N93-12390

# 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* $(1 \leq i \leq n)$, $b_i$ 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, $1 \leq m \leq n$ (we can reorder component entries as necessary). Then *R* is a *subtype* of *S*, written $R \preceq S$, if for each $i_j$, $(1 \leq j \leq m)$, $t_j \preceq t'_j$.

## 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 $\perp$, 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. $\text{Facet}_{\emptyset}$ 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 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.

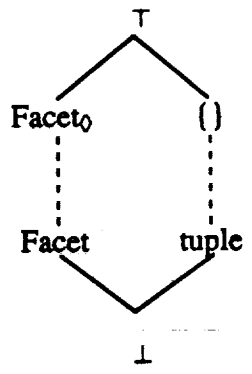

Figure 1. The reuse type lattice

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 Facet$_{\emptyset}$. The union operator generates the greatest lower bound. The intersection operator generates the least upper bound.

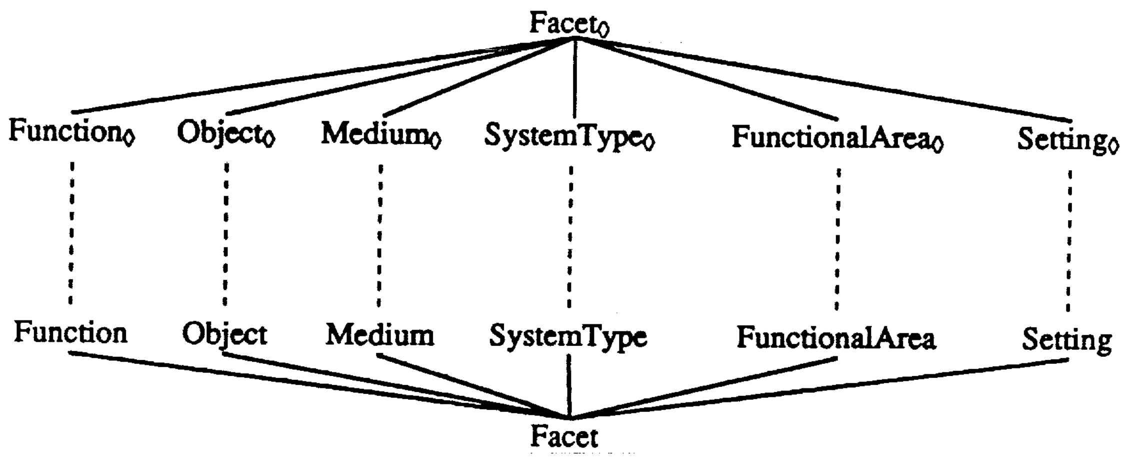

Figure 2. A Sublattice of Facet Sets

## 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-

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

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 singleton range $e$ is equivalent to $e \ldots e$. 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, $t_0$, 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, $\{i_1 : t_1, \ldots, i_n : t_n\}$. 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 between 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 repository 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.