N93-12385

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.

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 "editor". 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 botto is $\perp$, the void type. These two special types ensure that any two types in the lattice have a least upper bound an 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. $Facet_{\emptyset}$ 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 value 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 th examples in section 1.2.

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

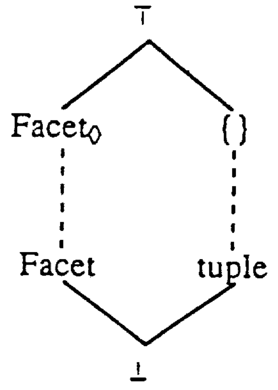

Figure 1. The reuse type lattice

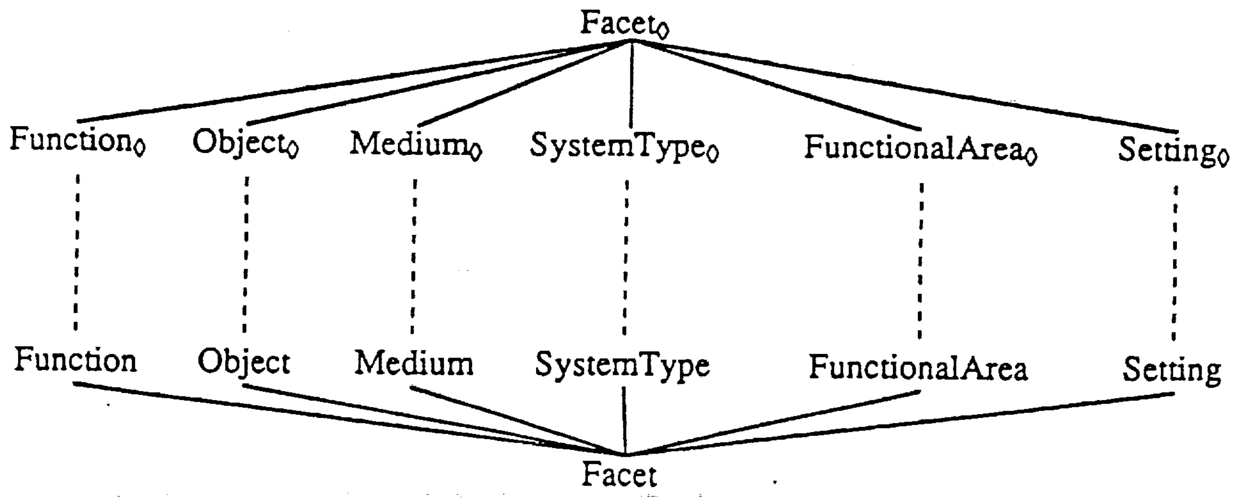

Figure 2. The Sublattice of Facet Sets

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 $\text{Facet}_{\langle\rangle}$. 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 ≤ (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, $t_{\langle a\ldots b,\, c\ldots d,\, e\rangle}$ denotes the interval that includes the subinterval a through b inclusive, the subinterval c through d inclusive, and the singleton value e. The singleton range e is equivalent to e…e. When we use such notation we intend that a ≤ b and c ≤ d, but not necessarily that b ≤ c or d ≤ e. An empty pair of brackets, $t_{\langle\rangle}$, 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

$$\frac{\begin{array}{l} A \vdash m \in t \\ A \vdash n \in t \\ A \vdash m \leq n \end{array}}{A \vdash t \preceq t_{\langle m\ldots n\rangle}} \qquad (1.1)$$

subrange of some type t; (1.2) does the same for two subranges of some type t.

$$\frac{\begin{array}{l} A \vdash m \in t \\ A \vdash m' \in t \\ A \vdash n \in t \\ A \vdash n' \in t \\ A \vdash m' \leq m \leq n \leq n' \end{array}}{A \vdash t_{\langle m'\ldots n'\rangle} \preceq t_{\langle m\ldots n\rangle}} \qquad (1.2)$$

ranges of some type t. Rule (1.3) extends subtyping to

$$\frac{\begin{array}{l} A \vdash t_{\langle m_1\ldots n_1\rangle} \preceq t_{\langle m_1'\ldots n_1'\rangle} \\ \vdots \\ A \vdash t_{\langle m_i\ldots n_i\rangle} \preceq t_{\langle m_i'\ldots n_i'\rangle} \end{array}}{A \vdash t_{\langle m_1\ldots n_1,\, \ldots,\, m_i\ldots n_i\rangle} \preceq t_{\langle m_1'\ldots n_1',\, \ldots,\, m_i'\ldots n_i'\rangle}} \qquad (1.3)$$

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

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

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

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

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).

$$\frac{\begin{array}{l} A \vdash x : t_{\langle \ldots,\, a\ldots c,\, b..d,\, \ldots \rangle} \\ A \vdash a \le b \le c \le d \end{array}}{A \vdash x : t_{\langle \ldots,\, a..d,\, \ldots \rangle}} \tag{1.7}$$

$$\frac{\begin{array}{l} A \vdash x : t_{\langle \ldots,\, a\ldots d,\, b..c,\, \ldots \rangle} \\ A \vdash t_{\langle a\ldots d \rangle} \preceq t_{\langle b..c \rangle} \end{array}}{A \vdash x : t_{\langle \ldots,\, a..d,\, \ldots \rangle}} \tag{1.8}$$

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

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

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

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

$$\frac{A \vdash x : t_{\langle \ldots,\, -\infty..\infty,\, \ldots \rangle}}{A \vdash x : t} \tag{1.12}$$

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.

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

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.

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

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

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

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

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

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.

$$\frac{\begin{array}{l} A \vdash r : \{t_1 \ldots t_n\} \\ A \vdash 1 \le i \le n \end{array}}{A \vdash r.i : t} \tag{2.2}$$

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.

$$\frac{\begin{array}{l} A \vdash T_1 : \{t_1, \ldots, t_m\} \\ A \vdash T_2 : \{t_{m+1}, \ldots, t_n\} \\ A \vdash 1 \le m < n \end{array}}{A \vdash T_1 \,\&\, T_2 = \{t_1, \ldots, t_n\}} \tag{2.3}$$

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-

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

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

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

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, "Facted 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.