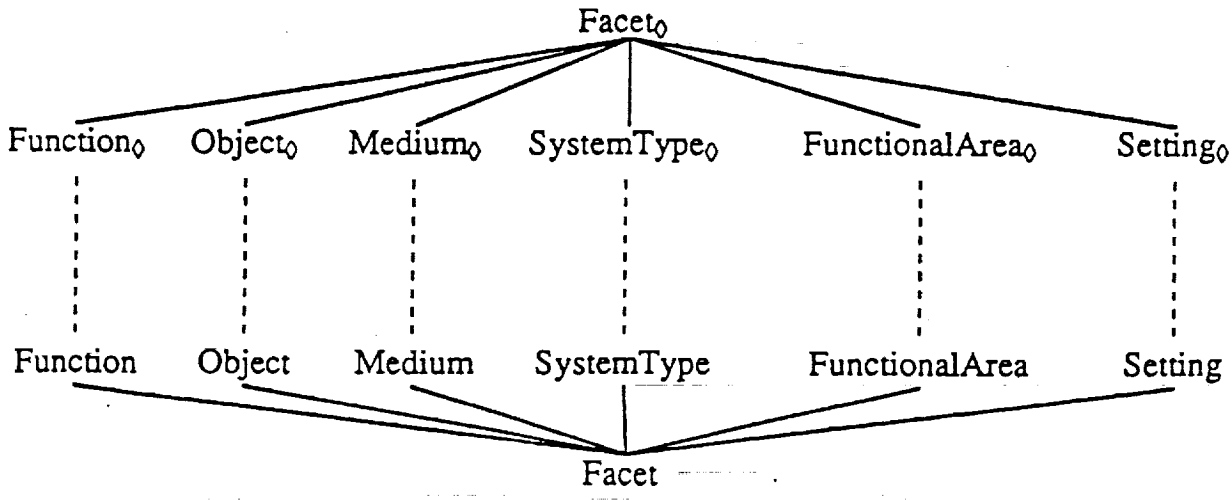


Figure 2. The Sublattice of Facet Sets

no facets. At the bottom is tuple, the tuple type with all possible facets.

2.1. Facets vs. Facet Value Sets

Traditional retrieval of individual facet values relies upon maximal conjunction of boolean terms for retrieval of matches on *all* facets and maximal disjunction of boolean terms for matches on *any* facet of an expression. In order to fit the notion of facet into the type lattice, we look at sets of facets. A set of facets corresponds to a conjunction on all of the facets comprising the set. Each set occupies a unique position in the type lattice. We handle disjunction by allowing a given component to occupy multiple lattice positions. Matching occurs on any of the positions, providing the same semantics as disjunction.

Facet values are equivalent to enumeration values. We attach no particular connotation within the type system to a particular facet value. Values are bound to some semantic concept in the problem domain.

The subset relation is our partial order. The least value of this portion of the lattice is the set of all facet values from all facets in the problem domain, denoted by the distinguished name Facet. The greatest value of this portion of the lattice is the empty set, denoted by the distinguished name Facet$_\emptyset$. The union operator generates the greatest lower bound. The intersection operator generates the least upper bound.

3. Type Inference Rules

We begin with a brief remark concerning notation. In the inference rules that follow, the symbol A represents an existing set of assumptions. A always contains the type information generated by the database schema which implements the repository. It is occasionally necessary to extend the set of assumptions with some additional information. A.x denotes the set of assumptions extended with the fact x. $A \vdash x$ states that given a set of assumptions A, x can be inferred. Inferences above the horizontal line act as premises for the conclusions, the inferences below the horizontal line. An expression is *well–typed* if a type for the expression can be deduced using the available inference rules, otherwise it is *ill–typed.*

3.1. Domain Interval Subtyping

We adapt the notion of a domain interval [7] to formalize our notion of facet value sets. In [7] a subtype was smaller than its supertype; here the reverse is true, a subtype is a larger collection of values than its supertype.

A *domain interval* is a type qualification that explicitly denotes the valid subrange(s) for a base type. Assume that t is a base type ordered by $\leq$ (the ordering may be arbitrary). A domain that is (inclusively) delimited by two values, a and b, is denoted $t_{\langle a \ldots b \rangle}$. A non–inclusive lower bound is denoted a^+ and a non–inclusive upper bound is denoted by b^-. Intervals made up of more than a single continuous value range are denoted by a set of ranges, for example, $t_{\langle a \ldots b,\, e \ldots d,\, e \rangle}$ denotes the interval that includes the subinterval a through b inclusive, the subinterval c through d inclusive, and the singleton value e. The singleton range e is equivalent to e...e. When we use such notation we intend that $a \leq b$ and $c \leq d$, but not necessarily that $b \leq c$ or $d \leq e$. An empty pair of brackets, $t_\emptyset$, denotes an empty interval, i.e., one which contains no elements. In our particular application, the base types are finite sets of enumeration (facet) values.

Premises concerning membership of interval boundary values (e.g., m and n in (1.1) and (1.2)) are assumed to be part of the assumptions, and will not be explicitly mentioned after this. Rule (1.1) provides for subtyping a

$$\frac{A \vdash m \in t \quad A \vdash n \in t \quad A \vdash m \leq n}{A \vdash t \trianglelefteq t_{\langle m \ldots n \rangle}} \qquad (1.1)$$

subrange of some type t; (1.2) does the same for two subranges of some type t. Rule (1.3) extends subtyping to

$$\frac{A \vdash m \in t \quad A \vdash m' \in t \quad A \vdash n \in t \quad A \vdash n' \in t \quad A \vdash m' \leq m \leq n \leq n'}{A \vdash t_{\langle m' \ldots n' \rangle} \trianglelefteq t_{\langle m \ldots n \rangle}} \qquad (1.2)$$

domain intervals, where each subinterval in the subtype is a subtype of some interval in the supertype.

$$\frac{A \vdash t_{\langle m_1 \ldots n_1 \rangle} \trianglelefteq t_{\langle m_1' \ldots n_1' \rangle} \quad \vdots \quad A \vdash t_{\langle m_i \ldots n_i \rangle} \trianglelefteq t_{\langle m_i' \ldots n_i' \rangle}}{A \vdash t_{\langle m_1 \ldots n_1, \ldots, m_i \ldots n_i \rangle} \trianglelefteq t_{\langle m_1' \ldots n_1', \ldots, m_i' \ldots n_i' \rangle}} \qquad (1.3)$$

The following rules are used to combine ranges in domain intervals. In rule (1.4), two ranges in an interval

$$\frac{A \vdash x : t_{\langle \ldots, a \ldots b,\, b \ldots c, \ldots \rangle}}{A \vdash x : t_{\langle \ldots, a \ldots c, \ldots \rangle}} \qquad (1.4)$$

that share a common endpoint can be combined into a single range. This 4can also be done when one end point is inclusive and the other is exclusive (rules (1.5) and

(1.6)). Overlapping ranges are merged into a single

$$\frac{A \vdash x : t_{\langle \ldots, a \ldots b^{-}, b \ldots c, \ldots \rangle}}{A \vdash x : t_{\langle \ldots, a \ldots c, \ldots \rangle}} \quad (1.5)$$

$$\frac{A \vdash x : t_{\langle \ldots, a \ldots b, b^{+} \ldots c, \ldots \rangle}}{A \vdash x : t_{\langle \ldots, a \ldots c, \ldots \rangle}} \quad (1.6)$$

range that uses the minimum of the two lower bounds as the new lower bound and the maximum of the two upper bounds as the new upper bound in rules (1.7) and (1.8).

$$\frac{A \vdash x : t_{\langle \ldots, a \ldots c, b \ldots d, \ldots \rangle} \quad A \vdash a \le b \le c \le d}{A \vdash x : t_{\langle \ldots, a \ldots d, \ldots \rangle}} \quad (1.7)$$

$$\frac{A \vdash x : t_{\langle \ldots, a \ldots d, b \ldots c, \ldots \rangle} \quad A \vdash t_{\langle a \ldots d \rangle} \le t_{\langle b \ldots c \rangle}}{A \vdash x : t_{\langle \ldots, a \ldots d, \ldots \rangle}} \quad (1.8)$$

The next two inference rules deal with unary domain values. And the last two deal with complete intervals.

$$\frac{A \vdash x : t_{\langle \ldots, a, \ldots \rangle}}{A \vdash x : t_{\langle \ldots, a \ldots a, \ldots \rangle}} \quad (1.9)$$

$$\frac{A \vdash x : t_{\langle \ldots, a \ldots a, \ldots \rangle}}{A \vdash x : t_{\langle \ldots, a, \ldots \rangle}} \quad (1.10)$$

$$\frac{A \vdash x : t}{A \vdash x : t_{\langle -\infty \ldots \infty \rangle}} \quad (1.11)$$

$$\frac{A \vdash x : t_{\langle \ldots, -\infty \ldots \infty, \ldots \rangle}}{A \vdash x : t} \quad (1.12)$$

In order to establish the type of the result of an operation such as union, some notion of domain interval union is needed. If M and N are two intervals over the same type, then $M \cup N$ is constructed by merging the two sets of ranges making up the intervals, and using the domain inference rules described above to reduce the result.

$$\frac{A \vdash x : t_{\langle M \cup N \rangle}}{A \vdash x : t_{\langle M, N \rangle}} \quad (1.13)$$

In a similar fashion, for two intervals M and N over the same type, their intersection, $M \cap N$, can be constructed by selecting only those ranges which are common to both domain intervals. The domain inference rules are used to decompose the given ranges into sets of disjoint ranges and common ranges. The set of common ranges makes up the intersection interval.

$$\frac{A \vdash m_b < n_a}{A \vdash t_{\langle (m_a \ldots m_b) \cap (n_a \ldots n_b), M \rangle} = t_{\langle M \rangle}} \quad (1.14)$$

$$\frac{A \vdash m_a < n_a \le m_b < n_b}{A \vdash t_{\langle (m_a \ldots m_b) \cap (n_a \ldots n_b), M \rangle} = t_{\langle (n_a \ldots m_b), M \rangle}} \quad (1.15)$$

$$\frac{A \vdash m_a \le n_a \le n_b \le m_b}{A \vdash t_{\langle (m_a \ldots m_b) \cap (n_a \ldots n_b), M \rangle} = t_{\langle (n_a \ldots n_b), M \rangle}} \quad (1.16)$$

3.2. Tuple Subtyping

This collection of inference rules explicitly types the tuples that classify components. We view a tuple r to be of type record, $\{t_1, \ldots, t_n\}$. The type t_i must be a facet type. The empty tuple (i.e., the tuple containing no facets) is of type $\{\}$, the tuple type with no components. The order in which types appear is not arbitrary, since position is used to distinguish facets.

Inference rules (2.1) and (2.2) allow for the definition of a tuple and the extraction of an attribute from a tuple. If e_1 through e_n are type expressions of type t_1

$$\frac{\begin{array}{c} A \vdash e_1 = t_1 \\ \vdots \\ A \vdash e_n = t_n \end{array}}{A.(r = \{e_1, \ldots, e_n\}) \vdash r : \{t_1, \ldots, t_n\}} \quad (2.1)$$

through t_n respectively, then the tuple constructed from them will be of the type resulting from the record constructor '{}' applied to those types. We use type expressions to allow construction of attribute types without requiring the earlier definition of all the types needed. Note that the same syntax is used to denote both the *definition* of the tuple and its *type*. If attribute i in tuple r is of type t then the result type for the component extraction r.i is t.

$$\frac{A \vdash r : \{t_1 \ldots t_n\} \quad A \vdash 1 \le i \le n}{A \vdash r.i : t} \quad (2.2)$$

New tuple types are constructed from existing tuple types using the tuple constructor '&' which accepts two tuple types and returns a tuple type containing all components of both argument types.

$$\frac{A \vdash T_1 : \{t_1, \ldots, t_m\} \quad A \vdash T_2 : \{t_{m+1}, \ldots, t_n\} \quad A \vdash 1 \le m < n}{A \vdash T_1 \,\&\, T_2 = \{t_1, \ldots, t_n\}} \quad (2.3)$$

Rules (2.1) and (2.2) give the type semantics for construction of tuples from attributes and for extraction of an attribute from a tuple. Rule (2.4) characterizes the notion of subtype between two tuples: One tuple is a subtype of another if it has all of the attributes of the other (attributes common to both tuple types must be of the same type in both tuple types), and possibly some additional attributes. This may seem contrary to the in-

$$\frac{\begin{array}{c} A \vdash t_1 \\ \vdots \\ A \vdash t_m \\ \vdots \\ A \vdash t_n \\ A \vdash 1 \le m \le n \end{array}}{A \vdash \{t_1, \ldots, t_m, \ldots, t_n\} \preceq \{t_1, \ldots, t_m\}} \quad (2.4)$$

tuitive notion of subtype being a restriction of a type. Consider, however, that an instance of a subtype must be able to be used as an instance of its supertype, and thus must contain all of the supertype's attributes.

Rule (2.5) extends record subtyping to handle the

$$\frac{\begin{array}{c} A \vdash 1 \le m \le n \\ A \vdash t'_1 \preceq t_1 \\ \vdots \\ A \vdash t'_m \preceq t_m \end{array}}{A \vdash \{t'_1, \ldots, t'_m, \ldots, t_n\} \preceq \{t_1, \ldots, t_m\}} \quad (2.5)$$

situation where a component of the subtype is a subtype of the corresponding component in the supertype. Inference rule (2.4) required that the corresponding attributes be of the same type. Rule (2.5) generalizes (2.4) by dealing with subtyping of the attributes in addition to the respective record types.

4. Querying the Repository

The repository is partitioned by structural similarity (package, function, etc.). Each partition is associated with a set of facets which characterize and classify the members of the partition. The particular facets and the number of facets associated with a partition varies as needed to adequately characterize it. A given facet may be unique to a partition, or it may be shared by many partitions. The function facet from section 1.2. is a good example of a facet likely to be shared by a majority of partitions in the repository.

Each partition instance has one or more lattice vertices that correspond to the sets of section 2.1. There is always the primary lattice vertex corresponding to the tuple of facet value sets characterizing this component as a member of the partition. Additionally, there may be zero or more secondary lattice vertices corresponding to alternative characterizations of the component or characterizations of subcomponents contained within this component.

4.1. Repository Structure

Two persistent storage areas comprise the actual repository: a set of text files, and a set of database relations. The text files contain the body of the components themselves, or descriptions of them (in the case of a commercial product described in a local repository). The database relations store the lattice vertices.

Each database relation corresponds to the lattice vertex characterizing a particular repository partition. The type of the relation is then the type of the partition, which is the least upper bound of all the tuple types of the component vertices comprising the partition. Efficient algorithms for lattice operations such as LUB are described in [1].

There is also a relation made up of facet value/synonym pairs. This relation is described in section 4.2. Additional relations may also be present if there are alternative characterizations or subcomponents characterizations not equivalent to some primary partition characterization.

4.2. Query Evaluation

A query is a boolean expression containing predicates and the operators and, or, and not. A predicate is simply a constant of type tuple. When a user issues a query, the query evaluator first treats all of the facet values in the query as synonyms and replaces them with actual facet values from the value/synonym relation. For example, "database," "databases," "data base," and "data bases" might all be replaced with "database." The evaluator then locates all of the relations in the database whose type conforms to some predicate of the query using the inference rules of section 3. Specific tuples which conform to some predicate are then retrieved from the conforming relations (once more using the inference rules). The result is then a set of component references, which can be optionally retrieved from the text storage area.

4.3. Browsing as Retrieval of Subtypes

Treating a query as an editable entity in the user interface provides a straightforward browsing tool. For example, attaching facets to a query comprised of a single tuple makes the query less general. Fewer and fewer partitions conform to the tuple type. Specifying exactly those facets found in a given partition restricts retrieval to only that partition. Over–qualification results in empty retrieval.

Removing facets from the query tuple makes the query in turn more general. Specifying an empty tuple results in all partitions of the repository conforming to the type of the query tuple (all record types are subtypes of the empty record {}).

5. Conclusions

The reuse architecture described here uses the proven method of faceted classification as a starting point for a retrieval mechanism providing both precise characterization of components and flexible specification of queries. Its simple user interface encapsulates a data model founded in formal lattice and type theory.

6. References

[1] H. Ait-Kaci, R. Boyer, P. Lincoln, R. Nasr, "Efficient Implementation of Lattice Operations," *ACM Transactions on Programming Languages and Systems*, vol. 11, no. 1, p. 115, 1989.

[2] A. Albano, L. Cardelli, and R. Orsini, "Galileo: A Strongly-Typed, Interactive Conceptual Language," *ACM Transactions on Database Systems*, vol. 10, no. 2, p. 230, 1985.

[3] V. R. Basili, H. D. Rombach, J. Bailey, A. Delis, F. Farhat, "Ada Reuse Metrics," *Workshop Proceedings: Ada Reuse and Metrics*, Atlanta, Ga., June 15–16, 1988.

[4] G. Booch, *Software Components with Ada*, benjamin/cummings, Menlo Park, California, 1987.

[5] L. Cardelli, "A Semantics of Multiple Inheritance," in *Semantics of Data Types (Proceedings International Symposium Sophia-Antipolos, France, June 1984)*, Springer-Verlag, Lecture Notes in Computer Science, vol. 173, p. 51.

[6] L. Cardelli, P. Wegner, "On Understanding Types, Data Abstraction, and Polymorphism," *ACM Computing Surveys*, vol. 17, no. 4, p. 471, 1985.

[7] D. Eichmann, *Polymorphic Extensions to the Relational Model*, Ph.D. dissertation, The University of Iowa, Iowa City, Ia., August 1989. Also available as technical report 89-05.

[8] R. Gagliano, G. S. Owen, M. D. Fraser, K. N. King, P. A. Honkanen, "Tools for Managing a Library of Reusable Ada Components," *Workshop Proceedings: Ada Reuse and Metrics*, Atlanta, Ga., June 15–16, 1988.

[9] A. Goldberg, D. Robson, *Smalltalk-80: The Language and Its Implementation*, Addison-Wesley, 1983.

[10] C. Horn, "Conformance, Genericity, Inheritance and Enhancement," *ECOOP'87 – Proc. European Conference on Object-Oriented Programming*, p. 223, Paris, France, June 15–17, 1987.

[11] T. C. Jones, "Technical and Demographic Trends in the Computing Industry," *Proceedings of the 1983 DSSD Conference*, Topeka, Kansas, October, 1983.

[12] R. Prieto-Diaz, "Domain Analysis for Reusability," *Proceedings of COMPSAC 87*, Tokyo, Japan, October, 1987.

[13] R. Prieto-Diaz, "Faceted Classification and Reuse Across Domains," Unpublished Draft.

[14] R. Prieto-Diaz, P. Freeman, "Classifying Software for Reusability," *IEEE Software*, vol. 4, no. 1, p. 6, 1987.

tant to the success and utility of a user interface incorporating conceptual closeness.

2.1.3. Lattice-Based Faceted Classification

Eichmann and Atkins [6] described an approach to faceted classification that focused upon a structural framework (type lattices) as an alternative to explicit closeness weights. Each component possessed one or more tuples characterizing it, each comprised of a non-empty set of facet values. Users posed queries as tuples, and reuse candidates were retrieved based upon their conformance to the query tuple.

2.2. Type Signatures

An algebraic specification contains both a syntactic characterization of a component (the signature) and a semantic characterization of a component (the axioms). Algebraic specifications therefore are aptly suited as formal descriptions of software components.

Traditional efforts in reuse concentrated on the structural interfaces between components [1, 2], and hence solely on the signature portion of the specification. This proved less than adequate for component discrimination, in the face of numerous candidate components, all with the same interface, and directly prompted the work in faceted classification described above.

2.3. Type Inference

Recent research in programming language has resulted in a number of languages that are strongly typed, and yet, are flexible and remarkable expressive, (e.g., ML [13]). Such languages rely heavily on inferential mechanisms to ensure safe computation [5, 12]. The concept of conformance is particularly relevant to software repository query mechanisms [11]. Conformance allows one type instance to be treated as if it were an instance of another type, and can hold for arbitrary types, regardless of the type ordering scheme (e.g., inheritance).

Type inference notation organizes around a set of inference rules, comprised of sets of premises and conclusions, separated by a horizontal line. The symbol A represents an existing set of assumptions. A always contains the type information generated by the database schema implementing the repository. A.x denotes the set of assumptions extended with some fact x. $A \vdash x$ states that given a set of assumptions A, and the currently defined set of inference rules, x can be inferred. An expression is well-typed if a type for the expression can be deduced using the available inference rules, otherwise it is ill-typed.

3. A Hybrid Approach

The approach advocated here combines the semantic flexibility of faceted classification with the structural formality of type signatures. We accomplish this through the incorporation of function and abstract data type (ADT) definitions into the type lattice of [6].

3.1. The Type Lattice

As shown in figure 1, there are four principle sublattices comprising the complete type lattice, corresponding to the types generated by facet sets, tuples, functions and ADTs. In addition, the universal type, $\top$, and the void type, $\bot$, ensure that a least upper bound and a greatest lower bound, respectively, exist for any two types in the lattice. The usual built-in types (e.g., integers, strings, etc.) are not shown, in order to simplify the presentation. In principle, they can be specified as ADTs if needed.

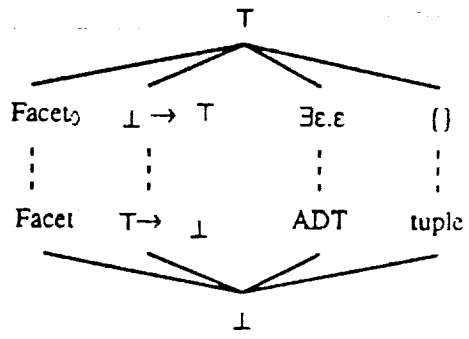


Figure 1.

$Facet_0$ characterizes the empty generic facet type; it contains no values, but is still a facet. Likewise, Facet characterizes the set of all possible facet values. The dotted line indicates an arbitrary number of intermediate types.

The tuple sublattice has a similar structure. At the top is the empty tuple type, {}, characterizing a type with no components. At the bottom is Tuple, the tuple type with all possible components.

Function types are bounded above by $\bot \rightarrow \top$, the function type with a void domain and universal range, and are bounded below by $\top \rightarrow \bot$, the function type with a universal domain and void range.

ADT types are bounded above by $\exists\varepsilon.\varepsilon$, the abstract type denoting a a hidden type, ε, with no information or operations available, and are bounded below by ADT, the type denoting all possible types with all possible operations.

3.2. Inference Rules

3.3. Facets

As in [6], we characterize facets as the inverse of our usual notion of interval subtypes; a facet subtype denotes a larger collection of facet values than does its supertype. Inference rule (1) formalizes this for a complete facet.

$$\frac{A \vdash m \in t \quad A \vdash n \in t \quad A \vdash m \le n}{A \vdash t \lesssim t_{\langle m \ldots n \rangle}} \tag{1}$$

Inference rule (2) does likewise for two singleton intervals, and inference rule (3) for two arbitrary collections of intervals.

$$\frac{A \vdash m \in t \quad A \vdash m' \in t \quad A \vdash n \in t \quad A \vdash n' \in t \quad A \vdash m' \le m \le n \le n'}{A \vdash t_{\langle m' \ldots n' \rangle} \lesssim t_{\langle m \ldots n \rangle}} \tag{2}$$

$$\frac{A \vdash t_{\langle m_1 \ldots n_1 \rangle} \lesssim t_{\langle m_1' \ldots n_1' \rangle} \quad \vdots \quad A \vdash t_{\langle m_i \ldots n_i \rangle} \lesssim t_{\langle m_i' \ldots n_i' \rangle}}{A \vdash t_{\langle m_1 \ldots n_1, \ldots, m_i \ldots n_i \rangle} \lesssim t_{\langle m_1' \ldots n_1', \ldots, m_i' \ldots n_i' \rangle}} \tag{3}$$

A number of inference rules no presented here address the reduction and manipulation of intervals [6].

3.3.1. Tuples

We view a tuple r to be of type record, $\{a_1 : t_1, \ldots, a_n : t_n\}$, where attribute a_i is of type t_i. We assume that t_i is some facet, function, or ADT type. Since attributes are labeled, components may appear in any order, and two types are assumed to be equivalent if they only differ in the order of their respective attributes.

Inference rule (4) characterizes subtyping for tuples. Informally, one tuple type is a subtype of another if it has all of the attributes of the other (and possible more), and for those common attributes, the type of a given attribute in the tuple subtype must be a subtype of that attribute's type in the tuple supertype.

$$\frac{A \vdash 1 \le m \le n \quad A \vdash t'_1 \lesssim t_1 \quad \vdots \quad A \vdash t'_m \lesssim t_m}{A \vdash \{i_1 : t'_1, \ldots, i_m : t'_m, \ldots, i_n : t_n\} \lesssim \{i : t_1, \ldots, i_m : t_m\}} \tag{4}$$

Inference rules (5) and (6) support definition of tuple constants and extraction of an attribute value, respectively.

$$\frac{A \vdash e_1 = t_1 \quad \vdots \quad A \vdash e_n = t_n}{A.(r = \{i_1 = e_1, \ldots, i_n = e_n\}) \vdash r : \{i_1 : t_1, \ldots, i_n : t_n\}} \tag{5}$$

$$\frac{A \vdash r : \{i_1 : t_1 \ldots i_n : t_n\} \quad A \vdash 1 \le j \le n}{A \vdash r.i_j : t_j} \tag{6}$$

3.3.2. Functions

Function types are useful both for characterizing programs and for characterizing the operations of ADTs. Inference rule (7) characterizes the usual notion of lambda abstraction, where x is the parameter, t' the parameter's type, e is the body of the function, and t the type of the function's result.

$$\frac{A, x : t' \vdash e : t}{A \vdash \lambda(x : t')\ e\ : (t' \to t)} \tag{7}$$

One function type, $s \to t$, is a subtype of another, $s' \to t'$, if the subtype function accepts the entire domain of the function supertype (i.e., $s' \lesssim s$), and produces a range contained in the supertype range (i.e., $t \lesssim t'$), as shown in inference rule (8).

$$\frac{A \vdash s' \lesssim s \quad A \vdash t \lesssim t'}{A \vdash s \to t \lesssim s' \to t'} \tag{8}$$

Function subtyping seems a little strange at first, but a simple example helps. Assume that f is a function type (1..4) → true and g is a function type (2..3) → (true..false). Function type f is a subtype of g. Any instance of f can always replace an instance of g in an expression without effecting the type–safety of the expression. The instance of f handles at least the values the supertype function does, and produces no more values than does the supertype function.

Inference rule (9) characterizes the type of the result of a function application; if the expression supplied as an ar-

gument is of the proper type, then the result of the function applied to that expression will be well-typed.

$$\frac{A \vdash e : (t' \rightarrow t) \quad A \vdash e' : t'}{A \vdash e\,(e') : t} \qquad (9)$$

3.3.3. ADTs

Inference rules (10) and (11) define type inference for existential types [4]. An existential type consists of a type variable a, representing the type, packaged with some number ($j_1 \ldots j_n$) of instances of the type and/or operations over the type.

$$\frac{\begin{array}{c} A \vdash e_1 : s_1|_{t/a} \\ \vdots \\ A \vdash e_n : s_n|_{t/a} \end{array}}{\begin{array}{l} A \vdash \text{pack } (a = t \text{ in } (j_1 : s_1, \ldots, j_n : s_n)) \\ \qquad (e_1, \ldots, e_n) : \exists a.(j_1 : s_1, \ldots, j_n : s_n) \end{array}} \qquad (10)$$

$$\frac{\begin{array}{l} A \vdash e : \exists b.(j_1 : s_1, \ldots, j_n : s_n) \\ A.(x : (j_1 : s_1, \ldots, j_n : s_n))|_{a/b} \vdash e' : t \end{array}}{A \vdash \text{open } e \text{ as } x\,[a] \text{ in } e' : t} \qquad (11)$$

A given expression e_i is of type s_i when t is substituted for a in s_i, and serves as the implementation of the value or operation labeled j_i in the abstract type. This substitution results in a concrete type (i.e., one with no type variables in it) for the expression. The substitution type t serves as the representation of the abstract type, denoted externally by the existential variable a. The actual representation and the implementations of the operations are not visible externally.

The pack operation constructs an instance of an abstract type, and encapsulates its representation. The open operation performs the converse, binding an abstract type variable to a concrete type, and evaluating some expression in the context of the (now concrete) abstract type.

Subtyping of ADTs derives from subtyping of the type parameters for the abstract type. Inference rule (12) characterizes subtyping of two instances of abstract types.

$$\frac{A.(t_1 \leq t_2) \vdash (t \leq t')}{A \vdash (\exists(t_1 \leq t_2).t) \leq (\exists(t_1 \leq t_2).t')} \qquad (12)$$

Note that in addition to providing subtyping of two ADTs, rule (12) also supports subtyping of two instances of the same ADT.

For an example of the former, $\exists T' \exists(T \leq T').T''$ denotes an existential type T'' generated by a type parameter T, which must be a subtype of the existential type T'. Since instances of abstract types are cross products of instances and operations, T would be a subtype of T' through additional operations. An example of this appeared in [17], showing stacks and dequeues as subtypes of queues.

For an example of the latter, stack of $\text{integer}_{(1..10)}$ is a subtype of stack of integer.

4. The User Interface

A query is a boolean expression containing predicates and the operators *and*, *or*, and *not*. A predicate is simply a constant of type tuple. When a user issues a query, the query evaluator first treats all of the facet values in the query as synonyms and replaces them with actual facet values from a value/synonym relation. For example, *database*, *databases*, *data base*, and *data bases* might all be replaced with database.

The evaluator then locates all of the relations in the database whose type conforms to some predicate of the query by testing the type of each relation in turn, using the inference rules previously described. The query lattice space for a given predicate is bounded above by the predicate type itself, and bounded below by the partition tuples that conform to it. For each user-specified predicate, the evaluator forms the disjunction of conforming relation tuples (with variables in each position) and then substitutes the conjunction of the disjunction and the new predicate in place of the original, user-specified predicate. The result of evaluating this query is then a set of component references for display and optionally, retrieval from the text storage area.

Note that since tuples of more than a single type may be displayed to the user, the query language is polymorphic in one of the manners discussed in [7].

5. Discussion

The work described here is another in a series of experimental user interfaces for software reuse repositories. Our initial efforts concentrated specifically on providing substructure for faceted classification [9]. This approach relied only upon the expertise of the classifier in populating the repository, and as such, suffered from what we refer to as the *vocabulary problem*.

The interface described here ameliorates the situation by supporting as part of the query tuple the specification of a formal interface structure to which the components of interest must conform.

A parallel effort exploring the role that algebraic specification can play in repository retrieval appears in [8]. This work is concerned particularly with retrieval over type signatures and behavioral axioms.

6. References

[1] J. Atkins, private communication, 1989.

[2] T. J. Biggerstaff and A. J. Perlis, *Software Reusability, vol. 1 – Concepts and Models*, Addison–Wesley, New York, NY, 1989.

[3] T. J. Biggerstaff and A. J. Perlis, *Software Reusability, vol. 2 – Applications and Experience*, Addison–Wesley, New York, NY, 1989.

[4] T. J. Biggerstaff and C. Richter, "Reusability Framework, Assessment, and Directions," *IEEE Software*, vol. 4, no. 2, pages 41–49, March, 1987.

[5] L. Cardelli, "Basic Polymorphic Typechecking," *Science of Computer Programming*, vol. 8, pages 147–172, 1987.

[6] L. Cardelli and P. Wegner, "On Understanding Types, Data Abstraction, and Polymorphism," *ACM Computing Surveys*, vol. 17, no. 4, pages 471–522, December 1985.

[7] D. Eichmann, *Polymorphic Extensions to the Relational Model*, Ph.D. dissertation, Dept. of Computer Science, The University of Iowa, Iowa City, IA, August 1989.

[8] D. Eichmann, "Selecting Reusable Components Using Algebraic Specifications," *Second International Conference on Algebraic Methodology and Software Technology (AMAST)*, Iowa City, IA, May 22–25, 1991.

[9] D. Eichmann and J. Atkins, "Design of a Lattice–Based Faceted Classification System," *Second International Conference on Software Engineering and Knowledge Engineering*, Skokie, IL, pages 90–97, June 21–23, 1990.

[10] E. Guerrieri, "On Classification Schemes and Reusability Measurements for Reusable Software Components," SofTech Technical Report IP–256, SofTech, Inc, Waltham, MA 1987.

[11] C. Horn, "Conformance, Genericity, Inheritance and Enhancement," *ECOOP–87 – Proc. European Conference on Object–Oriented Programming*, Paris, France, pages 223–233, June 15–17, 1987.

[12] R. Milner, "A Theory of Type Polymorphism in Programming," *Journal of Computer and System Sciences*, vol. 17, pages 348–375, 1978.

[13] R. Milner, M. Tofte, and R. Harper, *The Definition of Standard ML*, MIT Press, Cambridge, MA, 1990.

[14] R. Prieto–Diaz, *A Software Classification Scheme*, Ph.D. dissertation, Dept. of Information and Computer Science, University of California, Irvine, CA, 1985.

[15] R. Prieto–Diaz and P. Freeman, "Classifying Software for Reusability," *IEEE Software*, vol. 4, no. 1, pages 6–16, January, 1987.

[16] J. V. Guttag and J. J. Horning, "The Algebraic Specification of Abstract Data Types," *Acta Informatica*, vol. 10, pages 27–52, 1978.

[17] A. Snyder, "Inheritance in the Development of Encapsulated Software Components," *Research Directions in Object–Oriented Programming*, B. Shriver and P. Wegner, eds., MIT Press, Cambridge, MA, pages 165–188, 1987.

[18] W. Tracz, ed., *Tutorial, Software Reuse: Emerging Technology*, IEEE Computer Society Press, Los Angeles, CA, 1988.

[19] B. C. Vickery, *Faceted Classification: A Guide to Construction and Use of Special Schemes*, Aslib, London, 1960.

closeness metric. For example, if the facet value "editor" is closer to "word processor" in terms of the metric than any other value in any facet, then the system poses the query with the modified tuple

(format, text, file, file handler, editor, *)

and continues in this manner until a hit is obtained.

Although this appears to be a reasonable solution to the problem of continued searches, the difficulty lies in the need to assign meaningful closeness values to pairs of facet values. With a large collection of values, this is a daunting task.

4.2. Lattice–Based Classification

Lattice–based faceted classification extends simple faceted classification by organizing an arbitrary number of facets and n–tuples into a lattice [5]. As shown in figure 6, there are four sublattices comprising the complete type lattice, corresponding to the types generated by facet sets, functions, ADTs, and tuples. In addition, the universal type, $\top$, and the void type, $\bot$, ensure that a least upper bound and a greatest lower bound, respectively, exist for any two types in the lattice.

Facet_0 characterizes the notion of the empty facet type; it contains no values, but is still a facet. Likewise, Facet characterizes the notion of the set of all possible facet values. The dotted line between them indicates that a number of types appear here in the lattice. In particular, there is a vertex for each member of the power set formed from the elements comprising the facet. Figure 7 shows the lattice for the examples in section 4.1 expanded to show the sublattices for each of the facets.

Function types are bounded above by $\bot \rightarrow \top$, the function type with a void domain and universal range, and are bounded below by $\top \rightarrow \bot$, the function type with a universal domain and void range.

ADT types are bounded above by $\exists\varepsilon.\varepsilon$, the abstract type denoting a hidden type, ε, with no information or operations available, and are bounded below by ADT, the type denoting all possible types with all possible operations.

The tuple sublattice has a structure similar to that of the facets. At the top is the empty tuple type, {}, characterizing a type with no components. At the bottom is Tuple, the tuple type with all possible components. We restrict component types to facet, function, or ADT. Note that restricting queries to only Tuple (with all and only the Facets appearing as components) and allowing * as a default facet value reduces this approach to the of Prieto–Diaz.

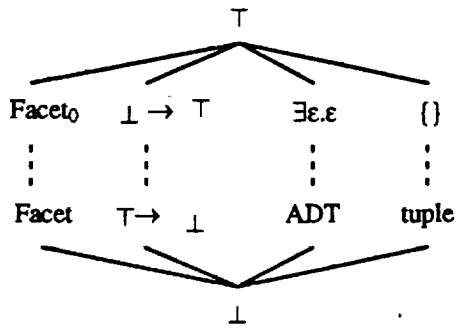


Figure 6.

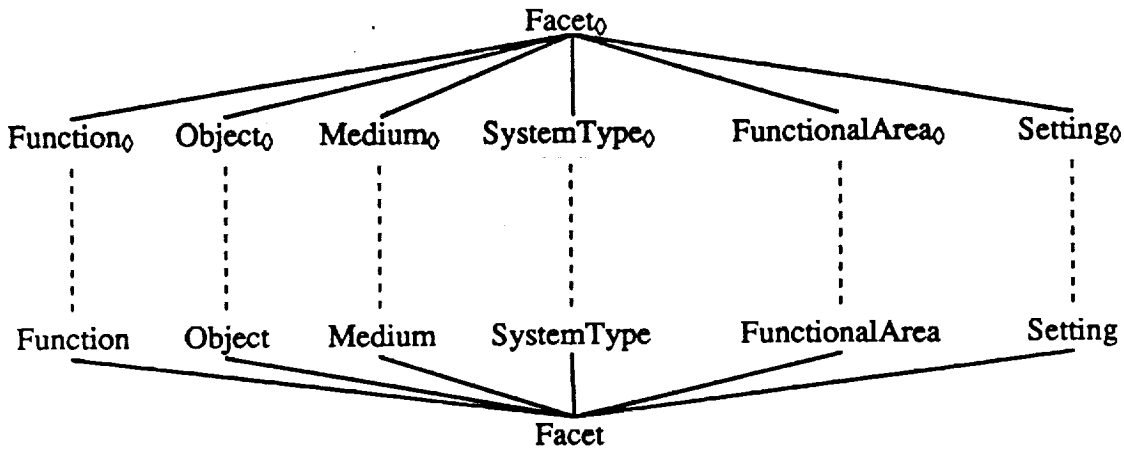


Figure 7. The Sublattice of Facet Sets

4.2.1. Facets vs. Facet Value Sets

Traditional retrieval of individual facet values relies upon maximal conjunction of boolean terms for retrieval of matches on *all* facets and maximal disjunction of boolean terms for matches on *any* facet of an expression. In order to fit the notion of facet into the type lattice, we look at sets of facets. A set of facets corresponds to a conjunction on all of the facets comprising the set. Each set occupies a unique position in the type lattice. We handle disjunction by allowing a given component to occupy multiple lattice positions. Matching occurs on any of the positions, providing the same semantics as disjunction.

Facet values are equivalent to enumeration values. We attach no particular connotation within the type system to a particular facet value. Values are bound to some semantic concept in the problem domain.

The subset relation is our partial order. The least value of this portion of the lattice is the set of all facet values from all facets in the problem domain, denoted by the distinguished name Facet. The greatest value of this portion of the lattice is the empty set, denoted by the distinguished name $Facet_{\emptyset}$. The union operator generates the greatest lower bound. The intersection operator generates the least upper bound.

4.2.2. Domain Interval Subtyping

We adapted the notion of a domain interval [4] to formalize our notion of facet value sets [6,5]. In [4] a subtype was smaller than its supertype; here the reverse is true, a subtype is a larger collection of values than its supertype.

A *domain interval* is a type qualification that explicitly denotes the valid subrange(s) for a base type. Assume that t is a base type ordered by ≤ (the ordering may be arbitrary). A domain that is (inclusively) delimited by two values, a and b, is denoted $t_{(a...b)}$. Intervals made up of more than a single continuous value range are denoted by a set of ranges; for example, $t_{(a...b,\ c...d,\ e)}$ denotes the interval that includes the subinterval a through b inclusive, the subinterval c through d inclusive, and the singleton value e. The singleton range e is equivalent to e...e. When we use such notation we intend that a ≤ b and c ≤ d, but not necessarily

```
1)      → TOI
2)      TOI × Integer → TOI
3)      TOI → TOI
4)      TOI → Integer
5)      TOI → Boolean
```

Figure 9 – Operation Partitions

ticularly when a candidate component's author chose misleading operation names. Figure 9 shows the five operation partitions for figures 1–3 and figure 8.

Singleton operation partitions are unambiguous, since there can be but a single binding possible between the query operation and the candidate operation. Hence, there is only a single binding possible between each of specifications in figures 1–3, since each of the partitions contains a single operation.

Operation partitions containing more than one operation *are* ambiguous, and using (11), contribute a proportional increase in the number of alternative bindings. Figure 8 has two operations in operator partition 4), Top and Depth; hence, the two alternative bindings discussed above.

6. Conclusions

Our approach merges traditional vocabulary and syntactic based retrieval mechanisms with the formal semantics of algebraic specification. Neither retrieval mechanism in isolation is sufficient to completely address the entire problem. Perhaps the most surprising result of this work was our realization concerning the fuzziness of even formal specifications, due to the ambiguity of the terms used in those specifications. This prompted the initiation of work in the application of neural networks to the problem [7].

We are still refining the approach described in this paper. Two specific avenues of research include refining partition equivalence and exploring fragmentary signatures. The current definition of partition equivalence does not adequately address parametric polymorphism, and therefore does not handle components that are instantiations of generic ADTs as well as it handles the generics themselves. Fragmentary signatures, signatures that only partially characterize an ADT, hold excellent promise in supporting the use of our retrieval mechanism in the incremental construction of software from a mix of newly–written code and reused components.

References

[1] J. A. Bergstra, J. Heering, and P. Klint, eds, *Algebraic Specification*, Addison–Wesley, 1989.

[2] L. Cardelli and P. Wegner, "On Understanding Types, Data Abstraction, and Polymorphism," *ACM Computing Surveys*, vol. 17, no. 4, pages 471–522, December, 1985.

[3] H. Ehrig and B Mahr, *Fundamentals of Algebraic Specifications 1*, Springer–Verlag, 1985.

[4] D. Eichmann, *Polymorphic Extensions to the Relational Model*, Ph.D. dissertation, Dept. of Computer Science, The University of Iowa, Iowa City, IA, August 1989.

[5] D. Eichmann, "A Hybrid Approach to Software Repository Retrieval: Blending Faceted Classification and Type Signatures," *Third International Conference on Software Engineering and Knowledge Engineering*, Skokie, IL, June 27–29, 1991.

[6] D. Eichmann and J. Atkins, "Design of a Lattice–Based Faceted Classification System," *Second International Conference on Software Engineering and Knowledge Engineering*, Skokie, IL, pages 90–97, June 21–23, 1990.

[7] D. Eichmann and K. Srinivas, "Neural Network–Based Retrieval from Software Reuse Repositories," *CHI'91 Workshop on Neural Networks and Pattern Recognition in Human–Computer Interfaces*, New Orleans, LA, April 28, 1991.

[8] J. V. Guttag and J. J. Horning, "The Algebraic Specification of Abstract Data Types," *Acta Informatica*, vol. 10, pages 27–52, 1978.

[9] W. P. Jones, "On the Applied use of Human Memory Models: The Memory Extender Personal Filing System," *Int. Journal of Man–Machine Studies*, vol. 25, no. 2, pages 191–228, August, 1986.

[10] D. Kapur and H. Zhang, "RRL: A Rewrite Rule Laboratory," *Ninth International Conference on Automated Deduction (CADE–9)*, Argonne, IL, May, 1988.

[11] R. Prieto–Diaz, P. Freeman, "Classifying Software for Reusability," *IEEE Software*, vol. 4, no. 1, pages 6–16, 1987.

[12] R. Prieto–Diaz, "Implementing Faceted Classification for Software Reuse," *Communications of the ACM*, vol. 34, no. 5, pages 80–97.

[13] A. Snyder, "Inheritance in the Development of Encapsulated Software Components," *Research Directions in Object–Oriented Programming*, B. Shriver and P. Wegner, eds., MIT Press, Cambridge, MA, pages 165–188, 1987.

[14] B. Weide, W. Ogden, S. Zweben, "Reusable Software Components," *Advances in Computers*, M. C. Yovits, ed., Academic Press, 1991.

[15] M. Wirsing, "Algebraic Specifications," *Handbook of Theoretical Computer Science, vol. B*, J. van Leeuwen, ed., MIT Press, 1991.

Neural Network-Based Retrieval from Software Reuse Repositories*

David Eichmann & Kankanahalli Srinivas
Department of Statistics & Computer Science
West Virginia University
Morgantown, WV 26506
eichmann/srini@a.cs.wvu.wvnet.edu

October 7, 1991

Overview. A significant hurdle confronts the software reuser attempting to select candidate components from a software repository – discriminating between those components without resorting to inspection of the implementation(s). We outline an approach to this problem based upon neural networks which avoids requiring the repository administrators to define a conceptual closeness graph for the classification vocabulary.

1 Introduction

Reuse has long been an accepted principle in many scientific disciplines. Biologists use established laboratory instruments to record experimental results; chemists use standardized measuring devices. Engineers design based upon the availability of components that facilitate product development. It is unreasonable to expect an electrical engineer to design and develop the transistor from first principles every time one is required.

Software engineers, however, are frequently guilty of a comparable practice in their discipline. The reasons for this are as varied as the environments in which software is developed, but they usually include the following:

*To appear in *Neural Networks and Pattern Recognition in Human Computer Interfaces*, R. Beale and J. Findlay (eds.), Ellis Horwood Ltd., West Sussex, UK, due out March, 1992.

- a lack of development standards;
- the *not invented here* syndrome;
- poor programming language support for the mechanical act of reuse; and
- poor support in identifying, cataloging, and retrieving reuse candidates.

The first three items involve organization mentality, and will not be addressed here.[1] We instead focus upon the final item in this list, the nature of the repository itself, and more specifically upon the mechanisms provided for classification and retrieval of components from the repository.

The complexity of non-trivial software components and their supporting documentation easily qualifies reuse as a "wicked" problem – frequently intractable in both description and solution. We describe an approach that we are currently exploring for making classification and retrieval mechanisms more efficient and natural for the software reuser. This approach centers around the use of neural networks in support of imprecise classification and querying.

2 The Problem

A mature software repository can contain thousands of components, each with its own specification, interface, and typically, its own *vocabulary*. Consider the signatures presented in Figures 1 and 2 for a stack of integers and a queue of integers, respectively.

$$\begin{aligned} \text{Create: } &\Longrightarrow \text{Stack} \\ \text{Push: Stack} \times \text{Integer} &\Longrightarrow \text{Stack} \\ \text{Pop: Stack} &\Longrightarrow \text{Stack} \\ \text{Top: Stack} &\Longrightarrow \text{Integer} \\ \text{Empty: Stack} &\Longrightarrow \text{Boolean} \end{aligned}$$

Figure 1: Signature of a Stack

[1]Concerning language support – there *are* languages which readily support reuse, but they must be available to the programmers. Consider for a moment the inertia exhibited by FORTRAN and COBOL in commercial data processing. The very existence of such large bodies of code in languages ill-suited for reuse acts as an inhibitor for the movement of organizations towards better suited languages.

Create: $\Longrightarrow$ Queue
Enqueue: Queue $\times$ Integer$\Longrightarrow$ Queue
Dequeue: Queue $\Longrightarrow$Queue
Front: Queue $\Longrightarrow$ Integer
Empty: Queue $\Longrightarrow$ Boolean

Figure 2: Signature of A Queue

These signatures are isomorphic up to renaming, and thus exemplify what we have come to refer to as the *vocabulary problem*. Software reusers implicitly associate distinct semantics with particular names, for example, pop and enqueue. Thus, by the choice of names, a component developer can mislead reusers as to the semantics of components, or provide no means of discriminating between components. Figure 3, for example, appears to be equally applicable as a signature for both stack and queue, primarily due to the neutral nature of the names used.

Create: $\Longrightarrow$ Sequence
Insert: Sequence $\times$ Integer $\Longrightarrow$ Sequence
Remove: Sequence $\Longrightarrow$ Sequence
Current: Sequence $\Longrightarrow$ Integer
Empty: Sequence $\Longrightarrow$ Boolean

Figure 3: Signature of a Sequence

3 Software Classification

Retrieval mechanisms for software repositories have traditionally provided some sort of classification structure in support of user queries. Keyword-based retrieval is perhaps the most common of these classification structures, but keywords are ill-suited to domains with rich structure and complex semantics. This section lays out the principle representational problems in software classification and selected solutions to them.

3.1 Literary Warrant

Library scientists use *literary warrant* for the classification of texts. Representative samples drawn from the set of works generate a set of descriptive terms, which in turn generate a classification of the works as a whole. The adequacy of the classification system hinges a great deal on the initial choice of samples.

With appropriate tools, literary warrant in software need not restrict itself to a sample of the body of works. Rather, it can examine each of the individual works in turn, providing vocabularies for each of them. This may indeed be *required* in repositories where the component coverage in a particular area is sparse.

3.2 Conceptual Closeness

The vocabulary of terms built up through literary warrant typically contains a great deal of semantic overlap words whose meanings are the same, or at least similar. For instance, two components, one implementing a stack and the other a queue might both be characterized with the word insert, corresponding to push and enqueue, respectively, as discussed in section 2.

Synonym ambiguity is commonly resolved through the construction of a restricted vocabulary, tightly controlled by the repository administrators. Repository users must learn this restricted vocabulary, or rely upon the assistance of consultants already familiar with it. It is rarely the case, however, that the choice is between two synonyms. More typically it is between words which have similar, but distinct, meanings (e.g., insert, push, and enqueue, as above).

3.3 Algebraic Specification

While not really a classification technique, algebraic specification techniques (e.g., [GH78]) partially (and unintentionally) overcome the vocabulary problem through inclusion of behavioral axioms into the specification. The main objection to the use of algebraic specifications in reuse is the need to actually *write* and *comprehend* the specifications. The traditional examples in the literature rarely exceed the complexity of the above Figures. Also, algebraic techniques poorly address issues such as performance and concurrency.

A repository containing algebraic specifications depends upon the expertise of the reusers browsing the repository; small repositories are easily understood whereas it is unreasonable to require a reuser to examine all components in a large repository for suitability.

3.4 Basic Faceted Classification

Basic faceted classification begins by using domain analysis (aka literary warrant) "to derive faceted classification schemes of domain specific objects." The classifier not only derives terms for grouping, but also identifies a vocabulary that serves as the values that populate those groups. From the software perspective, the groupings, or *facets* become a taxonomy for the software.

Prieto-Díaz and Freeman identified six facets: function, object, medium, system type, functional area, and setting [PDF87]. Each software component in the repository has a value assigned for each of these facets. The software reuser locates software components by specifying facet values that are descriptive of the software desired. In the event that a given user query has no matches in the repository, the query may be relaxed by wild-carding particular facets in the query, thereby generalizing it.

The primary drawback in this approach is the flatness and homogeneity of the classification structure. A general-purpose reuse system might contain not only reusable components, but also design documents, formal specifications, and perhaps vendor product information. Basic faceted classification creates a single tuple space for all entries, resulting in numerous facets, tuples with many "not applicable" entries for those facets, and frequent wildcarding in user queries.

A number of reuse repository projects have incorporated faceted classification as a retrieval mechanism (e.g., [Gue87][Atk]), but they primarily address the vocabulary problem through a keyword control board, charged with creating a controlled vocabulary for classification.

Gagliano, et. al. computed conceptual closeness measures to define a semantic distance between two facet values [GOF+88]. The two principle limitations to this approach are the static nature of the distance metrics and the lack of inter-facet dependencies; each of the facets had its own closeness matrix.

3.5 Lattice-Based Faceted Classification

Eichmann and Atkins extended basic faceted classification by incorporating a lattice as the principle structuring mechanism in the classification scheme [EA90]. As shown in Figure 4, there are two major sublattices making up the overall lattice.

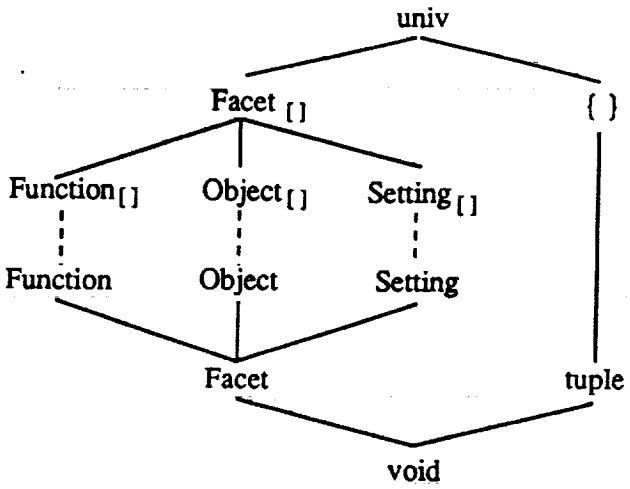


Figure 4: The Type Lattice

On the left is the sublattice comprised of sets of facet values (for clarity, shown here with only three facets), partially ordered by the subset relation. The $Facet_{[]}$ vertex in the lattice represents the empty facet set, while the *Facet* vertex represents the set of all facet values in the classification scheme. Each member of the power set of all facet values falls somewhere within this sublattice.

On the right is the tuple sublattice, containing facet set components, and partially ordered by the subtype relation [Eic89]. The vertex denotes the empty tuple. The *tuple* vertex denotes the tuple containing all possible facet components, with each component containing all the values for that facet. Adding facet values to a component or adding a new component to a tuple instance moves the tuple instance down through the lattice.

Queries to a repository supporting lattice-based faceted classification are similar to those to one supporting basic faceted classification, with two important distinctions – query tuples can mention as many or as few facets as the reuser wishes, thereby avoiding the need for wildcarding, and classifiers can similarly classify a given component with as many or as few facets as are needed for precise characterization of the component.

Lattice-based faceted classification avoids conceptual closeness issues through the specification of sets of facet values in the classification of components. If there are a number of semantically close facet values that all characterize the component, all are included in the facet instance for that component. This avoids the need to generate closeness metrics for facet values, but it also may result in reuser confusion about just what the component does.

3.6 Towards Adaptive Classification and Retrieval

The principle failing in the methods described so far is the static nature of the classification. Once a component has been classified, it remains unchanged until the repository administrators see fit to change it. This is unlikely to occur unless those same administrators closely track reuser retrieval success, and more importantly, retrieval *failure* – particularly in those cases where there are components in the repository matching reuser requirements, but those components were not identified during the query session.

Manual adjustment of closeness metrics becomes increasingly unreasonable as the scale of the repository increases. The number of connections in the conceptual graph is combinatorially explosive. The principle design goal in our work is the creation of an *adaptive* query mechanism – one capable of altering its behavior based upon implicit user feedback. This feedback appears in two guises; *failed queries,* addressed by widening the scope of the query; and *reuser refusals,* cases where candidate components were presented to the reuser, but not selected for retrieval. The lattice provides a nice structure for the former, but a different approach is required for the latter.

4 Our Approach

We are currently designing a new retrieval mechanism using previous work described in [EA90] as a starting point, and employing neural networks to address the vocabulary and refusal problems. The motivations behind using neural networks include:

- **Associative Retrieval from Noisy and Incomplete Cues:** Traditional methods for component retrieval are based on strict pattern matching methods such as unification. In other words, the query should contain exact information about the component(s) in the repository. Since exact information about components is usually not known, queries fail in cases where exact matching does not occur. Associative retrieval based on neural networks uses relaxation, retrieving components based on partial/approximate/best matches. This is sometimes referred to as *data fault tolerance* and is ideally suited for our problem domain.

- **Classification and Optimization by Adaptation:** In approaches using the conceptual closeness measure, the problem of defining correlations between various components and assigning a numerical correlation value rests

upon the designer or the administrator of the repository. Designers idiosyncratically arrive at these correlations and their values, which may not be appropriate from the perspective of the software retriever/reuser. It is our belief that the best way to arrive at these correlations and their values is for the system to learn them in responding to user queries.

We also intend to use another adaptation strategy for optimizing the retrieval of similar repetitive queries. Since in most situations, reusers repeatedly issue similar queries, the system will adapt to these queries by weight adjustment. The weight adjustment will settle the relaxation process quickly in response to these repetitive queries and hence result in faster retrieval. The effect here is similar to that of *caching* frequently issued queries. Note, however, that once the system has learned that two concepts are conceptually close, we want it to remember this, irrespective of how often the reusers inquire about it.

- **Massive Parallelism:** The neurocomputing paradigm is characterized by *asynchronous, massively parallel, simple* computations. Since neural networks are massively parallel, retrieval from large repositories is possible, using the fast associative search techniques that are natural and inherent in these networks.

5 System Architecture

In this section, we describe some of the potential neural-network architectures and discuss their strengths and limitations in employing them for our task.

5.1 Hopfield Networks

These networks can be used as content-addressable or associative memories. Initially the weights in the network are set using representative samples from all the exemplar classes. After this initialization, the input pattern I is presented to the network. The network then iterates and converges to a output. This output represents the exemplar class which matches the input pattern best.

Although this network has many properties that are desirable for our system, some of the serious limitations in our context include:

1. The networks have limited capacity [Lip87] and may converge to novel spurious patterns.

2. They result in unstable exemplar patterns if many bits are shared among multiple exemplar patterns.

3. There are no algorithms to incrementally train these networks, i.e., to adjust the initial weights in a manner that creates a specific alteration in subsequent query responses. This is important for our application, since we seek an architecture capable of adapting over time to user feedback.

5.2 Supervised Learning Algorithms

Many good *supervised* learning algorithms exist, including backpropagation [RHW86], cascade correlation and others, but they cannot be used in this context because our problem requires an *unsupervised* learning algorithm. Hence, we are investigating unsupervised learning architectures, such as Adaptive Resonance Theory (ART) [Gro88].

5.3 ART

ART belongs to a class of learning architectures known as *competitive* learning models [Gro88][CG88]. The competitive learning models are usually characterized by a network consisting of two layers L_1 and L_2. The input pattern I is fed into layer L_1 where it is normalized. The normalized input is fed forward to layer L_2 through the weighted interconnection links that forms an adaptive filter. Layer L_2 is organized as a *winner-take-all network* [FB82][Sri91][BSD90]. The network layer L_2 is usually organized as a mutually inhibitory network wherein each unit in the network inhibits every other unit in the network through a value proportional to the strength of its activation. Layer L_2 has the task of selecting the network node a_{max}, receiving the maximum total input from L_1. The node a_{max} is said to cluster or code the input pattern I.

In the ART system the input pattern I is fed in to the lower layer L_1. This input is normalized and is fed forward to layer L_2. This results in a network node n_{max} of layer L_2 being selected by virtue of it having the maximum activation value among all the nodes in the layer. This node n_{max} represents the hypothesis H put forth by the network about the particular classification of the input I. Now a *matching* phase occurs wherein the hypothesis H and the input I are matched, with the quality of the required match controlled by the *vigilance parameter*.

If the quality of match is worse than the value specified in the vigilance parameter, a mismatch occurs and the layer L_2 is reset thereby deactivating node n_{max}. The input I activates another node and the above process recurs, comparing another hypothesis or forming a new hypothesis about the input pattern I. New

hypotheses are formed by learning new classes and recruiting new uncommitted nodes to represent these classes.

Some of the properties of ART that makes it an potential choice for our task include

1. Real-time (on-line) learning;
2. Unsupervised learning;
3. Fast adaptive search for best match as opposed to strict match; and
4. Variable error criterion which can be fine-tuned by appropriately setting the *vigilance* parameter.

However, one of the limitations of ART for our particular task arises from its inability to distinguish the queries for particular components by users, from the component classes which form the exemplar classes. Another limitation arises from the fact that only one exemplar class is chosen at a time which represents the *best* match, rather than choosing a collection of *close* matches for reuser consideration.

Our proposed system will operate in two phases. The first, *loading* phase populates the repository with components. The second, *retrieval* phase identifies candidate components in response to user queries. The distinguishing factor between the two phases is the value of the vigilance parameter. In the loading phase, the system will employ a high vigilance value. This ensures the formation of separate categories for each of the components in the repository. In the retrieval phase, the system will employ a low vigilance value, thereby retrieving components that best match the query.

We also intend to modify the winner-take-all network layer of the ART to choose k winners instead of one. This is extremely useful in our context because there may be multiple software components which meet the user specifications. The software reuser may select a subset $m \leq k$ of these components based upon requirements. The system should associate these m components with the user query and retrieve them for subsequent queries having similar input specifications. This can be achieved by associating small initial weights on the lateral links of the winner-take-all network and modifying them appropriately based on user feedback (i.e., reuser refusals).

6 Discussion

6.1 Our Placement in the User-Based Framework

Discussions in the workshop placed our work in the region of user intention / no feedback in the user-based framework. Upon further reflection, we have slightly altered our perspective. While this placement is certainly proper in the strict context of a single user query, it is not accurate in the broader context of a community of users accessing the repository over time.

As the system is rewarded for providing true hits to users and punished for providing false hits, there is a consensual drift, providing feedback for subsequent user queries. Thus, viewing the amortized effect of user behavior, rather than the immediate effect of user behavior, our system shifts down towards passive observation and left towards immediate feedback.[2] The net result is that our system occupies two distinct points in the framework, one for the semantics involved in the immediate query query and one for the semantics involved in the aggregate behavior of the repository over time.

6.2 The Relationship to Gestural Recognition

Beale [BE], Rubine [Rub], and Zhao [Zha], the other occupants of the Novel Input category of the task-based framework, respectively address sign language recognition, drawing geometric figures, and diagram editing – all interpreting imprecise human gestures and mapping them to a precise application domain. They all address the inability of humans to accurately repeat physical movement.

Our mechanism, on the other hand, accepts a precisely phrased user query and adapts it to an imprecise application domain. Ignoring the issue of poor typing skills, our user community can accurately repeat a given user intention (query) any number of times, and we know exactly what that intention is. The challenge in our domain occurs when that intention has no exact match in the system. It's similar to Rubine's system offering to draw a square or a hexagram (or perhaps even a five-sided star) when the user gestured a pentagram, but the system had no training in pentagram gestures.

[2] or more precisely, non-immediate feedback.

6.3 Directions for Future Research

Options available to us at this point in our work lie in two general directions, further extending repository semantics and exploring the application of neural networks to these types of application domains.

With respect to the former, the classification scheme described here is restricted to facets and tuples containing facets. In other work, the classification scheme was first extended to include signatures for abstract data types [Eic91a] and then further extended to support axioms in a second phase in the query process [Eic91b]. A merger of that work with that described here has appeal – particularly the imprecise matching of signatures.

With respect to the latter, we are interested in studying the tradeoffs between individual user adaptation versus the consensual adaptation described above. These two actually are the extremes in a continuum of user groupings. This coupled with an additional dimension of user expertise forms a state space of user behavior where the system might more heavily weight certain semantic connections for experts and other semantic connections for novices. This will require the development of new algorithms for relaxation.

7 Conclusions

Our approach extends previous work in component retrieval by incrementally adapting the conceptual closeness weights based upon actual use, rather than an administrator's assumptions. Neural networks provide a quite suitable framework for supporting this adaptation. Reuse repository retrieval provides a unique and challenging application domain for neural networking techniques.

This approach effectively adds an additional dimension to the conceptual space formed by the type lattice. This additional dimension allows traversal from one vertex to another using the adapted closeness weights derived from user activity, rather than the partial orders used in defining the lattice. The resulting retrieval mechanism supports both well-defined lattice-constrained queries and ill-defined neural-network constrained queries in the same framework.

References

[Atk] J. Atkins. private communication.

[BE] R. Beale and A. D. N. Edwards. Gestures and neural networks in human-computer interaction. in this volume.

[BSD90] J. Barden, K. Srinivas, and D. Dharmavaratha. Winner-take-all networks: Time-based versus activation-based mechanisms for various selection goals. In *IEEE International Conference on Circuits and Systems*, pages 215–218, New Orleans, LA, May 1990.

[CG88] G. A. Carpenter and S. Grossberg. The art of adaptive pattern recognition by a self-organizing neural network. *IEEE Computer*, 21(3):77–88, 1988.

[EA90] D. Eichmann and J. Atkins. Design of a lattice-based faceted classification system. In *Second International Conference on Software Engineering and Knowledge Engineering*, pages 90–97, Skokie, IL, June 1990.

[Eic89] D. Eichmann. *Polymorphic Extensions to the Relational Model.* PhD dissertation, The University of Iowa, Dept. of Computer Science, August 1989.

[Eic91a] D. Eichmann. A hybrid approach to software repository retrieval: Blending faceted classification and type signatures. In *Third International Conference of Software Engineering and Knowledge Engineering*, pages 236–240, Skokie, IL, June 1991.

[Eic91b] D. Eichmann. Selecting reusable components using algebraic specifications. In *Second International Conference on Algebraic Methodology and Software Technology*, pages 37–40, Iowa City, IA, May 1991. to appear in AMAST'91, Workshops in Computing Series, Springer-Verlag.

[FB82] J. A. Feldman and D. H. Ballard. Connectionist models and their properties. *Cognitive Science*, 6:205–254, 1982.

[GH78] J. V. Guttag and J. J. Horning. The algebraic specification of abstract data types. *Acta Informatica*, 10:27–52, 1978.

[GOF+88] R. Gagliano, G. S. Owen, M. D. Fraser, K. N. King, and P. A. Honkanen. Tools for managing a library of reusable ada components. In *Workshop on Ada Reuse and Metrics*, Atlanta, GA, June 1988.

[Gro88] S. Grossberg, editor. *Neural Networks and Natural Intelligence.* MIT Press, Cambridge, MA, 1988.

[Gue87] E. Guerrieri. On classification schemes and reusability measurements for reusable software components. SofTech Technical Report IP-256, SofTech, Inc., Waltham, MA, 1987.

[Lip87] R. P. Lippmann. An introduction to computing with neural nets. *IEEE ASSP Magazine*, 4:4–22, 1987.

[PDF87] R. Prieto-Díaz and P. Freeman. Classifying software for reusability. *IEEE Software*, 4(1):6–16, 1987.

[RHW86] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning internal representations by error propagation. In D. E Rumelhart and J. L. McClelland, editors, *Parallel Distributed Processing*, volume 1. MIT Press, Cambridge, MA, 1986.

[Rub] D. Rubine. Criteria for gesture recognition technologies. in this volume.

[Sri91] K. Srinivas. *Selection in Massively Parallel Connectionist Networks.* PhD dissertation, New Mexico State University, Dept. of Computer Science, 1991.

[Zha] R. Zhao. On the graphical gesture recognition for diagram editing. in this volume.

Inheritance for Software Reuse: The Good, The Bad, and The Ugly

Murali Sitaraman and David Eichmann

Dept. of Statistics and Computer Science
West Virginia University
Morgantown, WV 26506
(murali, eichmann)@cs.wvu.wvnet.edu

Abstract

Inheritance is a powerful mechanism supported by object-oriented programming languages to facilitate modifications and extensions of reusable software components. This paper presents a taxonomy of the various purposes for which an inheritance mechanism can be used. While some uses of inheritance significantly enhance software reuse, some others are not as useful and in fact, may even be detrimental to reuse. The paper discusses several examples, and argues for a programming language design that is selective in its support for inheritance.

Keywords: extensions, implementation, inheritance, reusable software components, specification

1 Introduction

Inheritance has been widely recognized as an important mechanism for constructing new reusable software components from existing components [Liskov 87, Meyer 88]. This paper proposes a taxonomy for inheritance-based reuse. Some members of this taxonomy permit effective reuse and must be supported by object-oriented programming languages. However, there are other uses of inheritance that do not enhance reuse, and may even be detrimental to reuse. A language must, therefore, be selective in its support for inheritance.

2 A Framework for Discussion

We will use the "3C reference model" (for reusable software components) as the basis for our taxonomy in this paper [Edwards 90, Latour 90, Tracz 90b]. This model is the result of the discussions at the Reuse in Practice Workshop (July 1989) and the Workshop on Methods and Tools for

Reuse (June 1990). The 3C model associates three key ideas with reusable software components as summarized in [Weide 91]:

Concept An abstract (formal) specification explaining (precisely) what functionality is provided by a software piece. without saying how the functionality can be realized.

Content (for a concept) A piece of code that (precisely) describes the data structures and algorithms for implementing (in a formal, programming language) the concept.

Context A statement (precisely) explaining the environment (using formal notations) in which a concept or content is presented.

Several contents may implement the same concept. They will all be identical with respect to their functionality, but may be different with respect to their performance behaviors (e.g., space or time characteristics) To use a component, a client (user) needs to understand only its concept. The functional correctness of the client program depends only on this concept [Parnas 72]. The client will remain unaffected even if it switches from one content of the concept to another. These observations have important implications for modification and maintenance of software built from reusable components. We have used a similar model in our research to characterize the nature of a components industry that would evolve when current reuse efforts prove successful [Muralidharan 90b, Sitaraman 90, Weide 91].

3 A Classification of Uses of Inheritance

Inheritance can be used, in the above framework, to extend (or modify), and thus, reuse each aspect of a software component - concept, content, and context. This section presents a classification of such uses of inheritance. We restrict our attention in this paper to inheritance of concepts and contents alone. It is important to note that our classification has nothing to do with the actual inheritance mechanisms supported in object-oriented languages; it deals only with the possible uses of inheritance.

3.1 A Classification Scheme

The critical issues in inheritance mechanisms from a reuse perspective are who inherits, what is inherited, and what can be done with that which is inherited. We consider each of these issues in turn. This discussion supports both single and multiple inheritance.

(i) Who inherits and from whom

Specification inheritance occurs when parents are concepts. Implementation inheritance occurs when parents are contents. These definitions are similar in spirit to those found in [LaLonde 89]. The heir can be either a concept or a content for either specification or implementation inheritance. The only combination that is not meaningful (based on our definitions) is inheritance of a content by a concept.

(ii) What parts are inherited

We focus our attention here only on formally defined concepts and contents that implement these concepts. A formal concept for a data abstraction has two parts: the abstract model(s) that describes the type(s) provided by the concept, and the abstract specifications of the operations on the provided type(s). (When a concept provides only a procedural abstraction, only the second part is present.) The appendix describes an example concept - a formal specification of a stack data abstraction.

A content for a concept defining a data abstraction also has two parts: the representation(s) of the provided type(s), and the code for the provided operations.

An heir may selectively inherit only parts of a concept or content.

(iii) The mode of inheritance

An heir may inherit parts of a concept or a content for read only or for redefining purposes. When a heir redefines a part of its parent, the re-definition may or may not be "compatible" with its parent. The definition of compatibility depends on what is inherited: usually it involves restricting the domain of one or more inherited types.

3.2 Specification Inheritance - Inheritance of a Concept

A concept can be inherited by either another concept or by a content. (When multiple concepts are inherited, different concepts could be affected differently.)

3.2.1 Inheritance by a concept

First, we define what it means for an heir to compatibly redefine its parent's parts. The abstract model A of an heir is compatible with the corresponding model B of its parent, only if the parent concept is unaffected by substituting A for B. (For. example. the heir's model should satisfy the invariants in the parent concept.) An operation P in an heir is compatible with the corresponding operation Q in its parent. only if P's pre-condition is no stronger than Q's and P's post-condition is no weaker than Q's.

Because few object-oriented programming languages have included rigorous formal specifications, the issues raised by some of these combinations have not been explored in the community. In table 1, the meaningful combinations are marked with a •. For want of space, we discuss the meaning and relevance of only some of these combinations here.

(i) Read only - both abstract model(s) and operations

Table 1: Inheritance a concept by another concept

Mode	*None*	*Model*	*Operations*	*Both*
Read only				•
Read and compatible redefine			•	•
Read and incompatible redefine		•	•	•

This is probably the most common mode for specification-based extensions. For example, a basic stack concept may provide the operations push, pop, and is-empty. This concept may be extended to include, say, an operation to reverse a stack. The typical reason for extending a concept is either that the original concept is not sufficiently complete or that it is in the developmental stage. In [Sitaraman 91], we have argued for a reason to extend even well-designed concepts for building efficient implementations. Without the ability to inherit a concept, this is impossible to do. This use of inheritance can enhance reuse and programming languages must support this possibility.

(ii) Read all and compatibly redefine - operations

Sometimes, it may be essential to create a new concept by modifying the specifications of an existing concept. If the changes are compatible (according to the definitions of compatibility in this section) with the specifications in the original concept, then the new concept can be used wherever the original concept was being used. For example, a stack concept can inherit from a bounded stack concept, and relax the pre-condition on the push operation. Intuitively, an unbounded stack can be used wherever a bounded stack can be used.

(iii) Read all and incompatibly redefine - operations

If a stack concept is already defined, and someone extends it to be a bounded stack, this will be the case. In this case, the model of the stack has to be extended to include a bound. In addition, while the original stack will have no pre-condition for the push operation, the heir concept will have one. This is incompatible because the heir has a stronger pre-condition. Intuitively, a bounded stack cannot be used where an unbounded stack was previously used. If the abstract model of a type is redefined, the specifications of most, if not all, operations will have to be redefined. In this case, inheritance may result in some, but not in significant reuse.

3.2.2 Inheritance by a content

When a concept is inherited by a content, only few combinations are meaningful.

(i) Read only - both abstract model(s) and operations

This is the most normal case of concept inheritance by content. To implement a concept, a content must inherit it for read only purposes. Of course, more than one content may inherit the same concept in this mode, resulting in multiple implementations of a concept. This is an important use of inheritance [Meyer 88, Sitaraman 90], and is crucial for the evolution of a successful components industry.

Table 2: Inheritance of a concept by a content

Mode	*None*	*Model*	*Operations*	*Both*
Read only				•
Read and compatible redefine			•	
Read and incompatible redefine			•	

(ii) Read all and compatibly redefine - operations

Sometimes, an implementation of an operation may require fewer pre-conditions than stated in its specifications and ensure more post-conditions. In this case, the operation does more than what the specification of the operation needs it to do. For example, an operation may reclaim unused storage even if it is not explicitly stated in its specification.

(iii) Read all and incompatibly redefine - operations

This is an implementation where the code for some operations do not provide the behavior specified in the concept. In otherwords, this content does not correctly implement its concept, i.e., it is incorrect. Clearly, this is a bad use of inheritance.

3.3 Implementation Inheritance - Inheritance of a Content

A content can be inherited only by another content. The concept of the parent and the heir may or may not be the same. Just as in the case of a concept, a content may be inherited in three different modes. A content redefines a representation compatibly only if the heir's representation when used in the place of the parent's representation leaves the parent content unaffected. A compatible redefinition of an operation does not violate the specification of the operation in the parent content's concept. Content inheritance may also be selective. (When multiple contents are inherited, different contents could be affected differently.)

(i) Read only - both representation(s) and operations

Apparently, this use of content inheritance is to permit an heir take advantage of the otherwise hidden details of another content. For a well-designed component, providing "sufficiently complete" functionality, all essential details of the content may be accessed by calling the operations in its concept. This use of inheritance helps in avoid a few procedure calls, but clearly violates the principle of information hiding. This can lead to serious pitfalls, including poor developmental independence and maintainability [Muralidharan 90a, Raj 90]. This may, however, be a useful way of keeping track of different versions of the same content.

(ii) Read all and compatibly redefine - operations

This case of content inheritance probably is most useful to keep track of the different versions of an evolving content.

(iii) Read all and compatibly redefine - both rep. and operations

Table 3: Inheritance of a content by a content

Mode	*None*	*Rep.*	*Operations*	*Both*
Read only		•		•
Read and compatible redefine	•	•	•	•
Read and incompatible redefine	•		•	•

Sometimes, when a new concept is created by compatibly redefining an existing concept, it may be possible to create a content for the new concept by compatibly redefining a content of the original concept. The new content, in this case, will also be a content for the original concept.

Incompatible redefinitions may be useful in some rare cases. It must be noted, however, that all uses of content inheritance suffer from certain basic problems because their violate information hiding.

4 Discussion

Object-oriented programming languages typically support one mechanism for inheritance that is useful for various purposes. While this is important, we believe the mechanism should be discriminatory and allow only certain uses. We have shown that most uses of specification inheritance are useful and some uses of implementation inheritance may not be desirable. The components of a library that would evolve from discriminatory uses of inheritance will facilitate construction of software systems that are reliable, modifiable, and maintainable.

The work presented here can be formalized, and extended to compare inheritance mechanisms in various languages and the forms of uses that are supported. Also, it is important to identify interesting examples for the various classes, thereby leading to a better understanding of the usefulness of these classes. The present scheme should also be enhanced to account for context inheritance.

References

[Edwards 90] Edwards, S., "The 3C Model of Reusable Software Components," *Third Annual Workshop: Methods and Tools for Reuse*. Syracuse, 1990.

[LaLonde 89] LaLonde, W. R., "Designing Families of Data Types Using Exemplars," *ACM Transactions on Programming Languages and Systems* 11, 2, April 1989, pp. 212-248.

[Latour 90] Latour, L., T. Wheeler, and W. Frakes, "Descriptive and Predictive Aspects of the 3Cs Model: SETA1 working group summary," *Third Annual Workshop: Methods and Tools for Reuse*, Syracuse, 1990.

[Liskov 87] Liskov, B., "Data Abstraction and Hierarchy," *Addendum to the Procs. of OOPSLA 1987*. Orlando, FL, pp. 17-34.

[Meyer 88] Meyer, B., *Object-Oriented Software Construction*, Prentice-Hall, Englewood Cliffs, NJ, 1988.

[Muralidharan 90a] Muralidharan, S., and B. W. Weide, "Should Data Abstraction Be Violated to Enhance Software Reuse?," *Proc. 8th Annual National Conf. on Ada Technology*, ANCOST, Inc., Atlanta, GA, Mar. 1990, 515-524.

[Muralidharan 90b] Muralidharan, S., and B. W. Weide, "Reusable Software Components = Formal Specifications + Object Code: Some Implications," *3rd Annual Workshop: Methods and Tools for Reuse*, Syracuse Univ. CASE Center, Syracuse, NY, July 1990.

[Parnas 72] Parnas, D. L., "On the Criteria to be Used in Decomposing Systems into Modules," *Communications of the ACM* 15, 12, December 1972, 1053-1058.

[Raj 90] Raj, R. K., "Code Inheritance Considered Harmful," *3rd Annual Workshop: Methods and Tools for Reuse*, Syracuse Univ. CASE Center, Syracuse, NY, July 1990.

[Sitaraman 91] Sitaraman, M. and D. Eichmann, *Building and Using Efficient Extensions: An Approach Based on Inheritance*, TR 91-01-02, Dept. of Stat. and Comp. Science, West Virginia University, Morgantown, WV 26506.

[Sitaraman 90] Sitaraman, M., *Mechanisms and Methods for Performance Tuning of Reusable Software Components*, Ph. D. Dissertation, Dept. of Comp. and Info. Science, Ohio State Univ., Columbus, OH. July 1990.

[Tracz 90a] Tracz, W., "Where Does Reuse Start?," *ACM SIGSOFT Software Engineering Notes* 15, 2, pp. 42-46.

[Tracz 90b] Tracz, W., "The Three Cons of Software Reuse," *Third Annual Workshop: Methods and Tools for Reuse*, Syracuse, 1990.

[Weide 91] Weide, B. W., W. Ogden, and S. H. Zweben, "Reusable Software Components," *Advances in Computers*, M. C. Yovits, eds., Academic Press, New York, NY, 1991.

[Weide 86] Weide, B. W., *Design and Specification of Abstract Data Types Using OWL*, OSU- CISRC-TR-86-01, Dept. of Comp. and Info. Science, Ohio State Univ., Columbus, OH, March 1986.

[Wing 90] Wing, J. M., "A Specifier's Introduction to Formal Methods," *IEEE Computer* 23, 9, September 1990, pp. 8-24.

5 Appendix: An Example Concept

Figure 1 shows a concept for a Stack component explained using a model-based specification. For our purposes, it does not matter which specific specification language and/or programming language is used in explaining concepts and contents. The concepts could use any of the formal methods described in [Wing 90]. We have chosen a dialect of RESOLVE [Weide 91].

Here, the type Stack is modeled as a mathematical STRING of Items and the operations are formally specified using mathematical string functions EMPTY and POST. Each operation has been explained using two clauses: a requires clause that states what must be true of the arguments

```
concept Stack_Template (type Item)
        type Stack is modeled by STRING (Item)
                initially  for all s: Stack, s = EMPTY

        operation Push(s: Stack, x: Item)
                ensures  s = POST(s, x) and Item.Init (x)

        operation Pop(s: Stack, x: Item)
                requires s /= EMPTY
                ensures  #s = POST (s, x)

        operation Is_Empty(s: Stack) return Boolean
                ensures  Is_Empty iff s = EMPTY
end Stack_Template
```

Figure 1: Formal Specification of a Stack Abstraction

passed to the operation and an ensures clause that states what will be true of the parameters at the completion of the operation. In the ensures clause, the notation "#x" for a parameter x denotes its incoming value and RxS denotes its value when the operation returns. (In the requires clause, the variables always denote the incoming values.) The specification of Push, for example, states that the value of the returned stack (s) is its incoming value (#s) with the incoming value of x (#x) appended to the end. The returned value of x is an initial value of the type Item.

6 About the Authors

Murali Sitaraman is an Assistant Professor of Computer Science at the West Virginia University. He holds a Ph. D. in Computer Science from The Ohio State University and M. E. (distinction) from the Indian Institute of Science. His current research interests are in data structures and algorithms, programming languages, software reuse, verification and validation, and some aspects of distributed computing. His Internet address is murali@cs.wvu.wvnet.edu.

David Eichmann is currently an Assistant Professor of Computer Science at West Virginia University and heads the Software Reuse Repository Lab (SoRReL). He received his doctorate in computer science from The University of Iowa, and taught in Seattle University's Master's in Software Engineering Program before joining WVU. His research interests focus on software reuse systems, particularly in the representation and retrieval of life cycle artifacts, and on database systems, particularly in type systems for databases. SoRReL is currently supported in part by NASA's RBSE project (previously known as AdaNet).

Supporting Multiple Domains in a Single Reuse Repository

David Eichmann *
SoRReL Group
Dept. of Statistics and Computer Science
West Virginia University
Morgantown, WV 26506
eichmann@cs.wvu.wvnet.edu

Abstract: Domain analysis typically results in the construction of a domain–specific repository. Such a repository imposes artificial boundaries on the sharing of similar assets between related domains. A lattice–based approach to repository modeling can preserve a reuser's domain specific view of the repository, while avoiding replication of commonly used assets and supporting a more general perspective on domain interrelationships.

Keywords: domain analysis, software reuse, faceted classification, type lattices, record subtyping, repository views

1. Introduction

There is an emerging consensus on the importance of domain analysis in the success of a software reuse program [9]. We find it particularly significant that the construction of domain–specific repositories is a natural consequence of domain–specific analysis of various software system assets. These domain–specific repositories provide yet another guise for the NIH (not–invented–here) syndrome, and hence fail to capitalize on possible reuse scenarios that lie in related, but distinct domains.

We propose here that *repositories* should not be domain–specific, but rather that a particular *view* of the repository should be domain–specific, and that this view should be user–adjustable. We use our lattice–based approach to classification [4] to demonstrate how this can be accomplished. Section 2 briefly reviews issues in domain analysis, faceted classification, and the concepts of typing and lattices. Section 3 reviews our lattice–based repository model, followed by a demonstration of domain–specific support in section 4. The paper closes with a discussion and suggestions for future work with section 5.

* This work was supported in part by NASA under cooperative agreement NCC–9–16, and in part by MountainNet, Inc.

2. Background

Our work draws its motivation equally from the areas of domain analysis and type theory. Recent advances in the application of type lattices to database models and knowledge representation provide an excellent formal framework for repository structure.

2.1. Domain Analysis

> "Domain analysis is the process of identifying and organizing knowledge about some class of problems — the problem domain — to support the description and solution of those problems." [1]

The interest in domain analysis reflects its importance to the effective population and use of reuse repositories. There are substantial arguments in favor of the reasoned coverage of a particular software system problem domain, rather than a grab–bag approach to populating the repository. Reusers frustrated with gaps in the coverage of the repository frequently fail to return to the repository. We refer the reader to the excellent collection edited by Prieto–Diaz and Arango for a deeper presentation of domain analysis [9].

However, we do have reservations concerning the exclusiveness of domain–specific repositories. Particular classes of assets are best considered domain–independent — or perhaps more aptly — useful in a broad class of domains; the most obvious asset class of this nature is that of the simple abstract data types. These "trans–domain" assets effectively form their own domain, which numerous, more restrictive domains draw upon for representational infrastructure. Domain analysts are thereby presented a dilemma, to replicate the trans–domain assets into the domain–specific repositories (along with the inherent maintenance headaches), or to factor the trans–domain assets into their own domain — resulting in a multi–domain environment. The work presented here attempts to resolve this dilemma.

2.2. Faceted Classification

Faceted classification begins by using domain analysis to identify and examine a collection of work perceived to be related [12]. This process relies on a library notion known as literary

warrant, where a classifier collects a representative sample of titles which are to be classified, and extracts descriptive terms to serve as a grouping mechanism for the titles. From this process, the classifier not only derives terms for grouping but also identifies a vocabulary that serves as values within the groups. A facet then is the encapsulation of a set of related concepts, expressed in the vocabulary of the domain.

From the software perspective, the groupings or facets become a taxonomy for the software. Using Literary Warrant, Prieto–Diaz and Freeman identified six facets that can be used as a taxonomy [10]: Function, Object, Medium, System Type, Functional Area and Setting. Every software component is classified by assigning a value for each facet for that component. For example, a software component in a Relational Database Management System that parses expressions might be classified with the tuple

(parse, expression, stack, interpreter, DBMS,).

Thus, the Function facet value for this component is "parse", the Object facet value is "expression", etc. Note that no value has been assigned for the Setting facet as this software component does not seem to have an appropriate value for the Setting facet. The taxonomy formed is "flat" in that there is no nesting of facets within facets, as is the case with other popular classification schemes (e.g., the Dewey decimal system, the ACM Computing Reviews system, etc.).

2.3. Lattices

Our principle concept for structuring the repository is a lattice. Lattices handily support instances that are pairwise incomparable (e.g., a tuple characterizing a design document and a tuple characterizing a conference paper), but that are both comparable to some third instance (e.g., the more general notion of a document, which is an upper bound in lattice terminology). The remainder of this section provides a brief review of lattice theory, section 3 presents the application of lattices to faceted classification.

2.4. Subtypes and Inheritance

The object classes in an object–oriented system are organized into a partial ordering. Object classes (*subtypes*) *inherit* attributes and methods from their ancestors (*supertypes*) in the ordering. Single inheritance schemes restrict a given object class to at most one immediate ancestor in the partial ordering. Multiple inheritance schemes allow a given object class to have any number of immediate ancestors in the partial ordering. Cardelli formalized some of the semantics of multiple inheritance in [2].

Conformance allows one type instance to be treated as if it were an instance of another type [8]. Any type *a conforms* to any type *b* if the subtype relation holds between *a* and *b*, i.e., $a \preceq b$. In a limited sense, this is what happens with inheritance, but conformance is more general. Inheritance requires that this treatment only be allowed when moving up the type hierarchy or lattice. Inheritance uses a partial ordering of types (by subtype), plus an implicit definition of existence dependencies between a given type and its ancestors. Conformance can hold for arbitrary types, independent of any type ordering scheme. Such a notion is clearly superior to inheritance based upon hierarchies or lattices for type–related query languages, where intermediate results (derived from existing types, but not part of the database schema) need to be manipulated.

Our classification scheme requires the notion of subtype to be defined between instances of facet set types and between instances of record types. Let *a* be a facet set type containing m facet instances and *b* be a facet set type containing n instances. Then *a* is a *subtype* of *b*, written $a \preceq b$, if for each b_i in *b* ($1 \le i \le n$), b_i is also in *a*. Similarly, let $R = \{i_1 : t_1, \ldots, i_n : t_n\}$ be a record type containing n components and $S = \{i_1 : t'_1, \ldots, i_m : t'_m\}$ be a record type containing m components, $1 \le m \le n$ (we can reorder component entries as necessary). Then *R* is a *subtype* of *S*, written $R \preceq S$, if for each i_j, ($1 \le j \le m$), $t_j \preceq t'_j$.

3. Lattice–Based Faceted Classification

Inheritance–based systems are, in some sense, *navigational*. A user querying an object–oriented database must be aware of the inheritance structure of that specific database, just as a user

querying a network database must be aware of database structure. Because of their non–navigational characteristics, conformance–based models promise to gain prominence over inheritance–based models, just as relational models have over network models. Our approach uses conformance to identify components using their position in a type lattice. One particularly useful consequence of this choice is the ability to dynamically evolve the repository structure, adding new vertices to the lattice as analysts examine new domains.

3.1. The Type Lattice

Figure 1 shows the general structure of the reuse type lattice. At the top is $\top$, the special universal type. Any value conforms to the universal type. At the bottom is $\bot$, the void type. These two special types ensure that any two types in the lattice have both an upper bound and a lower bound. Between the universal and void types appear the upper and lower bounds for the two type constructors facet and tuple. Facet_0 characterizes the notion of the empty facet type; it contains no values, but is still a facet. Likewise, Facet characterizes the notion of the set of all possible facet values. The dotted line between them indicates that an arbitrary number of types may appear here in the lattice. For example, figure 2 shows the sublattice for facet sets for the examples in section 2.2.

The tuple sublattice has a similar structure. At the top is the empty tuple type { }, characterizing a tuple with no facets. At the bottom is tuple, the tuple type with all possible facets.

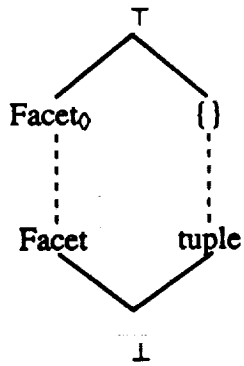


Figure 1. The reuse type lattice

Traditional retrieval of individual facet values relies upon maximal conjunction of boolean terms for retrieval of matches on *all* facets and maximal disjunction of boolean terms for matches on *any* facet of an expression. In order to fit the notion of facet into the type lattice, we look at sets of facets. A set of facets corresponds to a conjunction on all of the facets comprising the set. Each set occupies a unique position in the type lattice. We handle disjunction by allowing a given component to occupy multiple lattice positions. Matching occurs on any of the positions, providing the same semantics as disjunction.

Facet values are equivalent to enumeration values. We attach no particular connotation within the type system to a particular facet value. Values are bound to some semantic concept in the problem domain.

The subset relation is our partial order for facets. The least value of this portion of the lattice is the set of all facet values from all facets in the problem domain, denoted by the distinguished name Facet. The greatest value of this portion of the lattice is the empty set, denoted by the distinguished name $\text{Facet}_{\emptyset}$. The union operator generates the greatest lower bound. The intersection operator generates the least upper bound.

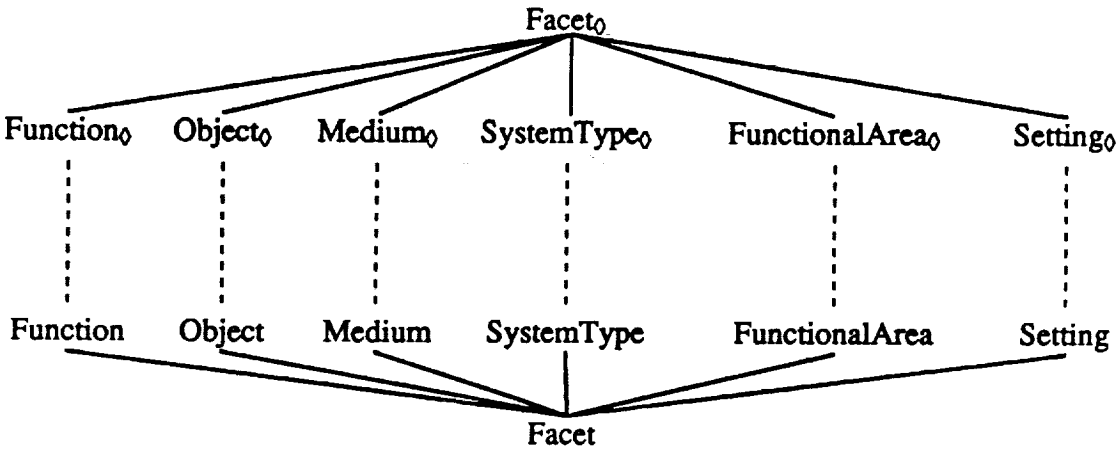


Figure 2. A Sublattice of Facet Sets

3.2. The Inference Rules

A formal mechanism for the specification of the query semantics is clearly of use. In this case, type inference directly applies to the problem. We begin with a brief remark concerning notation. In the inference rules that follow, the symbol A represents an existing set of assumptions. A always contains the type information generated by the database schema which implements the repository. It is occasionally necessary to extend the set of assumptions with some additional information. $A \,.\, x$ denotes the set of assumptions extended with the fact x. $A \vdash x$ states that given a set of assumptions A, x can be inferred. Inferences above the horizontal line act as premises for the conclusions, the inferences below the horizontal line. An expression is *well–typed* if a type for the expression can be deduced using the available inference rules, otherwise it is *ill–typed*. We give in this section only a minimal set inference rules to provide a flavor of the complete set, which may be found in [3, 4].

3.2.1. Domain Interval Subtyping

Typically, a subtype is "smaller" than its supertype, for example, the range of employee ages is a subtype of the integers. Here the reverse is true, a subtype is a larger collection of values than its supertype – some entry containing at least all the facet values of interest is thereby an instance of a subtype of the query instance's type.

A *domain interval* is a type qualification that explicitly denotes the valid subrange(s) for a base type. Rule (1) extends subtyping to domain intervals, where each subinterval in the sub-

$$\frac{\begin{array}{c} A \vdash t_{\langle m_1 \ldots n_1 \rangle} \leq t_{\langle m_1' \ldots n_1' \rangle} \\ \vdots \\ A \vdash t_{\langle m_i \ldots n_i \rangle} \leq t_{\langle m_i' \ldots n_i' \rangle} \end{array}}{A \vdash t_{\langle m_1 \ldots n_1,\ \ldots,\ m_i \ldots n_i \rangle} \leq t_{\langle m_1' \ldots n_1',\ \ldots,\ m_i' \ldots n_i' \rangle}} \qquad (1)$$

type is a subtype of some interval in the supertype. Assume that t is a base type ordered by $\leq$ (the ordering may be arbitrary). A domain that is (inclusively) delimited by two values, a and b, is denoted $t_{\langle a \ldots b \rangle}$. Intervals made up of more than a single continuous value range are denoted by a set of ranges, for example, $t_{\langle a \ldots b,\, c \ldots d,\, e \rangle}$ denotes the interval that includes the subinterval a through b inclusive, the subinterval c through d inclusive, and the singleton value e. The single-

ton range e is equivalent to $e...e$. When we use such notation we intend that $a \leq b$ and $c \leq d$, but not necessarily that $b \leq c$ or $d \leq e$. An empty pair of brackets, $t_\emptyset$, denotes an empty interval, i.e., one which contains no elements. In our particular application, the base types are finite sets of enumeration (facet) values.

3.2.2. Tuple Subtyping

This collection of inference rules explicitly types the tuples that classify components. The unlabelled record attributes used by Prieto–Diaz in tuples can be ambiguous when a given facet value is used in more than one domain. Rather than require that facet values be distinct across facets, we view a tuple r to be of type record, $\{i_1 : t_1, ..., i_n : t_n\}$. Type t_j for attribute i_j must be a facet type. The empty tuple (i.e., the tuple containing no facets) is of type $\{\}$, the tuple type with no components. The order in which components appear is arbitrary, since attribute name is used to distinguish facets.

Rule (2) characterizes record subtyping, handling situations where a component of the sub-

$$\frac{\begin{array}{c} A \vdash 1 \leq m \leq n \\ A \vdash t'_1 \preceq t_1 \\ \vdots \\ A \vdash t'_m \preceq t_m \end{array}}{A \vdash \{i_1 : t'_1, \ldots, i_m : t'_m, \ldots, i_n : t_n\} \preceq \{i_1 : t_1, \ldots, i_m : t_m\}} \qquad (2)$$

type is a subtype of the corresponding component in the supertype.

4. Modeling Multiple Domains in a Single Repository

The repository model presented in section 3 is well–suited to supporting multiple domains simultaneously, while allowing for the appearance of domain–specificity where necessary. Our model further supports the notion of a complete life cycle repository, as many of the issues applicable for component assets from multiple domains apply equally well to the characterization of life cycle assets.

4.1 Domain Analysis and Repository Structure

Consider the effect of domain analyses on the definition of the resulting repositories. If we assume that each domain analysis is carried out in isolation (in order to focus solely upon the

requirements of that particular domain), it naturally follows that the collection of facets used to characterize that domain (and the values that make up each of those facets) will also be independent. Realistically though, no domain is totally independent from all others, and there will be facets (or subsets of facet values) that two related domains will have in common.

A *maximal upper bound* for a domain is the distinguished vertex in the lattice that contains exactly those facets used in classifying the domain, but that contains no facet values. A *maximal lower bound* for a domain is that distinguished vertex in the lattice that contains exactly those facets used in classifying the domain, and for each of those facets, the n–tuple contains all values used by that facet. All instances in the domain fall somewhere between the maximal lower bound and the maximal upper bound for that domain. There are three possible relationships between domains in the unified lattice.

First, domains that share one or more complete facets, but differ by at least one facet, have facet n–tuples that are siblings in the lattice. Their only commonality is the n–tuple corresponding to the least upper bound of the two n–tuples involved; i.e., neither is a subtype of the other, but they do share a common supertype. By inference rule (2), this is the n–tuple comprised exactly of those facets which the two domains share. Domain interval subtyping does not come into play, since all facet instances contain all values in their respective facets.

Next, domains that share the same set of facets, but only partially share facet values for one or more facets, and differ by at least one facet value in some facet, are likewise siblings in the lattice. They share a single maximal upper bound, since they are classified by the same facets, and they have a greatest lower bound that is comprised of the union of each of the respective facet value sets.

Finally, domains that share some, but not all, facets, but only partially share facet values for one or more facets, are likewise siblings in the lattice. Both this and the second relationship be-

tween domains require inference rule (2), plus the entire set of inference rules for domain interval subtyping.

4.2 Sublattices as Repository Views

Reusers wishing to focus on a specific domain in our model need only concentrate on the sublattice defined by the maximal upper and lower bounds for that domain. Restricting queries to mentioning only those facets present in those n–tuples effectively reduces the repository data model to a flat tuple space in the tradition of Prieto–Diaz. The restriction is easily accomplished by providing repository views similar in nature to the relational definition of a view.

A repository view is defined by a pair of n–tuples: the first characterizing the upper extent of the lattice that the view may reference, and the second characterizing the lower extent of the lattice that the view may reference. By varying the placement of these view extents in the lattice, a variety of repository structures may be presented to the reuser. The upper extent specifies those facets which the user query must specify, and the lower extent specifies those facets which the user query may specify. Defining multiple repository views supports the presentation of arbitrary domains in a single composite view.

The most general example of this is an upper extent of { } and a lower extent of tuple opens the entire repository to the reuser.

An upper extent of the maximal upper bound for a domain and a lower extent of the maximal lower bound for that same domain restricts the reuser to specifying at most and at least those facets used in classifying that particular domain, i.e., a flat tuple space with a slight variation (sets of facet values may be specified, but need not be).

An upper extent comprised of two empty facets and a lower extent of tuple supports the notion of a multiple inheritance structure rooted at those two facets and including any vertex that includes at least those facets.

Specifying a lower extent with a facet containing only a subset of the complete facet restricts reusers employing that view from accessing any asset not classified using values from that subset.

4.3 Repository Synergy

As mentioned previously, few domains are truly independent from all others. A domain–specific repository with good coverage of that domain must necessarily duplicate at some level assets that are very similar to, if not duplicates of, assets found in repositories for closely related domains. Repositories supporting a collection of related domains avoid this unneeded replication of assets.

Many of the assets comprising these repositories will be adaptable to a variety of domains beyond the one for which they were initially designed. This synergy of assets promises a deeper understanding of the software process, but an understanding more difficult to achieve with the artificial boundaries of domains impeding access. Presenting a seamless integration of a diverse universe of assets is critical to the success of software reuse.

If the user interface for the reuse system supports the possibility of multiple repository back–ends, each specific to a given domain, it is possible to avoid asset replication. However, this implies cooperation between repository administrators that may not be convenient, or even feasible. In a mature reuse industry, repositories will be geographically distributed and span work groups, organizations, and even industries. Here again, seamless integration of multiple repositories is important, and not readily handled by a flat, static classification structure.

4.4 The Relationship to Life Cycle Assets and Granularity

As we previously mentioned, we are interested in a complete life cycle repository model, including requirements assets, design assets, and so on, as well as the traditional component assets. Granularity issues are particularly interesting in such a model, as reusers attempt to track particular concepts through requirements and design and on into maintenance.

Such a data model adds facets particular to a specific life cycle phase, or particular to a specific level of granularity, just as independent domain analysis adds facets to a particular domain. In effect, the resulting repository model contains three dimensions: domain, life cycle phase, and granularity. The definition of facet values and the corresponding set of lattice vertices handles domains and life cycle phases. Multiple vertex instances handle granularity issues under our current approach.

5. Conclusions and Future Work

We described here an approach unifying the specificity of domain–specific repositories with the flexibility of domain–independent repositories. The primary drawback we see in Prieto–Diaz' approach to classification is the flatness and homogeneity of the classification structure. A general reuse system might have not only reusable components, but also design documents, formal specifications, and perhaps vendor production information, to name a few possibilities, and have all of these things for multiple problem domains. Prieto–Diaz' scheme creates a single tuple space for all entries, resulting in numerous facets, tuples with many "not/applicable" entries for those facets, and frequent wildcarding in user queries. Our model supports precise characterization of assets, and lattice–based queries may be as restrictive or as broad as necessary to suit a reuser's needs.

Conceptual closeness is a very appealing concept in our framework, but offers its own collection of difficulties, particularly the establishment of distances for terms in a given domain, and the resolution of conflicting distances for terms occurring in multiple domains. We are currently exploring the use of neural networks to support adaptive distances, based upon user estimations of the relevance of query matches to the intended semantics. An early report on this work appears in [5].

Related to conceptual closeness is the idea of *conceptual neighborhoods* around n–tuples. Conceptual closeness addresses the semantic distance between two facet values, while conceptual neighborhoods address the semantic distance between two n–tuples in the lattice. The re-

pository model described here is one mechanism for constructing a conceptual neighborhood, based upon subtype relationships. We plan to consider alternative neighborhood definition mechanisms, including composing distances for n–tuples from the distances for facet values involved in those n–tuples. We are also considering the inclusion of signatures [7] and semantics [6, 11] into the repository model to improve query effectiveness.

References

[1] Arango, G. and R. Prieto–Diaz, "Part 1: Introduction and Overview – Domain Analysis Concepts and Research Directions," *Domain Analysis and Software Systems Modeling*, Prieto–Diaz, R. and G. Arango (eds.), IEEE Computer Society, Los Alamitos, CA, 1991, pages 9–32.

[2] Cardelli, L., "A Semantics of Multiple Inheritance," in *Semantics of Data Types (Proceedings International Symposium Sophia–Antipolos, France, June 1984)*, Springer–Verlag, Lecture Notes in Computer Science, vol. 173, pages 51–68.

[3] Eichmann, D., *Polymorphic Extensions to the Relational Model*, Ph.D. dissertation, The University of Iowa, Iowa City, IA, August 1989. Also available as technical report 89–05.

[4] Eichmann, D. A. and J. Atkins, "Design of a Lattice–Based Faceted Classification System," *Second International Conference on Software Engineering and Knowledge Engineering*, Skokie, IL, June 21–23, 1990, pages 90–97.

[5] Eichmann, D. A. and K. Srinivas, "Neural Network–Based Retrieval from Reuse Repositories," *CHI'91 Workshop on Pattern Recognition and Neural Networks in Human–Computer Interaction*, New Orleans, LA, April 28, 1991.

[6] Eichmann, D. A., "Selecting Reusable Components Using Algebraic Specifications," *Second International Conference on Algebraic Methodology and Software Technology*. Iowa City, IA, May 22–25, 1991, pages 37–40.

[7] Eichmann, D. A., "A Hybrid Approach to Software Repository Retrieval: Blending Faceted Classification and Type Signatures," *Third International Conference on Software Engineering and Knowledge Engineering*, Skokie, IL, June 27–29, 1991, pages 236–240.

[8] Horn, C., "Conformance, Genericity, Inheritance and Enhancement," *ECOOP'87 – Proc. European Conference on Object–Oriented Programming*, Paris, France, June 15–17, 1987, pages 223–233.

[9] Prieto–Diaz, R. and G. Arango (eds.), *Domain Analysis and Software Systems Modeling*, IEEE Computer Society, Los Alamitos, CA, 1991.

[10] Prieto–Diaz, R. and P. Freeman, "Classifying Software for Reusability," *IEEE Software*, vol. 4, no. 1, January, 1987, pages 6–16.

[11] Steigerwald, R., Luqi, and J. McDowell, "A CASE Tool for Reusable Software Component Storage and Retrieval in Rapid Prototyping," *Third International Conference on Software Engineering and Knowledge Engineering*, Skokie, IL, June 27–29, 1991, pages 34–39.

[12] Vickery, B. C., *Faceted Classification Schemes*, vol. 5, Rutgers Series on Systems for the Intellectual Organization of Information, S. Artandi (ed.), Rutgers University Press, New Brunswick, NJ, 1966.

Assessing Repository Technology: Where Do We Go From Here?*

David Eichmann†

Software Reuse Repository Lab (SoRReL)
Dept. of Statistics and Computer Science
West Virginia University

Send correspondence to:
David Eichmann
SoRReL
Dept. of Statistics and Computer Science
West Virginia University
Morgantown, WV 26506
email: eichmann@cs.wvu.wvnet.edu

* to appear in the *International Journal of Software Engineering and Knowledge Engineering.*
† This work was supported in part by NASA as part of the Repository Based Software Engineering project, cooperative agreement NCC-9-16, project no. RICIS SE.43, subcontract no. 089 and in part by a grant from MountainNet Inc.

Abstract

Three sample information retrieval systems, archie, autoLib, and WAIS, are compared as to their expressiveness and usefulness, first in the general context of information retrieval, and then as prospective software reuse repositories. While the representational capabilities of these systems are limited, they provide a useful foundation for future repository efforts, particularly from the perspective of repository distribution and coherent user interface design.

1 – Introduction

As information becomes an increasingly important sector of the global economy, the way in which we access that information – and thereby the way in which we access and structure *knowledge* – becomes a critical concern. The engineering of knowledge is quickly becoming an area of research in its own right, independent of its parent disciplines of artificial intelligence, database systems, and information retrieval; consider the title of the journal that you now hold in your hands. Wegner recognized the value of knowledge engineering in his landmark article on the role of capital in software development:

> "Knowledge engineering is a body of techniques for managing the complexity of knowledge... it is capital-intensive in the sense that reusability is a primary consideration in the development of books, expert systems, and other structures for the management and use of knowledge." [10, p. 33]

Just as Wegner observed that the products of software engineering are capital, so are the products of knowledge engineering a form of capital. Identification, structure, and locatability are critical to the enabling of this knowledge capital. Innovation in this area is driven from two diverse perspectives, the traditional perspective of researchers and a not-so-traditional perspective of what might be referred to as an information underground.

The goal of this information underground is not necessarily an extension of the state of the art, but a rather more pragmatic development of an informational infrastructure [4]. The prototypes resulting from this type of work propagate quickly over the Internet, immediately generating large numbers of users. Even while still experimental, systems that provide distinct benefit frequently need to limit access in order to maintain reasonable system performance for other users of the underlying platforms.

My reference to this community as an underground is calculated, for even within the computer science community (let alone the academic or commercial communities as a whole), only a small percentage of individuals are aware of such information systems. This article was spurred by my interest in software repositories, a number of conversations that I've had in recent months, and the

benefit I think can be gained by widening the forum for such systems to a larger audience.

In particular, it is interesting to evaluate these systems as an enabling technology for software reuse repositories. Repositories, and by implication, information retrieval mechanisms, play a critical role in successful reuse. This statement disagrees with the conventional wisdom [9], that reuse is a social and managerial issue, and not a technical one. A closer examination of the conventional wisdom leads to a recognition that without a repository with substantial representational capability many of the social and managerial requirements cannot be supported.

This paper surveys a number of interesting information server projects, with an eye towards enabling technologies. Section 2 lays down a typical scenario in which such systems are used. Sample sessions for three systems appear in section 3, and an analysis appears in section 4. I conclude with remarks on the potential of future systems.

2 – A Scenario and User Profile

Consider a programmer involved in a research project in some reasonably sized university. I choose this context not only for its personal familiarity, but also because

- such projects typically take place in facilities with rich local and wide area network connectivity;
- programmers typically have a personal workstation with substantial display capabilities (e.g., X-Windows); and
- there are strong incentives in avoiding the redevelopment of capabilities available from other projects, either local or remote.

In effect, the development environment is one which is typical, or will be within the next few years. In addition, the social infrastructure and equipment infrastructure for a successful reuse program are present, if not an explicit charter for reuse, or a true repository.

Our programmer is now faced with a dilemma — aware that there is a strong likelihood that a

needed tool or component already exists somewhere out on the network, but uncertain as to where to begin the search in the thousands of systems that currently make up the Internet, or even how to identify the needed artifact. Until recently the only choices included asking acquaintances for advice (although the study by Schwartz and Wood [7] demonstrated the amazing potential for even ad hoc mechanisms such as this), poring over intermittently posted electronic digest news articles for likely sounding names, or manually searching a few sites maintained by volunteers and accessible through anonymous ftp. Obviously, our programmer is ripe for recruitment as a client of the services provided by the information underground.

3 – Example Repositories

Early in the evolution of the Internet, system administrators began adapting file transfer facilities into what today is referred to as anonymous ftp, comprised of publicly accessible accounts, a limited file space, and a restricted command set. These facilities, while amazingly popular as a dissemination tool, presume a fair amount of user knowledge, not the least of which being where to look for the sought-after artifact. This section describes three information systems, archie, WAIS, and autoLib. Each of these systems has a distinct design focus, anonymous ftp access in archie, document retrieval/display in WAIS, and a limited form of electronic library in autoLib. However, the resulting systems have much in common, and their look and feel has several similarities. These systems were selected for discussion because they were designed primarily as *information retrieval systems*, rather than as software repository systems.

3.1 – archie

The archie system is "an on-line resource directory service for an internetworked environment" [3]. While archie isn't truly a repository per se, since it doesn't actually contain the artifacts that it classifies, when treated as a whole with the diverse anonymous ftp sites that it references, it does fit into our discussion. Archie grew out of the efforts of Emtage and Deutsch to automate the creation and referencing of previously hand-maintained lists of anonymous ftp sites. A demon peri-

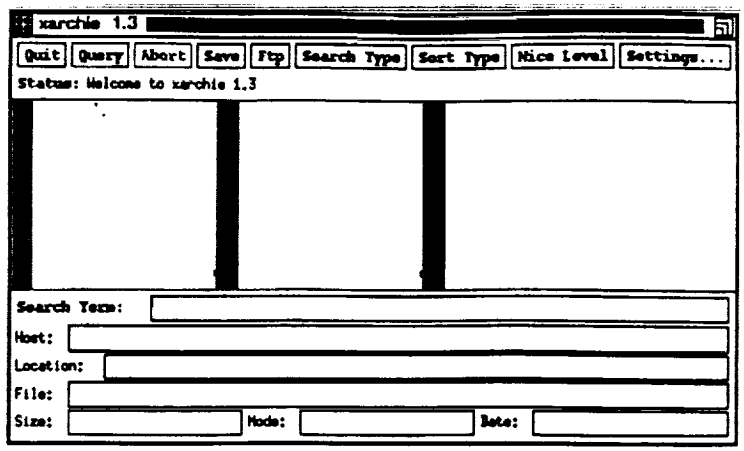


Figure 1: archie screen upon entry

odically sweeps through a list of known ftp sites, creating a list of artifacts accessible at each of them. This list of artifacts is then indexed for access by clients throughout the Internet seeking a site for some particular item.

I describe the xarchie user interface here, developed by Ferguson for the X-windows system from the ASCII user interface developed by Kehoe and the Prospero system developed by Neuman [5]. Xarchie and archie together form an example of a client/server application architecture, where the client application (xarchie) provides user-local support for commands, information display, and communication to the server application (archie), which provides access to a remove facility, in this case the archie database. Figure 1 shows xarchie's screen at entry. The series of buttons across the top of the window control the activity of the user's xarchie client and its interaction with an archie server and the ftp sites which the server indexes. Figure 2 shows the xarchie settings panel, including in particular the mode of search (exact, substring, regular expression, etc.), the order that hits are presented (sorted by name, modification date, etc.), and the archie server host to interrogate, in this case archie.sura.net.

Entering a search term for an artifact, say *xarchie.tar.Z*, a compressed Unix tar file of the xarchie source directory, and clicking the query button initiates the search, as shown in figure 3. As the search progresses, xarchie updates the status line, indicating establishment of connection, progress, and completion.

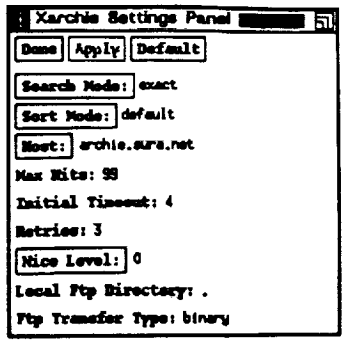


Figure 2: archie settings window

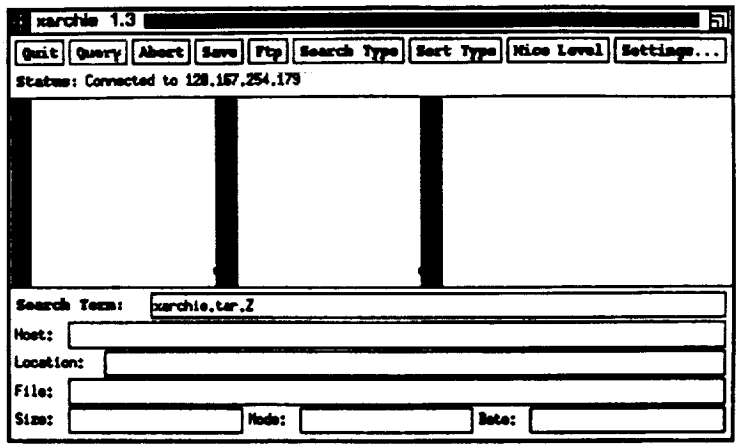


Figure 3: Initiating an archie search

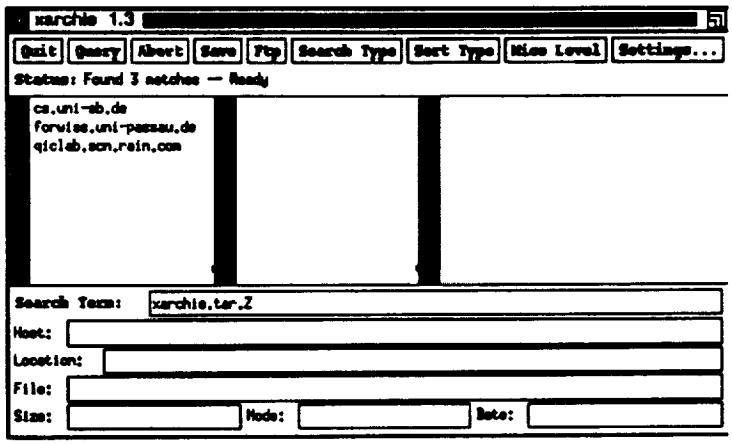


Figure 4: archie search results

Figure 4 shows the results of the search as a list of sites in the left scrolling region in the middle of the window. Selecting a particular site by clicking on it results in figure 5, with the location, size, and so on for this artifact on this site. A single instance of a match at the selected site automatically selects the middle scrolling region (corresponding to the directories) and the right scrolling region

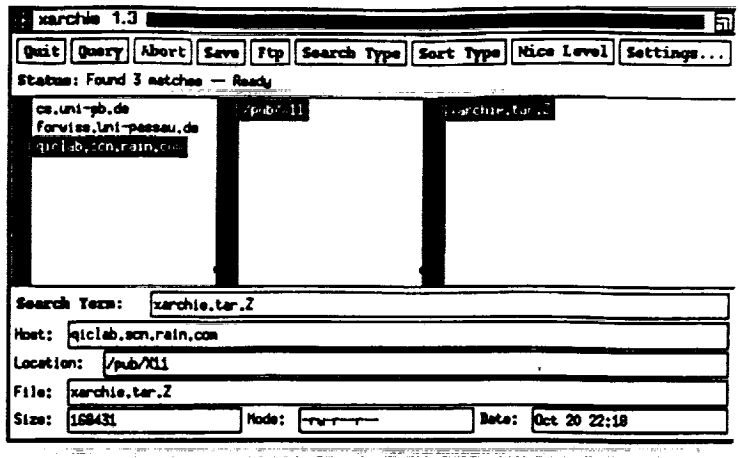


Figure 5: Selection of a site and copy

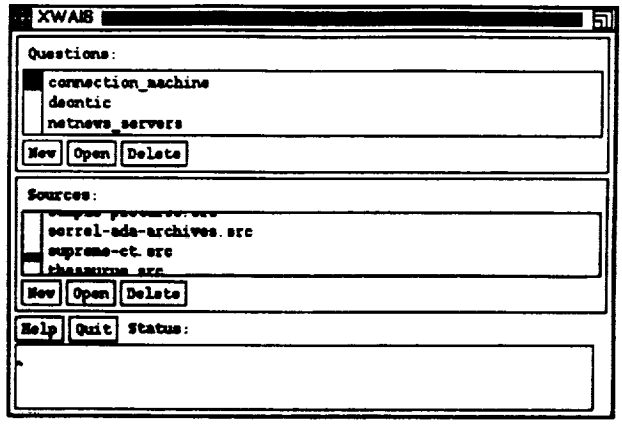


Figure 6: WAIS main window

(corresponding to the files). Multiple matches (typical with inexact matches) require the selection of both a directory and a file for the lower fields to be filled in. Clicking the ftp button establishes an anonymous ftp session to the archive site and retrieves a selected artifact into the local directory shown in the settings panel (shown in figure 2 as '.', the current directory).

3.2 – WAIS

The Wide Area Information Service (WAIS) is an experiment in text-based distributed information systems by Thinking Machines and a number of collaborators [4]. WAIS supports the notion of multiple sources of information; a user selects one or more sources to respond to a question, phrased as a string of words which are deemed relevant to the question. Figure 6 shows the main window, containing a list of previously phrased questions and a list of already known sources.

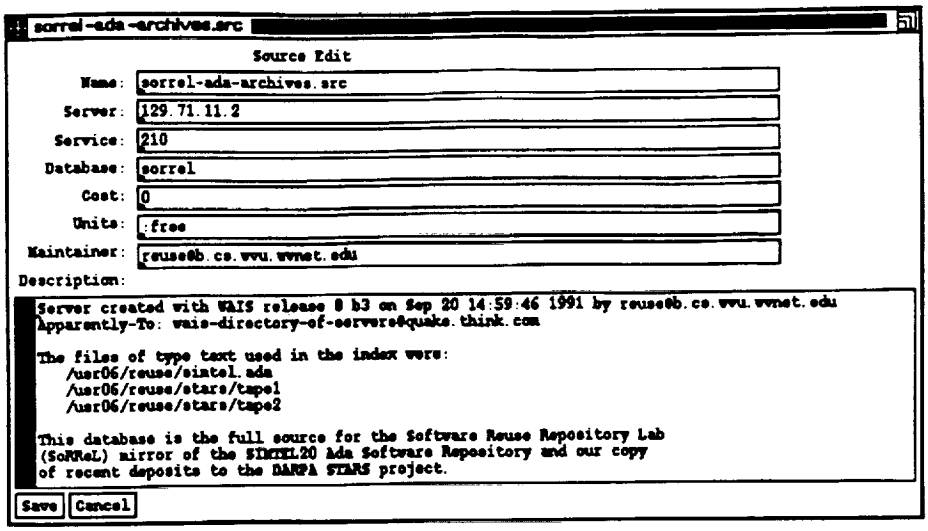


Figure 7: Source window for SoRReL archive

Opening a source displays a window containing information concerning the nature and location of that source, as shown in figure 7 for the Ada archive that the SoRReL group maintains. This information includes the Internet address and service port that the server for the source listens to, as well as unit and cost fields (as yet unused) and a textual description of the source. A single server can support multiple sources, each separately indexed and independently accessible. A distinguished source, maintained by Thinking Machines, acts as a directory to other sources by indexing source definitions such as the one shown in figure 7. These source definitions are retrievable using the same question mechanism employed for other questions. The sole distinction is in the saving of results; saving a source definition places it in the directory containing the user's known sources, making it accessible for subsequent questioning.

Figure 8 shows the question window following a successful search of the SoRReL source. Users select one or more already known sources to be consulted for this question by clicking the add source button and selecting from the resulting display of sources. The "Tell me about:" field accepts a collection of words to be used as a specification of the question. WAIS uses relevance feedback as its search mechanism; documents which match one or more of the words contained in the "Tell me about:" field are added to the collection of matching documents, and then presented to the user in the "Resulting Documents:" field ranked by a relevance metric, an indication of the fit to

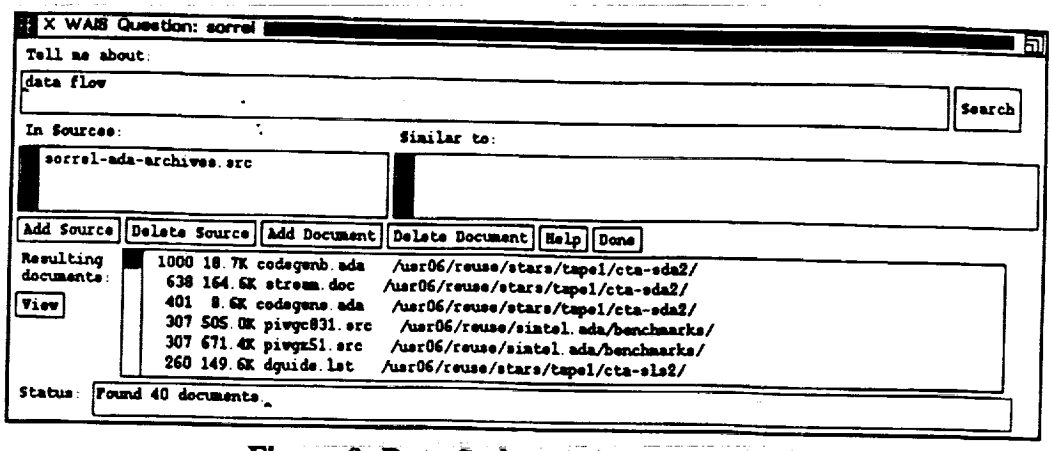


Figure 8: Data flow question and results

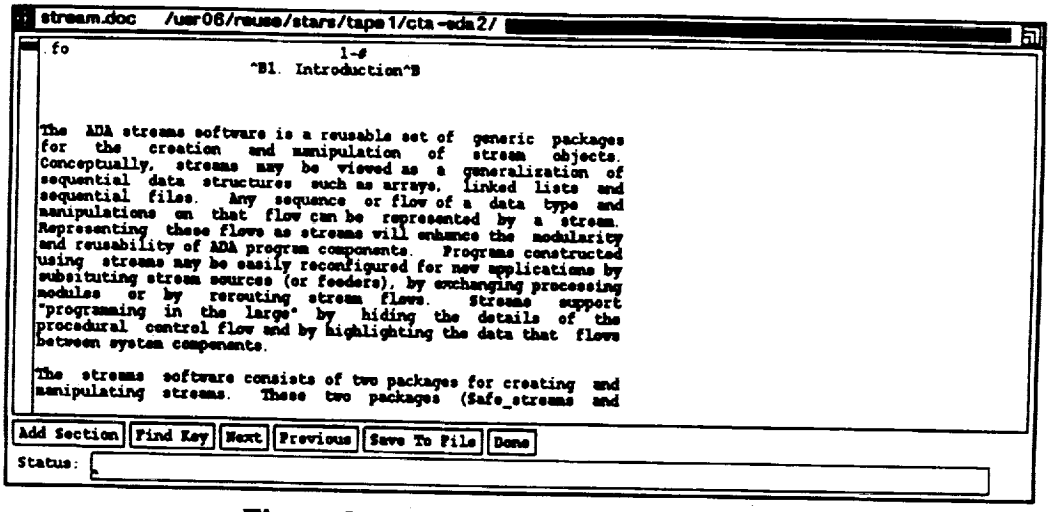


Figure 9: Viewing the streams document

the words occurring in the query string. Relevance feedback has been shown to be more effective than boolean expression as a search mechanism for textual information (a report of one such study appears in [6]).

Selecting a result document for viewing retrieves the document from its server and displays it in a window such as that shown in figure 9, which contains a portion of a document describing an Ada implementation of a stream package. The find key button scrolls the window and highlights in turn each occurrence of search words in the document. WAIS lets users specify an arbitrary program on the user's machine as the viewer to be invoked for a given class of documents, with the class defined using the suffix of the document's file name (for example, xgif is typically used to display images whose names end in '.gif').

Iterative refinement of a search that results in documents viewed with the text viewer is accomplished by selecting a salient portion of the document and clicking the add section button. An indication of the text selected is added to the "Similar to" field in the question window. Subsequent searches then append these refinements to the primary search phrase.

3.3 – autoLib

The autoLib system, under development by Barrios Technology and NASA's Johnson Space Center, is a monolithic application supported by a commercial relational database system (comprising the meta-information) and a UNIX file system (comprising the objects themselves). The structure of information provided by archie and WAIS is flat in the sense that there is little structure provided other that an indexing mechanism. The autoLib system, on the other hand, supports both a flexible single inheritance mechanism for definition of meta-information, and the definition of heterogenous collections of objects drawn from the inheritance scheme [1]. Figure 10 shows the main window for autoLib, including the topmost collection and its immediate sub-collections.

Clicking on an entry in the list moves the user down the hierarchy of collections to the corresponding subcollection, and that collection's subcollections are then displayed. The three buttons at the bottom of the window allow the user to step back up one level in the collection hierarchy, to move directly to the top of the hierarchy, or to view the objects associated with the current collection, respectively.

Figure 11 shows the object browser window, displaying the contents of one such collection. The three columns of information include the object's identifier, its filename, and a .short title.

The object viewer window for object 2446 appears in figure 12. AutoLib employs a commercial relational database package for information storage, but the user model for autoLib is object-oriented, defined not only as a hierarchy of class definitions, where superclass names are prefixes of subclass names, but also as a hierarchy of collections, as mentioned earlier. AutoLib maps each

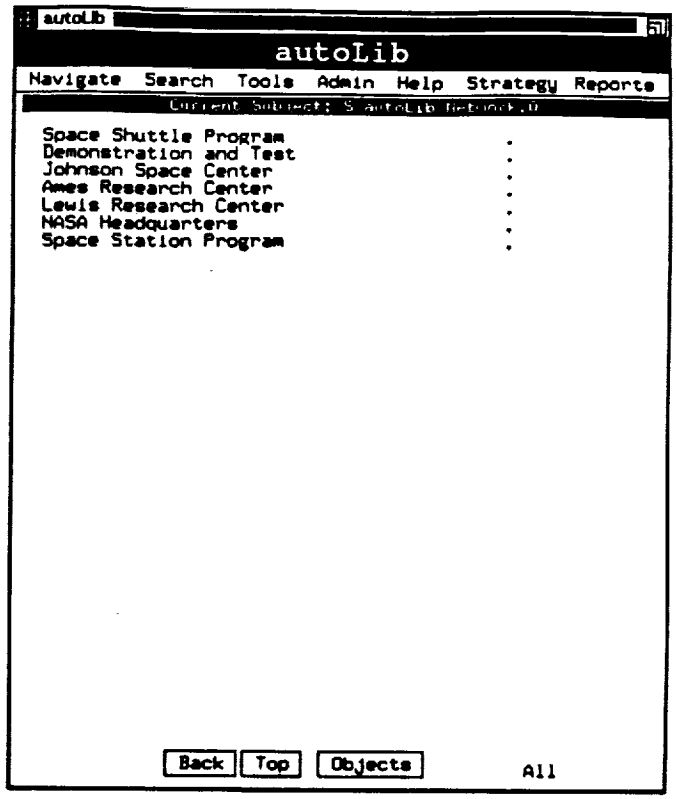


Figure 10: autoLib main window

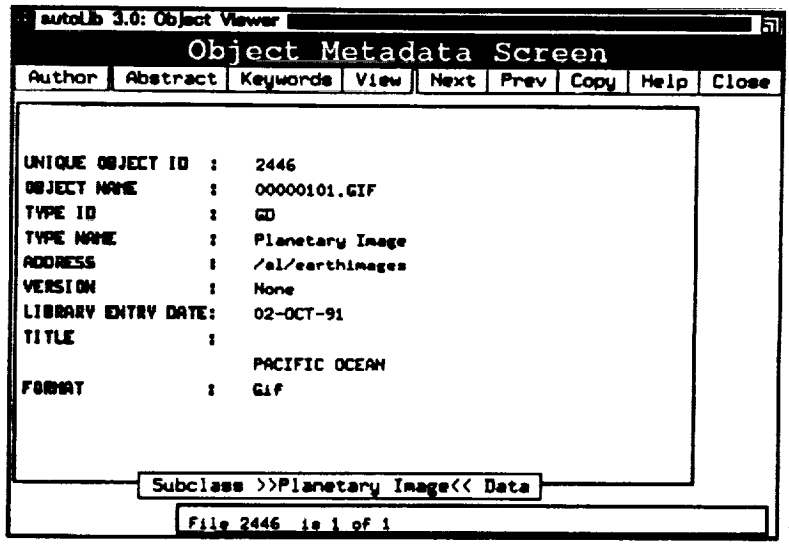


Figure 12: autoLib object viewer

concept (collection, class, object, etc.) into a corresponding database relation and maps each field in an autoLib window (e.g., the object filename, 00000101.GIF, for object 2446) to an attribute in the corresponding relation. The system derives the interpretation for a given object in the generic object relation from the field definitions stored by autoLib in the class field relation. While this is

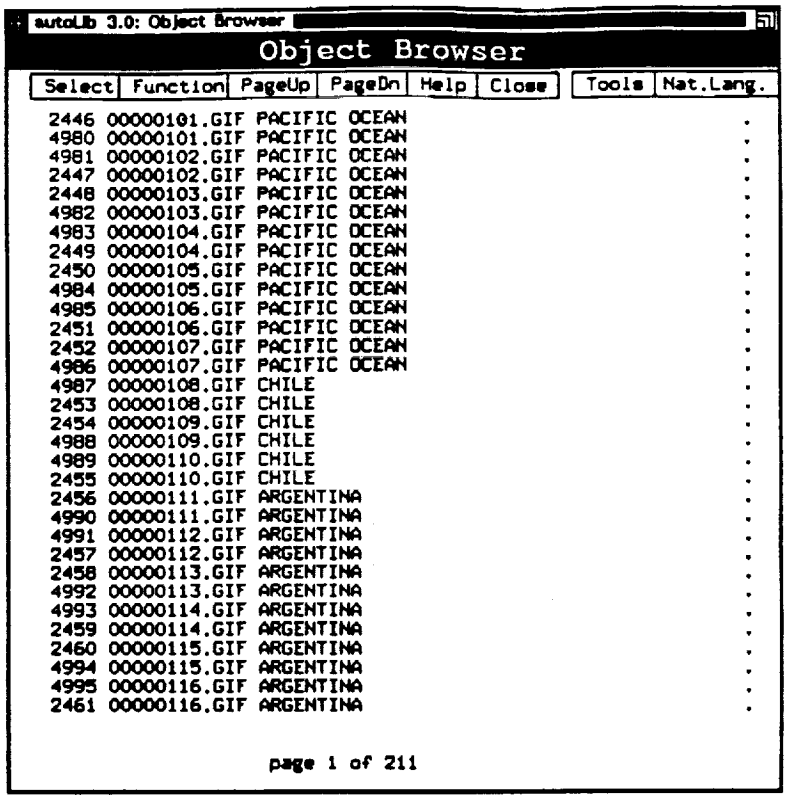


Figure 11: autoLib object browser

not a true object-oriented database, it provides much of the flexibility and rich structural mechanisms of a object-oriented database. The integration of objects and relations has been carried much further in work on extensible database systems such as POSTGRES [8].

In addition to the collection browsing mechanism described here, autoLib supports traditional boolean expression retrieval and a form of relevance feedback. Each object class has associated with it a *tool*, which is used to view the object itself, as opposed to the metadata characterizing that object, i.e., the fields presented in the object view window. Unlike WAIS, where tool execution occurred on the user workstation, tool execution in autoLib occurs on the autoLib server – the user workstation merely acts as an X-windows display.

4 – A Brief Comparison

Viewing these three systems as potential software repositories is interesting, and at the same time somewhat unfair to their designers, as none were created with that purpose in mind. However,

systems such as these are frequently called into service in such contexts, and the flexibility and adaptability exhibited provides interesting concepts and features for inclusion into systems specifically intended as repositories. Table 1 summarizes major aspects of the three systems. The popu-

Table 1: Systems Synopsis

	archie	WAIS	autoLib
architecture	client/server	client/server	monolithic
# server sites	~10	~100	1
interfaces	X-Windows, ASCII	X-Windows, ASCII	X-Windows (ASCII under development)
retrieval mechanisms	pattern-matching (on name only)	relevance feedback	browsing, boolean expression, relevance feedback (on abstract only, not full text)
information domain	material available by anonymous ftp	textual information	NASA flight center library materials
information stored	name, location, file attributes	word occurrence, headline, full text	full text / image, index terms, meta-information (administrator-defined)
archiving responsibility	decentralized	decentralized	centralized
indexing responsibility	centralized	decentralized	centralized
support required (archive)	none	moderate	high
support required (indexing)	moderate	low	high
promise as a repository	poor	limited	a potential framework
availability	public	public	private

larity of archie stems not from its rich representation scheme or novel search mechanisms, but rather from the low levels of effort required on the part of archive administrators and users to employ the system. It is an excellent example of how a limited purpose system implemented by volunteers can provide a valuable resource. Referring to archie as a software repository, however, stretches the definition of repository perhaps a little too far. Consideration of an artifact at a site as a candidate component requires that the user knows both the name and the purpose of that artifact, and the retrieval of the complete artifact (irrespective of the total size) before further consideration can be made.

The display facilities of WAIS alleviate the limitations of archie by presenting the user with a flexible means of query specification (without requiring classification by the archivist) and the opportunity to select from a variety of candidates and view portions of them prior to retrieving the complete text of the final selection. WAIS further increases flexibility in the nature of repositories by supporting interrogation of multiple sources for a given query and the generation of both public and private sources. (Note, however, that there is no technical impediment to doing this with archie as well – the archie designers simply chose global indexing rather than regional or local indexing.) The principle virtue of WAIS, its treatment of all material as text to be indexed, is also its principle failing from our perspective – there is no discrimination between code, supporting documents, and so forth – resulting in slightly more cumbersome search behavior.

The use of an administrator-defined set of collection and class definitions provides autoLib a great deal of flexibility in organizing the information. In addition to the ability to organize the global structure of the information base, this definitional facility supports meta-descriptions of artifacts, a useful feature in our chosen context.

The structuring, classification, and retrieval mechanisms of autoLib are by far the richest of the three systems compared here. Much of this power obviously stems from the fact that autoLib is a proprietary system, whereas archie is a volunteer effort and WAIS is a research project. However, autoLib's look and feel suffers dramatically in our sample context. Unlike archie and WAIS, which use a client/server paradigm, autoLib executes solely on the server platform. In wide-area domains like the one in which our programmer operates, this results in slow display and update of windows, and an inability for a user to select alternative viewing tools without the intervention of the repository administrator.

5 – Conclusions

This paper reviewed three example information retrieval systems currently in use by a broad diversity of users. I focussed on computer-supported repositories for software artifacts (i.e., com-

ponents, documents, test suites, executable images, etc.) rather than addressing the more broadly-scoped notion of an information repository, which could easily encompass entities such as public libraries.

While these systems were not explicitly designed as software repositories, they do each provide some aspect of repository requirements. Each is a legitimate step forward in utility from early techniques for wide distribution of software. This analysis leads to the following proposal for perceiving the current state of software repository efforts from the standpoint of information systems.

Generation 1 – Program Libraries

This includes not only traditional compiler libraries, but also more distributed mechanisms such as the Ada Software Repository [2] and the various archives for news groups such as comp.sources.unix.

Generation 2 – Information Servers

Examples of this generation include archie, autoLib, and WAIS. The emphasis here is on the indexing and retrieval mechanisms, rather than upon deep representation.

Generation 3 – Component Bases

Fine-grain characterization of components and interrelationships distinguishes this generation. The nature of reuse in this generation is compositional, and is typified by the Department of Defense STARs efforts and the Japanese Software Factory projects.

Generation 4 – Software Knowledge Bases

This generation provides deep knowledge about representation, generation, and composition of components and design schemes and the process of software development.

My separation criteria for repository generations involves the nature and accessibility of the knowledge of each artifact that comprises the repository. Generations one and two provide wide access to artifacts, but little supporting infrastructure (although it might be argued that autoLib

could through the proper configuration efforts of a repository administrator be turned into a rudimentary generation 3 system). Generations three and four provide increasingly rich information concerning the nature of the artifacts contained within them. However, with this richness comes increasing specialization of domain, and increasing difficulty in supporting interoperability between repositories. The component base services of today and the software knowledge base services of tomorrow should not loose sight of the design goals of today's successful information servers.

References

1. Barrios Technology, *autoLib Automated Online Library Version 3 Product Overview*, March 1990.

2. R. Conn, "The Ada Software Repository and Software Reusability," *Proceedings of the Fifth Annual Joint Conference on Ada Technology and Washington Ada Symposium*, 1987, 45-53. (Also appears in *Tutorial: Software Reuse: Emerging Technology*, W. Tracz (ed.), IEEE Press, 1988, 238-246.)

3. A. Emtage and P. Deutsch, "archie – An Electronic Directory Service for the Internet," *Proceedings of USENIX*, San Francisco, CA, January 1992, 93-110.

4. B. Kahle, *Wide Area Information Server Concepts*, Thinking Machines Inc., November 1989.

5. C. Neuman, *The Virtual System Model for Large Distributed Operating Systems*, The University of Washington, 1989.

6. S. E. Robertson and K. Sparck Jones, "Relevance Weighting of Search Terms," *Journal of the American Society for Information Science*, **27** (1976), 129-146.

7. M. F. Schwartz and D. C. M. Wood, "A Measurement Study of Organizational Properties in the Global Electronic Mail Community," Technical Report CU-CS-482-90, University of Colorado, Boulder, August 1990.

8. M. Stonebraker, L. A. Rowe, and M. Hirohama, "The Implementation of POSTGRES," *IEEE Transactions on Knowledge and Data Engineering*, **2**, 1 (1990), 125-142.

9. W. Tracz, "Software Reuse Myths," *ACM SIGSOFT Software Engineering Notes* **13**, 1(1988) 17–21.

10. P. Wegner, "Capital–Intensive Software Technology," *IEEE Software* **1**, 3 (1984) 7–45.

A Neural Net-Based Approach to Software Metrics*

G. Boetticher, K. Srinivas, D. Eichmann†

Software Reuse Repository Lab
Department of Statistics and Computer Science
West Virginia University
{gdb, srini, eichmann}@cs.wvu.wvnet.edu

Correspondence to:

David Eichmann
SoRReL Group
Department of Statistics and Computer Science
West Virginia University
Morgantown, WV 26506

email: eichmann@cs.wvu.wvnet.edu
fax: (304) 293-2272

* Submitted to the 4th International Conference on Tools With Artificial Intelligence, November 10-13, 1992, Arlington, Virginia.

† This work was supported in part by NASA as part of the Repository Based Software Engineering project, cooperative agreement NCC-9-16, project no. RICIS SE.43, subcontract no. 089.

Abstract

Software metrics provide an effective method for characterizing software. Metrics have traditionally been composed through the definition of an equation. This approach is limited by the fact that all the interrelationships among all the parameters be fully understood. This paper explores an alternative, neural network approach to modeling metrics. Experiments performed on two widely accepted metrics, McCabe and Halstead, indicate that the approach is sound, thus serving as the groundwork for further exploration into the analysis and design of software metrics.

1 – Introduction

As software engineering matures into a true engineering discipline, there is an increasing need for a corresponding maturity in repeatability, assessment, and measurement — of both the processes and the artifacts associated with software. Repeatability of artifact takes natural form in the notion of software reuse, whether of code or of some other artifact resulting from a development or maintenance process.

Accurate assessment of a component's quality and reusability are critical to a successful reuse effort. Components must be easily comprehendible, easily incorporated into new systems, and behave as anticipated in those new systems. Unfortunately, no consensus currently exists on how to go about measuring a component's reusability. One reason for this is our less than complete understanding of software reuse, yet obviously it is useful to measure something that is not completely understood.

This paper describes a preliminary set of experiments to determine whether neural networks can model known software metrics. If they can, then neural networks can also serve as a tool to create new metrics. Establishing a set of measures raises questions of coverage (whether the metric covers all features), weightings of the measures, accuracy of the measures, and applicability over various application domains. The appeal of a neural approach lies in a neural network's ability to model a function without the need to have knowledge of that function, thereby providing an opportunity to provide an assessment in some form, even if it is as simple as *this* component is reusable, and *that* component is not.

We begin in section 2 by describing two of the more widely accepted software metrics and then in section 3 briefly discuss various neural network architectures and their applicability. Section 4 presents the actual experiment. We draw conclusions in section 5, and present prospects for future work in section 6.

2 – Software metrics

There are currently many different metrics for assessing software. Metrics may focus on lines of code, complexity [7, 8], volume[5], or cohesion [2, 3] to name a few. Among the many metrics (and their variants) that exist, the McCabe and Halstead metrics are probably the most widely recognized.

The McCabe metric measures the number of control paths through a program [7]. Also referred to as cyclomatic complexity, it is defined for a program G as [8]:

$$v(G) = \text{number of decision statements} + 1$$

assuming a single entry and exit for the program, or more generally as

$$v(G) = \text{Edges} - \text{Nodes} + 2 \cdot \text{Units}$$

where Edges, Nodes, and Units correspond respectively to the number of edges in the program flow graph, the number of nodes in the program flow graph, and the number of units (procedures and functions) in the program.

The Halstead metric measures a program's volume. There are actually several equations associated with Halstead metrics. Each of these equations is directly or indirectly derived from the following measures:

- n_1 the number of unique operators within a program (operators for this experiment include decision, math, and boolean symbols);
- N_1 the total number of operators within a program;
- n_2 the number of unique operands in a program (including procedure names, function names, variables (local and global), constants and data types); and
- N_2 the total number of operands in a program.

The measurements for a program are equal to the sum of the measurements for the individual modules.

Based on these four parameters, Halstead derived a set of equations, which include the follow-

ing (in which we are most interested):

Actual Length: $N = N_1 + N_2$

Program Volume: $V = N \cdot \log_2(n)$

Program Effort: $E = V / (2 \cdot n_2)$

Traditionally, software metrics are generated by extracting values from a program and substituting them into an equation. In certain instances, equations may be merged together using some weighted average scheme. This approach works well for simple metrics, but as our models become more sophisticated, modeling metrics with equations becomes harder. The traditional process requires the developer to completely understand the relationship among all the variables in the proposed metric. This demand on a designer's understanding of a problem limits metric sophistication (i.e., complexity). For example, one reason why it is so hard to develop reuse metrics is that no one completely understands "design for reuse" issues.

The goal then is to find alternative methods for generating software metrics. Modeling a metric using a neural network has several advantages. The developer need only to determine the endpoints (inputs and output) and can disregard (to an extent) the path taken. Unlike the traditional approach, where the developer is saddled with the burden of relating terms, a neural network automatically creates relationships among metric terms. Traditionalists might argue that you must fully understand the nuances among terms, but full understanding frequently takes a long time, particularly when there are numerous variables involved.

We establish neural networks as a method for modeling software metrics by showing that we can model two widely accepted metrics, the McCabe and the Halstead metrics.

3 – Neural Networks

Neural networks by their very nature support modeling. In particular, there are many applications of neural network algorithms in solving classification problems, even where the classification

boundaries are not clearly defined and where multiple boundaries exist and we desire the best. It seems only natural then to use a neural network in classifying software.

There were two principle criteria determining which neural network to use for this experiment. First, we needed a supervised neural network, since for this experiment the answers are known. Second, the network needed to be able to classify.

The back-propagation algorithm meets both of these criteria [9]. It works by calculating a partial first derivative of the overall error with respect to each weight. The back-propagation ends up taking infinitesimal steps down the gradient [4]. However, a major problem with the back-propagation algorithm is that it is exceedingly slow to converge [7]. Fahlman developed the quickprop algorithm as a way of using the higher-order derivatives in order to take advantage of the curvature [4]. The quickprop algorithm uses second order derivatives in a fashion similar to Newton's method. From previous experiments we found the quickprop algorithm to clearly outperform a standard back-propagation neural network.

While an argument could be made for employing other types of neural models, due to the linear nature of several metrics, we chose quickprop to ensure stability and continuity in our experiments when we moved to more complex domains in future work.

4 – Modeling Metrics with Neural Networks

As mentioned earlier, the goal of the experiment is to determine whether a neural network could be used as a tool to generate a software metric. In order to determine whether this is possible, the first step is to determine whether a neural network can model existing metrics, in this case McCabe and Halstead. These two were chosen not from a belief that they are particularly good measures, but rather because they are widely accepted, public domain programs exist to generate the metric values, and the fact that the McCabe and Halstead metrics are representative of major metric domains (complexity and volume, respectively).

Since our long term goal of the experiment is to determine whether a neural network can be used to model software reusability metrics, Ada, with its support for reuse (generics, unconstrained arrays, etc.) seemed a reasonable choice for our domain language. Furthermore, the ample supply of public domain Ada software available from repositories (e.g., [1]) provides a rich testbed from which to draw programs for analysis.

Finally, programs from several distinct application domains (e.g., abstract data types, program editors, numeric utilities, system oriented programs, etc.) were included in the test suite to ensure variety.

We ran three distinct experiments. The first experiment modeled the McCabe metric on single procedures, effectively fixing the unit variable at 1. The second experiment extended the first to the full McCabe metric, including the unit count in the input vector, and using complete packages as test data. The third experiment used the same test data in modeling the Halstead metric, but a different set of training vectors.

4.1 – Experiment A: A Neural McCabe metric for Procedures

In this experiment all vectors had a unit value of one, so the unit column was omitted. In building both the training and test sets all duplicate vectors and stub vectors (i.e., statements of the form "PROCEDURE XYZ IS SEPARATE") were removed. The input for all trials in this experiment contained 26 training vectors and 8 test vectors (the sets were disjoint). Each training vector corresponded to an Ada procedure and contained three numbers, the number of edges, the number of nodes, and the cyclomatic complexity value.

The goals of this first experiment were to establish whether a neural network can be used to model a very simple metric function (the McCabe metric on a procedure basis) and to examine the influence neural network architecture has on the results. The input ran under 6 different architectures: 2-1 (two input layers, no hidden layers, and one output layer), 2-1-1 (two input layers, one

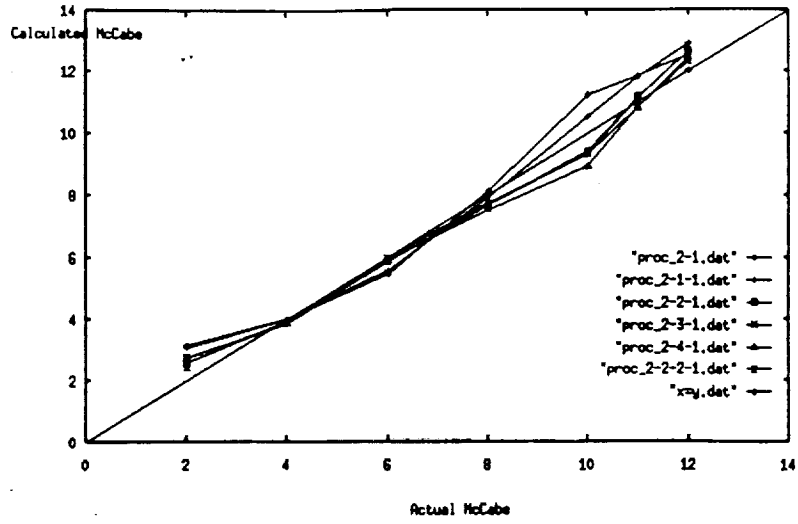


Figure 1: McCabe Results for Single Procedures

hidden layer, and one output layer), 2-2-1, 2-3-1, 2-4-1, and 2-2-2-1. In order to examine the impact of architecture, other parameters remained constant. Alpha, the learning rate, was set to 0.55 throughout the trials. An asymsigmoid squashing function (with a range of 0 to +1) was used to measure error. Finally, each trial was examined during epochs 1000, 5000, and 25,000. Figure 1 presents the results of these trials. In the graph, the neural calculated values are plotted against the actual values for the metric at 25,000 epochs*. In an ideal situation, all lines would converge to x = y, indicating an exact match between the actual McCabe metric (calculated using the traditional equation) on the x-axis, and the neural calculated McCabe metric on the y-axis.

This experiment provides good results considering the minimal architectures used. Most points tend to cluster towards the actual-calculated line regardless of architecture selection. This suggests that more complex architectures would not provide dramatic improvements in the results.

Considering that only 26 training vectors were used, the results were quite favorable, and we moved on to the next experiment.

* In fact, all figures in the paper correspond to the results following 25,000 epochs.

4.2 – Experiment B: A Neural McCabe Metric for Packages

The second experiment modeled the McCabe metrics on a package body basis. Changes in data involved the addition of another input column corresponding to the number of units (the number of procedures in an Ada package) and the selection of a slightly different set of training vectors, chosen to ensure coverage of the added input dimension.

The experiment ranged over five different architectures (3-3-1, 3-5-1, 3-10-1, 3-5-5-1, and 3+5-5-1 (hidden layers are connected to all previous layers)) and four training sets (16, 32, 48, and 64 vectors). Each smaller training set is a subset of the larger training set, and training and test sets were always disjoint. Alpha remained constant at 0.55 throughout the trials. Once again, we used an asymsigmoid squashing function in every trial. Data was gathered at epochs 1000, 5000, and 25,000.

We selected vectors for the test suite to ensure variety both in the number of units in the program and in the nature of the program (number crunching programs tend to provide higher cyclomatic complexity values than I/O-bound programs). For a given package body, its cyclomatic complexity is equal to the sum of the cyclomatic complexities for all its procedures.

Some packages contained stub procedures. These stub procedures generate an edge value of zero and a node value of one and thus produce a cyclomatic complexity of 1. Stub procedures did not seem to adversely affect the training set.

The four figures below depict the results first when neural network architectures remain constant and training set size varies and second when training set size remains constant and neural network architectures vary.

As the training set increases, the results converge towards the x = y line, indicating a strong correspondence to the actual McCabe metric. This behavior occurs in all architectures; we show the 3-3-1 architecture in Figure 2, and the 3+5-5-1 architecture in Figure 3. Except for the initial

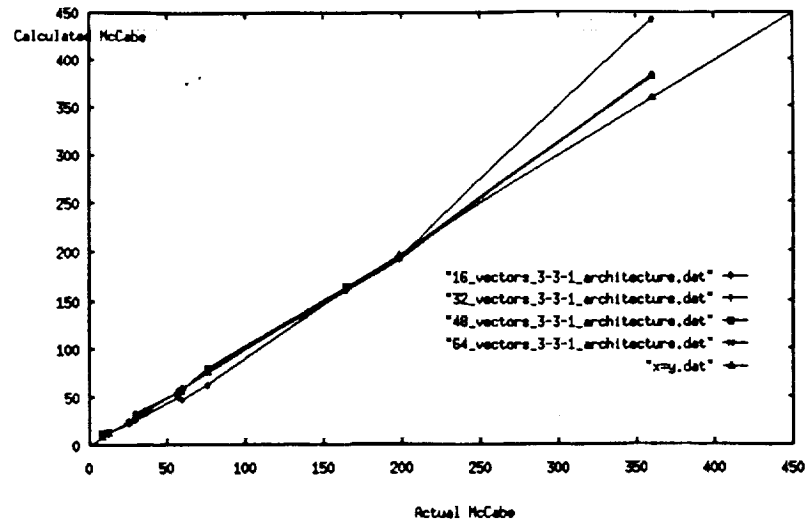


Figure 2: The 3-3-1 Architecture

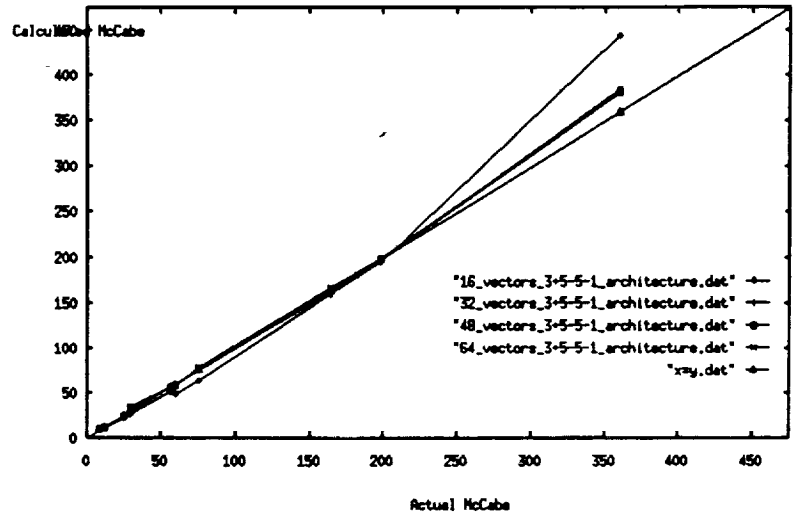


Figure 3: The 3+5-5-1 Architecture

improvement after 16 vectors, there is no significant improvement of results in the other three trials. This suggests that relatively low numbers of training vectors are required for good performance.

Furthermore, as shown in Figure 4 for 16 training vectors and Figure 5 for 64 training vectors, network architecture had virtually no effect on the results. These strong results are not surprising, given the linear nature of the McCabe metric.

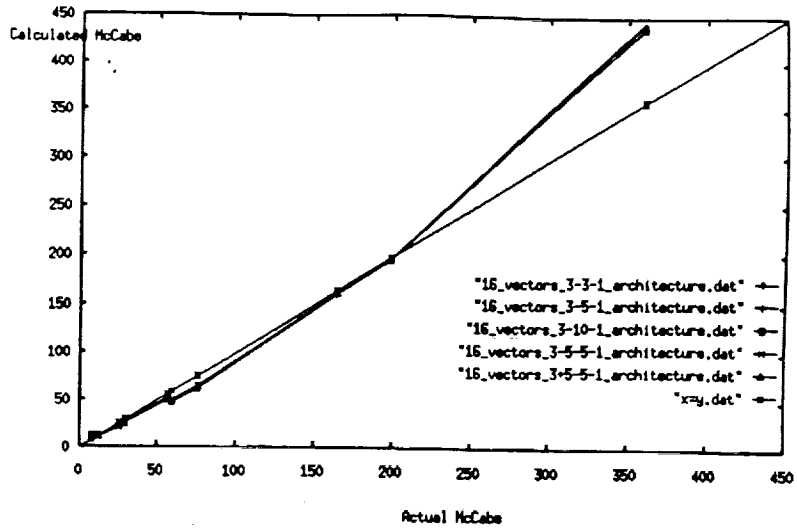


Figure 4: 16 Training Vectors for all Architectures

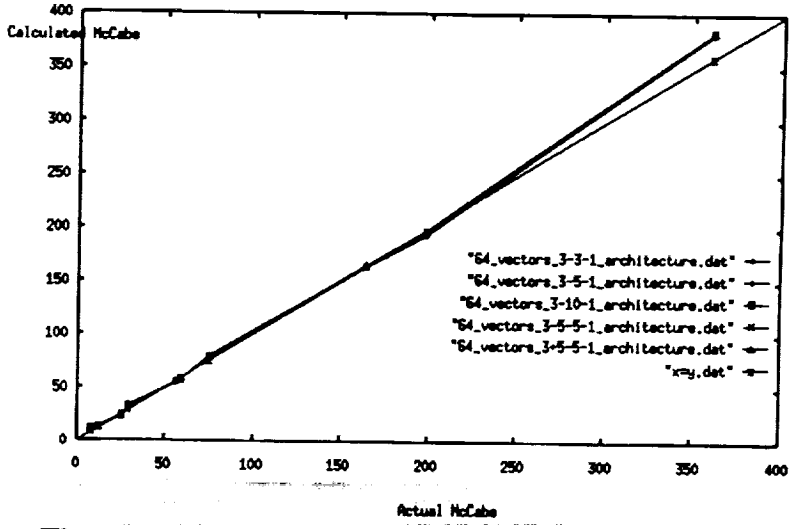


Figure 5: 64 Training Vectors for all Architectures

4.3 – Experiment C: A Neural Halstead Metric for Packages

Based upon the results of the first two experiments, we assumed for this experiment that if the experiment worked for packages, then it also worked for procedures, and further, that the increasing the number of training set vectors improves upon the results. Therefore, the focus of this experiment was on varying neural network architectures over a fixed-size training set.

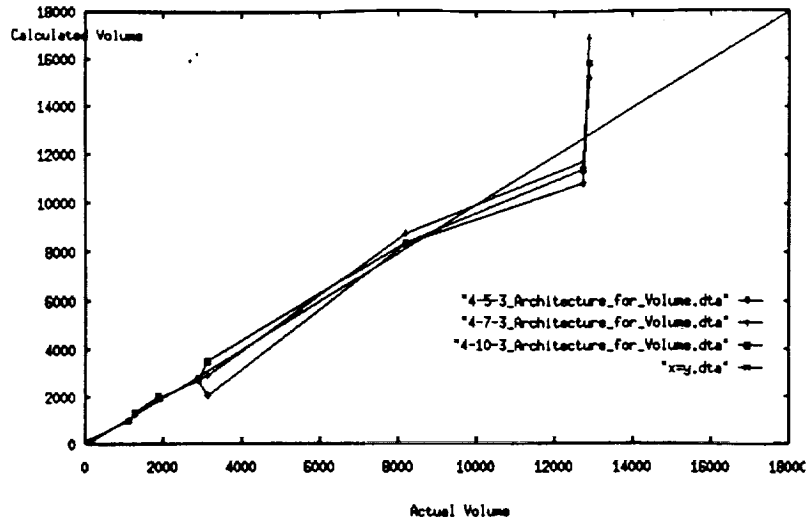


Figure 6: Volume Results, Broad Architectures

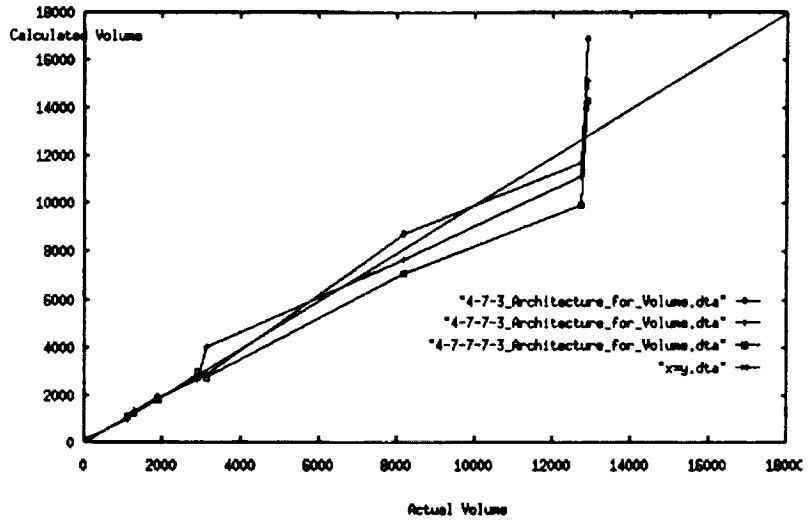


Figure 7: Volume Results, Deep Architectures

The experiment ranged over seven different neural network architectures broken into three groups: broad, shallow architectures (4-5-3, 4-7-3, and 4-10-3), narrow, deep architectures (4-7-7-3 and 4-7-7-7-3), and narrow, deep architectures with hidden layers that connected to all previous layers (4+7-7-3 and 4+7+7-7-3). We formed these three groups in order to discover whether there was any connection between the complexity of an architecture and its ability to model a metric.

Figures 6, 7, and 8 present the results for the Halstead volume for broad, deep, and connected

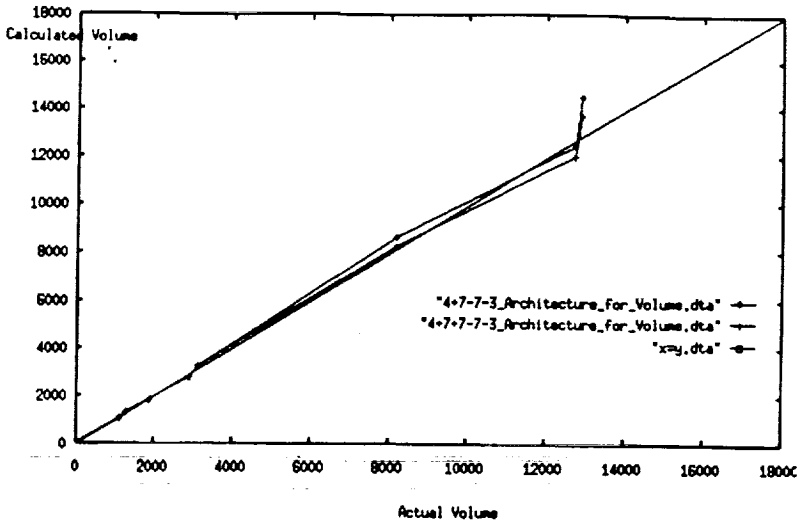


Figure 8: Volume Results, Connected Architectures

architectures, respectively. Note that both the broad and deep architectures do moderately well at matching the actual Halstead volume metric, but the connected architecture performs significantly better. Furthermore, there is no significant advantage for a five versus four layer connected architecture, indicating that connecting multiple layers may be a sufficient condition for adequately modeling the metric.

This pattern of performance also held for the Halstead length metric and the Halstead effort metric, so we show only the results for the connected architecture in Figure 9 and Figure 10, respectively.

5 – Conclusions

The experimental results clearly indicate that a neural network approach for modeling metrics is feasible. In all experiments the results corresponded well with the actual values calculated by traditional methods. Both the data set and the neural network architecture reached performance saturation points in the McCabe metric. In the Halstead experiment, the fact that the results oscillated over the actual-calculated line indicate that the neural network was attempting to model the desired values. Adding more training vectors, especially ones containing larger values, would smooth out

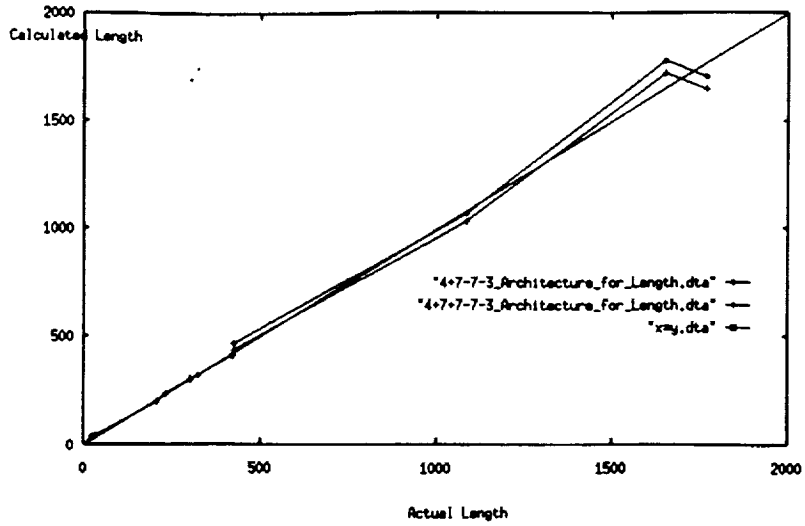


Figure 9: Length Results, Connected Architectures

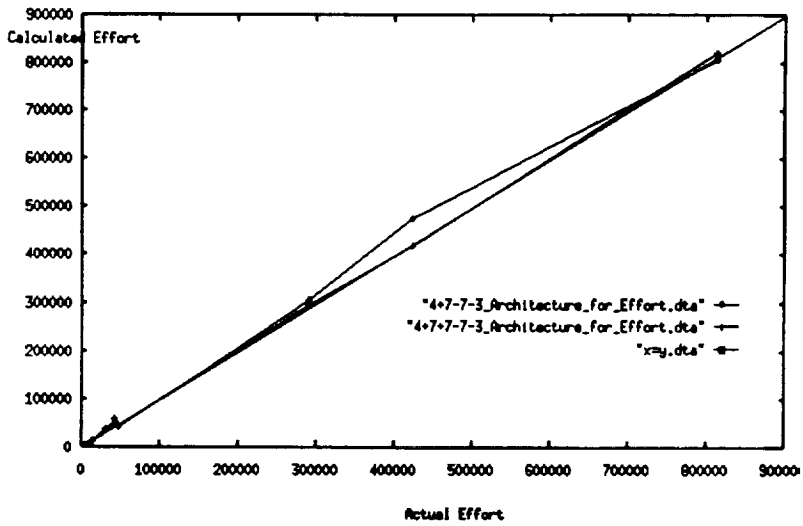


Figure 10: Effort Results, Connected Architectures

the oscillation.

6 – Future work

Applying this work to other existing metrics is an obvious extension, but we feel that the development of new metrics by applying neural approaches is much more significant. In particular, expanding this work to the development of a reusability metric offers great promise. Effective re-

use is only possible with effective assessment and classification. Since no easy algorithmic solutions currently exist, we've turned to neural networks to support the derivation of reusability metrics. Unsupervised learning provides interesting possibilities for this domain, letting the algorithm create its own clusters and avoiding the need for significant human intervention.

Coverage and accuracy are important aspects of developing a neural network to model a software reuse metric. McCabe and Halstead metrics are interesting and useful, but they do not provide coverage regarding reusability. We need to expand the number of parameters in the data set in order to provide adequate coverage with respect to reusability of a component. We also would like to improve the accuracy of answers by enlarging our data sets to include possibly hundreds of training set vectors. This will need to be a requirement when exploring more complex metric scenarios, and the cost of such extended training is easily borne over the expected usage of the metric.

Finally, it is possible to explore alternative neural network models. For example, the cascade correlation model [5] dynamically builds the neural network architecture, automating much of the process described here.

References

[1] Conn, R., "The Ada Software Repository and Software Reusability," *Proc. of the Fifth Annual Joint Conference on Ada Technology and Washington Ada Symposium,* 1987, p. 45-53.

[2] Emerson, T. J., "A Discriminant Metric for Module Cohesion," *Proc. 7th International Conference on Software Engineering,* Los Alamitos, California, IEEE Computer Society, 1984 p. 294-303.

[3] Emerson, T. J., "Program Testing, Path Coverage, and the Cohesion Metric," *Proc. of the 8th Annual Computer Software and Applications Conference,* IEEE Computer Society, p. 421-431.

[4] Fahlman, S. E., *An Empirical Study of Learning Speed in Back-Propagation Networks,* Tech Report CMU-CS-88-162, Carnegie Mellon University, September, 1988.

[5] Fahlman, S. E. and Lebiere, M., *The Cascade-Correlation Learning Architecture*, Tech Report CMU-CS-90-100, Carnegie Mellon University, August 1991.

[6] Halstead, M.H., *Elements of Software Science*, New York: North-Holland (Elsevier Computer Science Library), 1977.

[7] Hertz, J., Krogh A., Palmer, R. G., *Introduction to the Theory of Neural Computation*, Addison Wesley, New York, 1991.

[8] Li, H.F. and Cheung, W.K., "An Empirical Study of Software Metrics," *IEEE Transactions on Software Engineering*, (13)6, June 1987, p. 697-708

[9] Lippmann, R. P. "An Introduction to Computing with Neural Nets," *IEEE ASSP Magazine*, April 1987, p. 4-22.

[10] McCabe, T.J., "A complexity measure," *IEEE Transactions on Software Engineering*, (SE-2) 4, Dec. 1976, p. 308-320.

[11] McCabe, T.J., "Design Complexity Measurement and Testing," *Communications of the ACM*, (32)12, December 1989, p. 1415-1425.

Balancing Generality and Specificity in Component–Based Reuse*†

David Eichmann and Jon Beck

Software Reuse Repository Lab
Dept. of Statistics and Computer Science
West Virginia University

Send correspondence to:
David Eichmann
SoRReL
Dept. of Statistics and Computer Science
West Virginia University
Morgantown, WV 26506

Email: eichmann@cs.wvu.wvnet.edu

* Submitted to *The International Journal of Software Engineering and Knowledge Engineering*.
† This work was supported in part by NASA as part of the Repository Based Software Engineering project, cooperative agreement NCC-9-16, project no. RICIS SE.43, subcontract no. 089 and in part by a grant from MountainNet Inc.

Abstract

For a component industry to be successful, we must move beyond the current techniques of black box reuse and genericity to a more flexible framework supporting customization of components as well as instantiation and composition of components. Customization of components strikes a balance between creating dozens of variations of a base component and requiring the overhead of unnecessary features of an "everything but the kitchen sink" component. We argue that design and instantiation of reusable components have competing criteria – design-for-reuse strives for generality, design-with-reuse strives for specificity – and that providing mechanisms for each can be complementary rather than antagonistic. In particular, we demonstrate how program slicing techniques can be applied to customization of reusable components.

1 – Introduction

The impediments to a successful reuse infrastructure in the software engineering community have typically been separated into social and technological issues [26]. Furthermore, the social issues (e.g., comprehension, trust, and investiture) often are characterized as being the more critical, as there is a perception that all of the technical issues (e.g., environments, repositories, and linguistic support) have been solved [27]. We do not agree with this assessment (see [8] for our arguments regarding repositories and environments), and furthermore believe that appropriate application of technology can alleviate certain of the social issues just mentioned.

This paper addresses two reuse impediments – component comprehension by a reuser [14] and the fitness of a component for a given application – and how technical support, in this case language features and program slicing, alleviate these impediments. These two impediments drive the consumer side of reuse repository design, for without comprehensibility users will not select artifacts from the repository, and without adequate conformance to requirements users will not incorporate artifacts into systems even if they do select them. These two impediments also drive the design process for reusable components, since components perceived as ill-suited for reusers' application domains (and hence not incorporated into the resulting systems) have not met the requirements of a design-for-reuse effort.

We begin in section 2 by characterizing the inherent conflict between the design goals for design-for-reuse and design-with-reuse. We then review mechanisms that support particular structural and behavioral aspects of component design in section 3. The mechanisms described support flexibility in the ***design*** of a component. We consider mechanisms in section 4 to constrain an implementation, supporting specificity in the ***instantiation*** of a component, and show in section 5 how to employ program slicing as one such mechanism. Section 6 demonstrates the application of our technique to a moderate-sized example.

2 – Design-For-Reuse versus Design-With-Reuse

Design for reuse focuses on the potential reusability of the artifacts of a design process. *Design with reuse*, on the other hand, focuses on employing existing artifacts wherever possible in the design process. The intent of the two approaches, and hence the various criteria that each of them employ, is then quite distinct. In particular, design for reuse strives for generality, even to the point of additional cost to the current project, and design with reuse strives to reduce cost to the current project, even to the point of adapting non-critical project requirements to achieve conformance with existing artifacts.

Garnett and Mariani proposed the following attributes for reusable software [10]:

- environmental independence – no dependence on the original development environment;
- high cohesion – implementing a single operation or a set of related operations;
- loose coupling – minimal links to other components;
- adaptability – easy customization to a variety of situations;
- understandability;
- reliability; and
- portability.

These attributes clearly reflect goals that should apply to all products of a design-for-reuse effort, and some of these attributes (particularly understandability and reliability) apply to all software development efforts. Not so clear is whether these attributes reflect the goals of design-with-reuse efforts.

We contend that there is an inherent conflict between design-for-reuse and design-with-reuse that centers upon adaptability. Design-for-reuse strives to create artifacts that are as generally applicable as possible, in the worst case creating "everything-but-the-kitchen-sink" artifacts, loading a component with features in an effort to ensure applicability in all situations. Design-with-reuse strives to identify that artifact which most specifically matches a given requirement. Anything less

requires additional effort, both in comprehension and coding. Anything more carries with it the penalty of excess resource consumption and increased comprehension effort.

The specificity that we seek in design-with-reuse takes two forms – the first is that of avoiding additional functionality in a simple component; the second is that of avoiding additional functionality in an abstraction, implemented as a package/module. Specificity becomes increasingly critical when considering scale. The additional storage consumed and increased comprehension effort posed by a simple abstract data type quickly become the multi-megabyte "hello world" applications of today's user interface management systems, and threaten intractability in the domain of megaprogramming [4, 19].

3 – Language Mechanisms Supporting Design–For–Reuse

Designing a software component for reuse involves a number of issues, including analysis of the intended target domain [21, 22], the coverage that this component should provide for the domain [22], and the nature and level of parameterization of the component [7, 28, 29]. A number of developments in programming language design directly bear upon these issues. We focus here upon those we see as most beneficial.

3.1 – Procedural and Modular Abstraction

The obvious advantages that functions and procedures provide in comprehension and reuse of portions of a program (even if the reuse is only at a different location in the same program) are so well recognized, that no contemporary language proposal is taken seriously without them. The package (or module) concept, with separate specification and implementation of a collection of data and procedural definitions, has arguably reached the same level of acceptance. Sommerville's list of classes of reusable components (functions, procedures, declaration packages, objects, abstract data types, and subsystems) [25] indicates the depth of this acceptance – virtually every class listed is directly implementable using one of the two mechanisms (objects being the only non-obvious fit).

3.2 – Parameterization and Genericity

The utility of a function or procedure is severely limited without the ability to provide information customizing the effect of a specific invocation. Parameters comprise the explicit contract between a function and its invocations, and are generally accepted as far preferable to the implicit contract provided by shared global state. Genericity, or more formally, parametric polymorphism [6], involves the parameterization of program units (both functions/procedures and packages/modules) with types, variables, and operations (functions, procedures, tasks, and exceptions). Parameters effectively support families of *invocations*. Genericity extends this support to families of *instantiations*, each with its own family of invocations, providing increased adaptability and portability [28].

3.3 – Inheritance

Inheritance involves the creation of generalization/specialization structures, a tree in the case of single inheritance, a lattice in the case of multiple inheritance. These generalizations/specializations may be structural (in the case of subtypes [6]) or behavioral (in the case of classes [11]). Whatever the structuring mechanism, inheritance supports the creation of variations of a base component, each with its own interface [15], as well as instances of those variations. Inheritance thus is a very useful mechanism for the creation of certain classes of software artifacts. Note, however, that using inheritance as a reuse-enabling mechanism is not without its own hazards, most notably scalability and the violation of information hiding [23, 24].

4 – Language Mechanisms Supporting Design-With-Reuse

The previous section primarily addressed the *creation* of program structure. Our primary interest in this section involves not the creation of new reusable components, but rather their natural involvement in the development process. This corresponds to the responsibilities of Basili's project organization [3].

4.1 – Procedural and Modular Abstraction

Much of today's reuse takes place at the level of procedures and packages, either as source or object code. The linguistic and environmental mechanisms for this, including source and object libraries and separate compilation, provide little over what a simple text editor with cut and paste commands provides. The onus of comprehension and adaptation is placed upon the reuser, particularly if the reuser is interested in increasing the specificity of the component (which may even be proscribed by the social infrastructure, i.e. management). The consequence of design-with-reuse in this context is thus *monolithic* reuse, an all or nothing acceptance of an entire component.

4.2 – Genericity

Genericity readily supports the creation of specializations of the generic artifact through instantiation. However, genericity as defined in languages such as Ada provides little beyond complete instantiation of a generic component into a completely concrete instance. Further, partial instantiation does little in terms of additional flexibility, as every successive partial instantiation makes the resulting generic more concrete. Hence genericity provides the same form of monolithic reuse as that described in the previous section, with the option of customizing the instances.

4.3 – Inheritance

Inheritance performs as readily in support of a reuser as in support of a developer of components. The reuser can both instantiate new instances of the component and derive new component classes from the original. This second issue is a particularly beneficial one, as it allows for the development of unanticipated refinements to the program model without requiring adaptation of existing code. However, inheritance exhibits the same specificity limitations as abstraction and genericity, supporting only monolithic reuse, in the case of instantiation, or incremental monolithic reuse, in the case of class refinement.

5 – Program Slicing

The mechanisms discussed in sections 3 and 4 *add* structure and/or complexity to a program. Parameterization and genericity increase the interface complexity of a program unit. Packages and inheritance increase either the number of program units or the structural complexity of those units. Hence, current languages do not have explicit mechanisms that address the conflicting goals of design-for-reuse and design-with-reuse. We therefore propose a new mechanism for reconciling the two approaches (by increasing component structural specificity) which works in conjunction with the facilities provided in Ada – a new form of program slicing. We use Ada for our examples, as it is a language whose built-in features facilitate the types of transformations which we invoke. However, the concepts we present are not confined to any particular language.

In his thesis [30], Weiser introduced the concept of program slicing. In this form of slicing, called *static slicing*, a *slice* of a program is an executable subset of the source statements which make up program. A slice is specified by a variable and a statement number, and consists of all statements which contribute to the value of that variable at the end of execution of that statement, together with any statements needed to form a properly executing wrapper around the slice proper.

Dynamic slicing, [1, 2, 17] is a second form of slicing which is determined at runtime and is dependent on input data. A dynamic slice is the trace of all statements executed during a program run using a particular input data set, refined by specifying only those executed statements which reference a specified set of variables. Dynamic slicing was specifically designed as an aid in debugging, and is used to help in the search for offending statements in finding a program error.

By definition, static slicing is a pre-compilation operation, while dynamic slicing is a run-time analysis. Our interface slicing belongs in the category of static slicing, as it is a data-independent pre-compilation code transformation. Since our interest here is only with static slices, henceforth we will use *slicing* to mean static slicing, and we will not again discuss dynamic slicing.

```
 1  procedure wc (theFile : in string; nl, nw, nc : out natural := 0) is
 2  inword : boolean := FALSE;
 3  theCharacter : character;
 4  file : file_type;
 5  begin
 6      open(file, IN_FILE, theFile);
 7      while not end_of_file(file) loop
 8          get(file, theCharacter);
 9          nc := nc + 1;
10          if theCharacter = LF then
11              nl = nl + 1;
12          end if;
13          if theCharacter = ' '
14           or theCharacter = LF
15           or theCharacter = HT then
16              inWord = FALSE;
17          else if not inWord then
18              inWord = TRUE;
19              nw = nw + 1;
20          end if;
21      end loop;
22      close(file);
23  end wc;
```

Figure 1: wc, a procedure to count text

5.1 – Previous Work in Slicing

In his thesis [30] and subsequent work [31, 32, 33], Weiser used slicing to address various issues primarily concerned with program semantics and parallelism. Gallagher and Lyle more recently employed a variation of slicing in limiting the scope of testing required during program maintenance [20].

Program slicing has been proposed for such uses as debugging and program comprehension [32], parallelization [5], merging [12, 18], maintenance, and repository module generation [9].

As an example of program slicing, we present the following example, adapted from Gallagher & Lyle [9]. The procedure `wc`, presented in Figure 1, computes the count of lines, words, and characters in a file.* Figure 2 gives the results of slicing `wc` on the variable `nc` at the last line of the procedure. Since the variables `nl`, `nw`, and `inword` do not contribute to the value of `nc`, they do not appear in the slice. Also, the statements on lines 10 through 20 of the original procedure do not

* This procedure is not entirely correct, since the Ada *get* procedure skips over line terminators, unlike the C *getchar* function. We adapted wc in this way to clarify its actions and retain the flavor of the original function.

```
 1  procedure wc (theFile : in string; nc : out natural := 0) is
 2  theCharacter : character;
 3  file : file_type;
 4  begin
 5      open(file, IN_FILE, theFile);
 6      while not end_of_file(file) loop
 7          get(file, theCharacter);
 8          nc := nc + 1;
 9      end loop;
10      close(file);
11  end wc;
```

Figure 2: wc sliced on nc

appear in the slice. While this slice follows the spirit of a classic slice, and will serve to illustrate classic slicing, it also differs in several important ways, as described below.

5.2 – Interface Slicing

We propose a new form of slicing, *interface slicing*, which is performed not on a program but on a component. Similar to previous work in static slicing, our interface slice consists of a compilable subset of the statements of the original program. The interface slice is defined such that the behavior of the statements and the values of the variables in the slice is identical to their behavior and values in the original program.

However, while previous slicing efforts have attempted to isolate the behavior of a set of variables, even across procedural boundaries, our slice seeks rather to isolate portions of a component which export the behavior we desire. In the following discussion, we assume for simplicity that a package implements a single ADT, and we use package and ADT interchangeably.

Unlike standard slicing techniques which are usually applied to an entire program, interface slicing is done on a fragment of a program – a *component* – since our goal is to employ the necessary and sufficient semantics of a component for use in the target system. Interface slicing is at the level of procedures, functions, and task types. If a procedure is invoked at all, the entire procedure must be included, as we have no way of knowing *a priori* what portion of the procedure will be needed.* However, if an ADT is incorporated into a system, not necessarily all of its operations are

invoked. The interface slicing process determines which operations are to be included, and which can be eliminated. Because interface slicing treats procedures atomically, the complex program dependence graph analysis of standard slicing [13] is not necessary. A single pass of the call graph of an ADT's operations is sufficient to determine the slice. We use "operation" as a general term to encompass procedures, functions, and exceptions, and include tasks with procedures in that a task is another way of encapsulating a subprogram unit.

We will illustrate the concept of interface slicing first by examining a simple example, a toggle ADT. First consider package `toggle1`, in Figure 3. This package exports the public operations `on`, `off`, `set`, and `reset`. `On` and `off` are examination operations which query the state of the toggle, while `set` and `reset` are operations which modify the state of the toggle. Now suppose that we wish to have a toggle in a program which we are writing, but we have a need for only three of the four operations, namely `on`, `set`, and `reset`. In standard Ada, we have two choices. We can include the package as is, and have the wasted space of the off operation included in our program. This is the kitchen sink syndrome. Alternatively, we can edit the source code manually (assuming we have access to it) and remove the `off` operation, thereby saving space, but requiring a large amount of code comprehension and introducing the danger of bugs due to hidden linkages and dependencies. In both these cases, we see the generality of design-for-reuse competing with the desired specificity of design-with-reuse.

Instead, we propose the invocation of an interface slicing tool to which we give the `toggle1` package together with the list of operations we wish to include in our program. The tool then automatically slices the entire package based on the call graph of its operations, generating a slice containing only those operations (and local variables) needed for our desired operations. The slice of `toggle1` which contains only the three operations is shown in Figure 4.

* In other words, an interface slice is orthogonal to a standard static slice. The use of one technique neither requires nor inhibits the use of the other. We are not discussing the technique of standard static slicing here, other than to contrast it with our interface slice, and so we do *not* assume that an interprocedural slicer is operating at the same time as our interface slicer.

```
 1  package toggle1 is
 2
 3      function on return boolean;
 4
 5      function off return boolean;
 6
 7      procedure set;
 8
 9      procedure reset;
10
11  end toggle1;
12
13  package body toggle1 is
14
15      theValue : boolean := FALSE;
16
17      function on return boolean is
18      begin
19          return theValue = TRUE;
20      end on;
21
22      function off return boolean is
23      begin
24          return theValue = FALSE;
25      end off;
26
27      procedure set is
28      begin
29          theValue := TRUE;
30      end set;
31
32      procedure reset is
33      begin
34          theValue := FALSE;
35      end reset;
36
37  end toggle1;
```

Figure 3: A toggle package

As another example, consider the package `toggle2`, which in addition to the operations of toggle1 includes the operation swap. This package is shown in Figure 5. Suppose we wish to write a program which needs a toggle ADT and the operations `on` and `swap`. The interface slicing tool finds that the operation `on` has no dependencies, but the operation `swap` needs `on`, `set`, and `reset`, and so the desired slice of toggle2 which is produced for our program is contains the four operations, `on`, `set`, `reset`, and `swap`, and does not contain `off`. This slice is shown in Figure 6.

One of the differences between interface slices and standard slices is the way that interface slices are defined. While a standard slice is defined by a slicing criterion consisting of a program, a statement and a set of variables, an interface slice is defined by a package and a set of operations

```
 1  package toggle1 is
 2
 3      function on return boolean;
 4
 5      procedure set;
 6
 7      procedure reset;
 8
 9  end toggle1;
10
11  package body toggle1 is
12
13      theValue : boolean := FALSE;
14
15      function on return boolean is
16      begin
17          return theValue = TRUE;
18      end on;
19
20      procedure set is
21      begin
22          theValue := TRUE;
23      end set;
24
25      procedure reset is
26      begin
27          theValue := FALSE;
28      end reset;
29
30  end toggle1;
```

Figure 4: The toggle package sliced by on, set and reset

in its interface. The package is an example of design-for-reuse and implements a full ADT, complete with every operation needed to legally set and query all possible states of the ADT. The interface slicer is an aid to design-with-reuse and prunes the full ADT down to the minimal set of operations necessary to the task at hand. The interface slicer does not add functionality to the ADT, as the ADT contains full functionality to start with. Rather, the slicer eliminates unneeded functionality, resulting in a smaller, less complex source file for both compiler and reuser to deal with, and smaller object files following compilation.

6 – An Extended Example

The examples above illustrate the general concept of interface slicing, but leave out some important details. To fill in some of these details, we will next examine a pair of generic packages in the public domain. These packages were explicitly written to be used as building blocks for Ada

```
 1  package toggle2 is
 2
 3      function on return boolean;
 4
 5      function off return boolean;
 6
 7      procedure set;
 8
 9      procedure reset;
10
11      procedure swap;
12
13  end toggle2;
14
15  package body toggle2 is
16
17      theValue : boolean := FALSE;
18
19      function on return boolean is
20      begin
21          return theValue = TRUE;
22      end on;
23
24      function off return boolean is
25      begin
26          return theValue = FALSE;
27      end off;
28
29      procedure set is
30      begin
31          theValue := TRUE;
32      end set;
33
34      procedure reset is
35      begin
36          theValue := FALSE;
37      end reset;
38
39      procedure swap is
40      begin
41          if on then
42              reset;
43          else
44              set;
45          end if;
46      end swap;
47
48  end toggle2;
```

Figure 5: Version 2 of the toggle package

programs. The first is a generic package which provides the ADT *set*. The package is instantiated by supplying it with two parameters, the first being the type of element which the set is to contain, and the second a comparison function to determine the equality of two members of this type. The package provides all the operations necessary to create, manipulate, query, and destroy sets. The full interface specification of the set is given in Appendix A.

```
 1  package toggle2 is
 2
 3      function on return boolean;
 4
 5      procedure swap;
 6
 7  end toggle2;
 8
 9  package body toggle2 is
10
11      theValue : boolean := FALSE;
12
13      function on return boolean is
14      begin
15          return theValue = TRUE;
16      end on;
17
18      procedure set is
19      begin
20          theValue := TRUE;
21      end set;
22
23      procedure reset is
24      begin
25          theValue := FALSE;
26      end reset;
27
28      procedure swap is
29      begin
30          if on then
31              reset;
32          else
33              set;
34          end if;
35      end swap;
36
37  end toggle2;
```

Figure 6: Version 2 of toggle sliced by on and swap

This *set* package happens to use a *list* as the underlying representation upon which it builds the set ADT, and so requires the second generic package which supplies the *list* ADT. This happens to be a singly-linked list implementation which exports all the operations necessary to create, manipulate, query, and destroy lists. This package also requires two generic parameters, the same ones which *set* requires. The specification for the list package is given in Appendix B.

In the particular list and set packages we used for our example, there were no private operations. Private operations are not available to be used in an interface slicing criterion; only the exported operations in the interface can be in the slicing criterion. In general, however, private operations are treated identically to exported ones during the slicing process. The slicer, being a

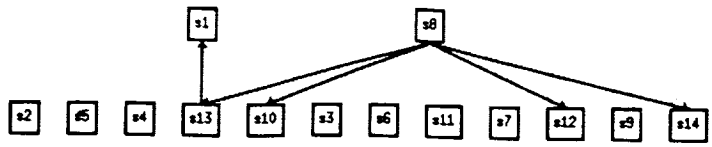


Figure 7: The call graph for set

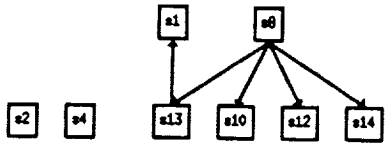


Figure 8: The sliced set

privileged pre-compilation code transformer, does not respect privacy.

6.1 – A Single Level of Slicing

Now suppose we wish to use the set package in a program we are writing, but we have a need for only a few of the set operations, specifically, in this example, *create*, *insert*, and *equal*. We would like to include all the code necessary to accomplish these operations, but would like to have *only* the necessary code, and no more.

In order to slice the set package, we must examine the call graph of operations in the set package for the transitive closure of the three desired operations. Figure 7 shows the complete call graph of the set package, and figure 8, shows the transitive closure of *create*, *insert*, and *equal* (nodes s2, s4 and s8, respectively).* Figure 8 shows the slice corresponding to these three operations. Out of the total of 14 operations exported by the original package, the slice based on *create*, *insert*, and *equal* contains only 8 operations, with a considerable reduction in total size of code, although the complexity of the call graph remains the same.

Notice that in this example, the sliced set package needs the same number and type of generic parameters as did the original package. This will not always be the case, however. In Figure 1, the

* The call graph node labels correspond to the comments associated with each operation for the package specifications appearing in the appendices.

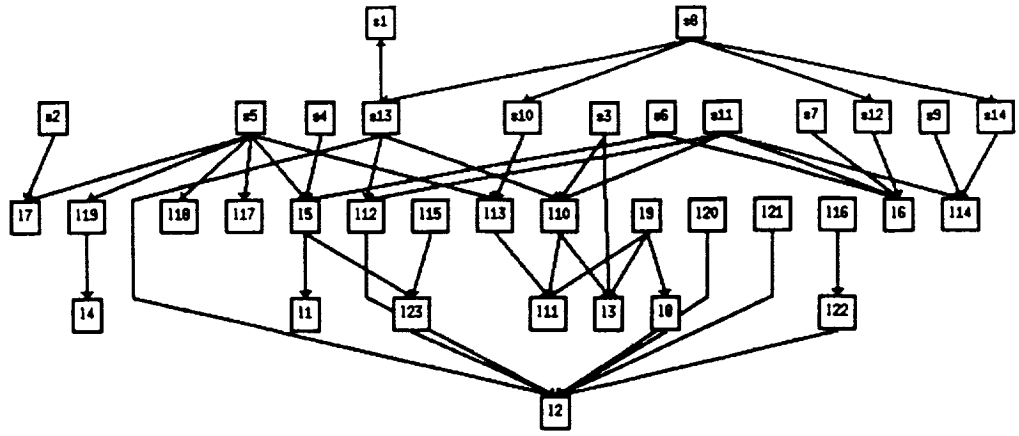


Figure 9: The combined set and list call graph

original wc procedure needed 4 parameters, but the slice based on nc shown in Figure 2 needed only 2 parameters. In general, out of all the local variables in a component, including both variables bound to parameters and those declared within the component's scope, a slice will include a subset of these local variables.

6.2 – A Second Level of Slicing

While the 8 operations represent an improvement over the original 14, we can go further, and examine not only the set package, but also the list package as well. If we examine the transitive closure of the three desired operations in the call graph of all the operations of both the set and list packages, we can accomplish a much more dramatic improvement in the size and complexity of the resulting slice. Figure 9 shows the full call graph of the set and list packages. In standard Ada usage, all of this would be included in a program were the generic set and list packages instantiated in a program. Figure 10 shows the call graph which is exactly the transitive closure of the set operations *create*, *insert*, and *equal*, as would be produced by interface slicing. The size and complexity of this call graph are obviously much less than that of the full graph. Table 1 gives some statistics on the relative sizes of the packages and their call graphs.

None of the examples above involved overloaded names. Interface slicing in the presence of overloading is somewhat more complicated. Assuming that the resolution can be accomplished

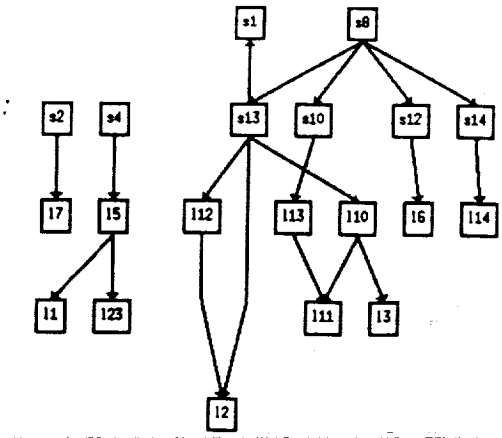


Figure 10: The sliced set and list

Table 1: Package Statistics

	# of nodes	# of edges	# of statements
Full Set	14	5	95
Sliced Set	8	5	57
% reduction	36	0	40
Full Set and List	37	46	345
Sliced Set and list	20	19	200
% reduction	46	59	42

completely at compile time, there are two options. The first is a simple, naive approach in which all versions of an overloaded operation are included. The second is to perform the type checking for parameters and return value (if any) to determine which of the overloaded versions are actually called. For example, assume that list's operation *attach* is a quadruply overloaded procedure which can be called with two elements, an element and a list, a list and an element, or two lists. Resolution of the overloading may, in a particular situation, allow three of the four procedures to be sliced away, resulting in improved reduction of size and complexity.

If the overloading cannot be resolved at compile time, but must wait until runtime, we have no option but to include the code for all possible operations which may be called. A static slice can only blindly assume worst-case in the presence of run-time binding of overloaded procedure

names. Although our example extends to only two levels, the slicing can extend to as many levels as exist in the compilation dependency graph of the packages included in the program.

7 – Conclusion: Balancing Genericity and Specificity

We have discussed two main reuse-oriented paradigms in software engineering, namely design-for-reuse and design-with-reuse, and how the goals of these two paradigms have in the past been viewed as being antagonistic, with the former striving for generality and the latter striving for specificity. We have shown that with the proper language mechanisms and development techniques, the goals are in fact complementary. The specific mechanism we use by way of example is a new form of static program slicing which we call interface slicing. Using interface slicing, a complete and generic component can be adapted to the specific needs of the program at hand, increasing comprehension and reducing complexity, without sacrificing the generality of the base component. Thus a developer designing a component for reuse can be completely unfettered of all size constraints and strive for total generality, knowing that a reuser of the components can effortlessly have all unneeded functionality sliced away in a pre-compilation step.

The artifacts produced by an interface slicer should not be considered as new components, any more than instantiations of a generic are viewed as new components. Rather, we want to emphasize the retention of the derivation *specification*, avoiding additional maintenance problems though the life-cycle of what would then be custom components. We should keep the desired interface specification, and alter that when we need to change the way in which we bind through the interface to the base component. Just as we don't associate any cost per se with the instantiation of a generic, we should not associate a cost with specialization through interface slicing, since it can be completely handled by the development environment.

Our approach addresses indirectly a critical social aspect of reuse, the trust that reusers place in the components extracted from the repository [16]. Deriving a family of interface slices from a

base component implies that if the base component is correct (or at least certified), then all of the slices must necessarily be correct (or at least certified) also.

References

1 H. Agrawal and J. Horgan, *Dynamic Program Slicing*, Technical Report SERC-TR-56-P, Software Engineering Research Center, Purdue University, West Lafayette, Indiana, December 1989.

2 H. Agrawal, R. DeMillo, and E. Spafford, *Efficient Debugging with Slicing and Backtracking*, Technical Report SERC-TR-80-P, Software Engineering Research Center, Purdue University West Layfayette, Indiana, October 1990.

3 V. R. Basili, G. Caldiera, and G. Cantone, "A Reference Architecture for the Component Factory," *ACM Transactions on Software Engineering and Methodology*, 1(1), January 1992, p. 53-80.

4 D. Batory, "On the Differences Between Very Large Scale Reuse and Large Scale Reuse," *Proc. 4th Annual Workshop on Software Reuse*, Reston VA, November 18-22 1991.

5 W. Baxter and H. R. Bauer, "The Program Dependence Graph and Vectorization," *Proc. Principles of Programming Languages: 16th Annual ACM Symposium*, Austin, TX, January 11-13, 1989, p. 1-11.

6 L. Cardelli and P. Wegner, "On Understanding Types, Data Abstraction, and Polymorphism," *ACM Computing Surveys*, 17(4), December 1985, p. 471-522.

7 S. H. Edwards, *An Approach for Constructing Reusable Software Components in Ada*, IDA Paper P-2378, Institute for Defense Analyses, Alexandria VA, Sept. 1990.

8 D. Eichmann, "A Repository Architecture Supporting Both Intra-Organizational and Inter-Organizational Reuse," to be submitted.

9 K. B. Gallagher and J. R. Lyle, "Using Program Slicing in Software Maintenance," *IEEE Transactions on Software Engineering*, 17(8), August 1991, p. 751-761.

10 E. S. Garnett and J. A. Mariani, "Software Reclamation," *Software Engineering Journal*, (5)3, May 1990, p. 185-191.

11 A. Goldberg and D. Robson, *Smalltalk-80: The Language and Its Implementation*, Addison-Wesley, 1983.

12 S. Horwitz, J. Prins, and T. Reps, "Integrating Non-interfering Versions of Programs," *Proc. Fifteenth Annual ACM Symposium on Principles of Programming Languages*, New York, January 13-15, 1988, p. 133-145.

13 S. Horwitz, T. Reps, and D. Binkley, "Interprocedural Slicing Using Dependence Graphs," *ACM Transactions on Programming Languages and Systems*, (12)1, p. 26-60, January 1990.

14 K. E. Huff, R. Thomson, and J. W. Gish, "The Role of Understanding and Adaptation in Software Reuse Scenarios," *Proc. 4th Annual Workshop on Software Reuse*, Reston VA, November 18-22 1991.

15 R. E. Johnson and B. Foote, "Designing Reusable Classes," *Journal of Object-Oriented Programming*, 1(2), June/July 1988, p. 22-35. Also appears in [22].

16 J. C. Knight, "Issues in the Certification of Reusable Parts," *Proc. 4th Annual Workshop on Software Reuse*, Reston VA, November 18-22 1991.

17 B. Korel and J. Laski, "Dynamic program slicing," *Information Processing Letters*, (29)3, p. 155-163, October 1988.

18 A. Lakhotia, *Graph Theoretic Foundations of Program Slicing and Integration*, Technical Report CACS-TR-91-5-5, Center for Advanced Computer Studies, University of Southwestern Louisiana Lafayette, LA, December 2, 1991.

19 H. Li and J. van Katwijk, "A Model to Reuse-in-the-Large," *Proc. 4th Annual Workshop on Software Reuse*, Reston VA, November 18-22 1991.

20 J. R. Lyle and K. B. Gallagher, "A Program Decomposition Scheme with Applications to Software Modification and Testing," *Proceedings of the 22nd Hawaii International Conference on System Sciences*, vol. 2, January 1989, p. 479-485.

21 R. Prieto-Díaz, "Domain Analysis for Reusability," *Proceedings of COMPSAC '87*, p. 24-29. Also appears in [22].

22 R. Prieto-Díaz and G. Arango, *Domain Analysis and Software Systems Modeling*, IEEE Computer Society Press, 1991.

23 R. K. Raj and H. M. Levy, "A Compositional Model for Software Reuse," *The Computer Journal*, (32)4, 1989, p. 312-322.

24 A. Snyder, "Encapsulation and Inheritance in Object-Oriented Programming Languages," *Proc. OOPSLA'86*, Portland OR, September 29 - October 2 1986, p. 38-45.

25 I. Sommerville, *Software Reuse*, ISF Study Paper ISF/UL/WP/IS-3.1, University of Lancaster, UK, January 1988.

26 W. Tracz, "Software Reuse: Motivators and Inhibitors," *Proc. of COMPCON '87*, 1987, p. 358-363.

27 W. Tracz, "Software Reuse Myths," *ACM SIGSOFT Software Engineering Notes*, (13)1, January 1988, p. 17-21.

28 W. Tracz, "Parameterization: A Case Study," *Ada Letters*, (IX)4, May/June 1989, p. 92-102.

29 B. W. Weide, W. F. Odgen, and S. H. Zweben, "Reusable Software Components," in *Advances in Computers*, v. 33, M. C. Yovits (ed.), Academic Press, 1991, p. 1-65.

30 M. Weiser, *Program Slicing: Formal, Psychological and Practical Investigations of an Automatic Program Abstraction Method*, PhD Thesis, University of Michigan, Ann Arbor, Michigan, 1979

31 M. Weiser, "Program slicing," *Proceedings of 5th International Conference on Software Engineering*, p. 439-449, May 1981.

32 M. Weiser, "Programmers use slicing when debugging," *Communications of the ACM*, 25(7), July 1982, p. 446-452.

33 M. Weiser, "Program Slicing," *IEEE Transactions on Software Engineering*, SE-10, July 1984, p. 352-357.

Appendix A – The Package Specification for Set

Note: the comments in the right margin refer to the node labels in the call graphs of Figures 7, 8, 9, and 10.

```
 1  generic
 2      type elemType is private;
 3      with function equal(e1, e2: elemType) return boolean is "=";
 4  package setPkg is
 5
 6      type set is private;
 7      type iterator is private;
 8
 9      noMore: exception;                                          -- s1
10
11      function create return set;                                 -- s2
12
13      procedure delete(s: in out set; e: in elemType);            -- s3
14
15      procedure insert(s: in out set; e: in elemType);            -- s4
16
17      function intersection(s1, s2: set) return set;              -- s5
18
19      function union(s1, s2: set) return set;                     -- s6
20
21      function copy(s: set) return set;                           -- s7
22
23      function equal(s1, s2: set) return boolean;                 -- s8
24
25      function isEmpty(s: set) return boolean;                    -- s9
26
27      function isMember(s: set; e: elemType) return boolean;      -- s10
28
29      function size(s: set) return natural;                       -- s11
30
31      function makeIterator(s: set) return iterator;              -- s12
32
33      procedure next(iter: in out iterator; e: out elemType);     -- s13
34
35      function more(iter: iterator) return boolean;               -- s14
36
37  end setPkg;
```

Appendix B – The Package Specification for List

Note: the comments in the right margin refer to the node labels in the call graphs of Figures 9 and 10.

```
 1  generic
 2      type elemType is private;
 3      with function equal(e1, e2: elemType) return boolean is "=";
 4  package listPkg is
 5
 6      type list is private;
 7      type iterator is private;
 8
 9      circularList: exception;                                    -- 11
10      emptyList: exception;                                       -- 12
11      itemNotPresent: exception;                                  -- 13
12      noMore: exception;                                          -- 14
13
14      procedure attach(l1: in out list; l2 in list);              -- 15
15
16      function copy(l: list) return list;                         -- 16
17
18      function create return list;                                -- 17
19
20      procedure deleteHead(l: in out list);                       -- 18
21
22      procedure deleteItem(l: in out list; e: in itemType);       -- 19
23
24      procedure deleteItems(l: in out list; e: in itemType);      -- 110
25
26      function equal(l1, l2: list) return boolean;                -- 111
27
28      function firstValue(l: list) return itemType;               -- 112
29
30      function isInList(l: list; e: itemType) return boolean;     -- 113
31
32      function isEmpty(l: list) return boolean;                   -- 114
33
34      function lastValue(l: list) return itemType;                -- 115
35
36      function length(l: list) return integer;                    -- 116
37
38      function makeIterator(l: list) return iterator;             -- 117
39
40      function more(l: iterator) return boolean;                  -- 118
41
42      procedure next(iter: in out iterator; e: itemType);         -- 119
43
44      procedure replaceHead(l: in out list; e: itemType);         -- 120
45
46      procedure replaceTail(l: in out list; newTail: in list);    -- 121
47
48      function tail(l: list) return list;                         -- 122
49
50      function last(l: list) return list;                         -- 123
51
52  end listPkg;
```