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.

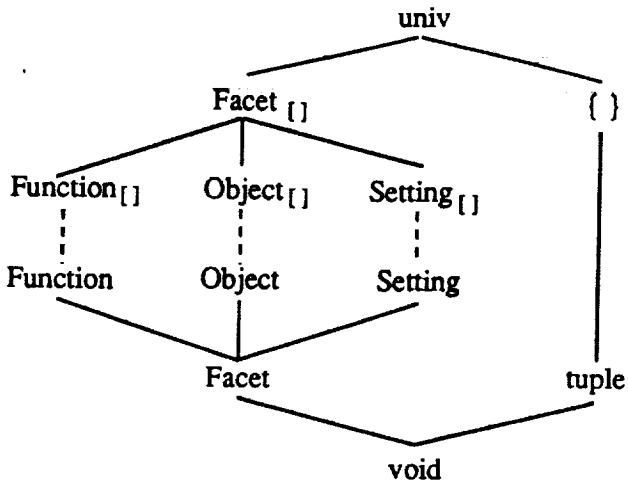

Figure 4: The Type Lattice

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.