View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by NASA Technical Reports Server

N93-12387

Selecting Reusable Components Using Algebraic Specifications *

David Eichmann

Dept. of Statistics and Computer Science
West Virginia University
Morgantown, WV 26506
email: eichmann@a.cs.wvu.wvnet.edu

1. Introduction

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 a mixed classification/axiomatic approach to this problem based upon our lattice–based faceted classification technique [6] and Guttag and Horning's algebraic specification techniques [8]. This approach selects candidates by natural language–derived classification, by their interfaces, using signatures, and by their behavior, using axioms.

We briefly outline our problem domain and related work in sections 2 and 3. Section 4 describes lattice–based faceted classification; we refer the reader to surveys of the extensive literature for algebraic specification techniques [1,3,15]. Behavioral support for reuse queries appears in section 5, followed by our conclusions.

2. The Problem

A mature software repository can contain thousands of components, each with its own specification, interface, and typically, its own *vocabulary*. Classification schemes based upon terminology used in components and/or their corresponding documentation can obviously fall prey to the ambiguity inherent in natural language. Less obvious is the ambiguity inherent in formal specifications. Consider the signatures presented in figures 1 and 2 for a stack of integers and a queue of integers, respectively. These signatures are isomorphic up to renaming, and thus exemplify the *vocabulary problem*. Software reusers implicitly

```
Create:   → Stack
Push:     Stack × Integer → Stack
Pop:      Stack → Stack
Top:      Stack → Integer
Empty:    Stack → Boolean
```

Figure 1 – Signature for the Stack Specification

```
Create:  → Queue
Enqueue: Queue × Integer → Queue
Dequeue: Queue → Queue
Front:   Queue → Integer
Empty:   Queue → Boolean
```

Figure 2 – Signature for the Queue Specification

* To appear in *AMAST'91 – Proc. of the Second Int. Conf. Algebraic Methodology and Software Technology*, Workshops in Computing Series, Springer–Verlag, London, UK, due out 1992.

```
Create:   → TOI
Insert:   TOI × Integer → TOI
Remove:   TOI → TOI
Current:  TOI → Integer
Empty:    TOI → Boolean
```

Figure 3 – Signature for the Ambiguous Specification

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. Renaming push enqueue, pop dequeue, and top front in a stack component is an example of the former. Renaming stack and queue to TOI, push and enqueue to insert, pop and dequeue to remove, and top and front to current in a stack component and a queue component, respectively, is an example of the latter. The signature for the resulting ambiguous specification appears in figure 3.

3. Related Work

Recent proposals for repository interfaces [6,5,9,11] failed to adequately address the vocabulary problem, since they concentrated on vocabulary–oriented classification techniques, e.g., from library science. Prieto–Diaz and Freeman used the notion of literary warrant to develop the faceted classification approach [11]. They clustered descriptive terms drawn from sample components in the repository into a number of facets comprising a single tuple schema. The faceted classification approach suffers from the vocabulary problem due to the probable ambiguity in the vocabulary used both in the components and the corresponding documentation. For example, consider the case where various components use terms such as destroy, delete, remove, discard, etc. – all pairwise synonyms, but with quite distinct semantics.

Eichmann and Atkins further structured the facets and facet tuples into a lattice, alleviating the requirement that all components contain a value for all facets [6]. The classification of a component contained a set of values drawn from a given facet, avoiding the need to compute closeness metrics.

Neither of the above approaches completely overcomes the true nature of the vocabulary problem, the issue of behavior. Algebraic specification techniques (e.g., [8]) partially (and unintentionally) overcome the vocabulary problem through inclusion of behavioral axioms into the specification. Figures 4 and 5 provide characterizations for figures 1 and 2, respectively (ignoring error semantics for the sake of simplicity). The main objection to algebraic specifications is in the need to *comprehend* the specifications retrieved from the

```
Pop(Push(S,I))  = S
Top(Push(S,I))  = I
Empty(S) = if (S == Create) then true else false
```

Figure 4 – Axioms for the Stack Specification

```
Dequeue(Enqueue(Q,I)) = if (Q == Create) then Create
                        else Enqueue(Dequeue(Q),I)
Front(Enqueue(Q,I)) = if (Q == Create) then I
                      else Front(Q)
Empty(Q) = if (Q == Create) then true else false
```

Figure 5 – Axioms for the Queue Specification

repository. The traditional examples in the literature rarely exceed the complexity exhibited in figures 1 and 2.

4. Faceted Classification

4.1. Single Tuple Classification

The faceted classification methodology, as studied by Prieto–Diaz, begins by using domain analysis "to derive faceted classification schemes of domain specific objects" [11,12]. This process relies on a library notion known as *literary warrant*, involving the collection of a representative sample of titles which are to be classified and extracting 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 reusable components. Using literary warrant, Prieto–Diaz identified six facets that can be used as a taxonomy: 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, 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. Problems arise 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 for continuing the search after an initial query involves *conceptual closeness*, where 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.

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.

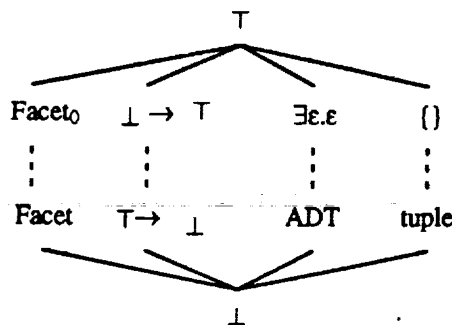

Figure 6.

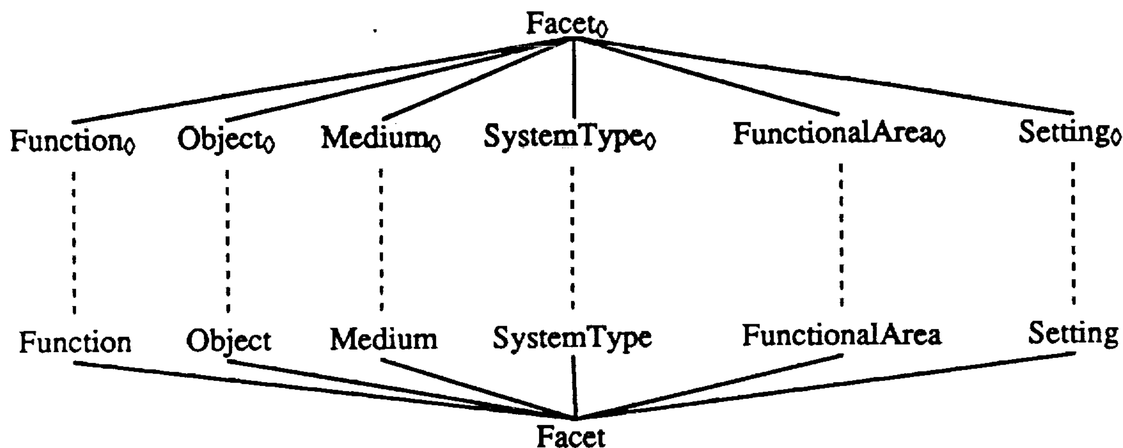

Figure 7. The Sublattice of Facet Sets

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$_\diamond$. 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 $t_{\langle a \ldots b \rangle}$. 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...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) and (2)) are assumed to be part of the assumptions, and typically are not explicitly mentioned. Rule (1) provides for subtyping a subrange of some type t; (2) does the same for two subranges of some type t. Rule (3) extends subtyping to

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

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

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

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

Not shown are rules used to combine ranges in domain intervals; two ranges in an interval that share a common endpoint combine into a single range, and overlapping ranges merge into a single 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.

4.2.3. Function Typing

Function types are useful both for characterizing programs and for characterizing the operations of ADTs. Inference rule (4) 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.

$$\frac{A, x : t' \vdash e : t}{A \vdash \lambda(x : t')\, e : (t' \rightarrow t)} \tag{4}$$

One function type, $s \rightarrow t$, is a subtype of another, $s' \rightarrow t'$, if the subtype function accepts the entire domain of the function supertype (i.e., $s' \preceq s$), and produces a range contained in the supertype range (i.e., $t \preceq t'$), as shown in inference rule (5).

$$\frac{A \vdash s' \preceq s \quad A \vdash t \preceq t'}{A \vdash s \rightarrow t \preceq s' \rightarrow t'} \tag{5}$$

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.

4.2.4. ADT Typing

Inference rules (6) and (7) define type inference for existential types [1]. 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.

$$\frac{A \vdash e_1 : s_1|_{t/a} \quad \vdots \quad A \vdash e_n : s_n|_{t/a}}{A \vdash \text{pack}\,(a = t \text{ in } (j_1 : s_1, \ldots, j_n : s_n))\;(e_1, \ldots, e_n) : \exists a.(j_1 : s_1, \ldots, j_n : s_n)} \tag{6}$$

$$\frac{A \vdash e : \exists b.(j_1 : s_1, \ldots, j_n : s_n) \quad A.(x : (j_1 : s_1, \ldots, j_n : s_n))|_{a/b} \vdash e' : t}{A \vdash \text{open } e \text{ as } x[a] \text{ in } e' : t} \tag{7}$$

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 (8) characterizes subtyping of two instances of abstract types.

$$\frac{A.(t_1 \preceq t_2) \vdash (t \preceq t')}{A \vdash (\exists(t_1 \preceq t_2).t) \preceq (\exists(t_1 \preceq t_2).t')} \tag{8}$$

Note that in addition to providing subtyping of two ADTs, rule (8) also supports subtyping of two instances of the same ADT.

For an example of the former, $\exists T' \; \exists(T \preceq T').T''$ 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 [13], 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.

4.2.5. Tuple Typing

We view a tuple r to be of type record, $\{t_1, \ldots, t_n\}$, where t_i is some facet, function, or ADT type. While components are not labeled, they may appear in any order since we assume that facet names are unique. Two record types are assumed to be equivalent if they only differ in the order of their respective components.

Inference rule (9) characterizes subtyping for tuples. Informally, one tuple type is a subtype of another if it has all of the components of the other (and possible more), and for those common components, the type

of a given component in the tuple subtype must be a subtype of that component's type in the tuple supertype.

$$\frac{\begin{array}{c} A \vdash 1 \leq m \leq n \\ A \vdash t'_1 \leq t_1 \\ \vdots \\ A \vdash t'_m \leq t_m \end{array}}{A \vdash \{t'_1, \ldots, t'_m, \ldots, t_n\} \leq \{t_1, \ldots, t_m\}} \quad (9)$$

Inference rule (10) supports definition of tuple constants and extraction of a component value, respectively.

$$\frac{\begin{array}{c} A \vdash e_1 = t_1 \\ \vdots \\ A \vdash e_n = t_n \end{array}}{A.(r = \{e_1, \ldots, e_n\}) \vdash r : \{t_1, \ldots, t_n\}} \quad (10)$$

4.2.6 Repository Structure

The repository itself consists of the actual components (we aren't concerned at this point whether they are stored in source form, or in executable form (as considered by Weide, et. al. [14])), axiomatic specifications for each of the components, and a vocabulary–based classification structure.

The components are partitioned by structural similarity (package, function, etc.). Each partition is associated with a set of facets which characterize and classify all 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 4.1 is a good example of a facet likely to be shared by a majority of partitions in the repository.

Each partition instance (i.e., each component) has one or more lattice vertices that correspond to the sets of section 4.2.1. 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.

5. Behavior Specifications In Reuse

We base repository retrieval interface upon both the vocabulary used in components and the *observable behavior* of components, that is, the axioms that formally characterize the semantics of components. The ability for a reuser to partially specify component behavior is a key element of the interface design.

Retrieval of components under this system proceeds in two phases. A reuser initially specifies a vocabulary/signature query, narrowing the field of candidates to those that are isomorphic to the query signature (if one is specified). The axioms characterizing each of the candidate components in turn are then used as theories supporting attempted proofs of the proposition(s) the reuser poses in the second phase of the query (using an existing theorem prover, e.g., RRL [10]). Successful proof of all of the propositions posed by the user indicates that the component of interest provides at least the semantics sought after.

This by no means implies that the components thus retrieved have the same semantics. For example, the query proposition

```
Create:  → Stack
Push:    Stack × Integer → Stack
Pop:     Stack → Stack
Top:     Stack → Integer
Empty:   Stack → Boolean
Depth:   Stack → Integer
```

Figure 8 – Signature for the Stack Specification

```
Remove(Insert(Create,x)) = Create
```

can be proven both by the stack axioms (with Remove bound to Pop and Insert bound to Push) and by the queue axioms (with Remove bound to Dequeue and Insert bound to Enqueue). However, the query proposition

```
Remove(Insert(Insert(Create,x),y)) = Insert(Create,x)
```

can be proven only by the stack axioms; having failed to prove the query proposition, the queue specification would be removed as a candidate. Our assumption in this approach is that the reuser will pose propositions that best characterize the behavior of interest (i.e., the second example proposition better characterizes a stack than does the first example proposition), thereby providing better discrimination between signature–isomorphic components.

Propositions posed by reusers need to be tested against a single specification's axiom set multiple times in cases where an operation from the reuser's query signature cannot be resolved to a single operation in a candidate component's signature. This usually results from an insufficient vocabulary framework. Consider the signature of figure 8, a slightly extended version of figure 1. In the absence of any classification information specifically concerning the Top and Depth operations for query propositions such as

```
Query(Insert(Insert(Create,x),y)) = y
Query(Insert(Insert(Create,x),y)) = 2
```

(assuming that these two propositions are posed in separate queries) the system must attempt a proof of the proposition using both a binding of Query to Top, Insert to Push, and Create to Create, and a binding of Query to Depth, Insert to Push, and Create to Create. The first proposition is successfully proved using the first binding and the second proposition by the second binding.

We associate an *operation partition* with each distinct operator type signature, e.g., "→ TOI" or "TOI × Integer → TOI", in a specification. A given operation is a member of an operation partition if it has the type associated with that partition. The number of alternative bindings, AB, for a given query/candidate pairing derives from the cardinalities of each of COP, the set of distinct operation partitions in the candidate signature, and QOP, the set of distinct operation partitions in the query signature.

$$AB = \prod_{a \in COP} \begin{cases} \prod_{b \in QOP} |a| * |b| & \text{if } b \simeq a \text{ exists} \\ 1 & \text{otherwise} \end{cases} \quad (11)$$

Two operations, a and b, drawn from the candidate signature and the query signature, respectively, are *partition equivalent*, written $a \simeq b$, if the types associated with the partitions differ only in TOI, the type of interest. In the absence of any other information, any member of a given query operation partition must be bound to all members of the corresponding candidate operation partition for proving user propositions, par-

```
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.