1992

A Tool for Reusable Software Component Retrieval via Normalized Specifications

Steigerwald, Robert

IEEE

http://hdl.handle.net/10945/43619

A Tool for Reusable Software Component Retrieval via Normalized Specifications

Robert Steigerwald, Luqi, Valdis Berzins

Computer Science Department
Naval Postgraduate School
Monterey, CA 93943

Abstract

This paper introduces the concept of reusable software component retrieval using normalized formal specifications. Reusable Ada software components are stored in a software base that supports a rapid prototyping system called CAPS (Computer Aided Prototyping System). Each component in the software base has a corresponding formal specification. A query in the form of a formal specification is used to search for candidate components that will satisfy the requirements of the query. The specification languages used are the Prototype System Description Language (PSDL) and OBJ3. Each specification is normalized to facilitate component retrieval. This paper describes the software base model, syntactic and semantic normalization, and the component retrieval mechanisms.

1: Introduction

The rapidly growing demand for software has shifted toward larger and more complex systems. The inadequacy of current software development methods is evident in high software costs and low programmer productivity. Software engineers need software tools that will help them better manage the complexity of these systems. Rapid prototyping has become an accepted software development method to rapidly construct and adapt software, validate and refine requirements, and check the consistency of proposed designs. Using rapid prototyping and design tools, we have experimented with a software development technique to increase productivity, improve software quality and reliability, and provide savings in both time and money for software development. As a component of a rapid prototyping system, the tool described here aids in storing and retrieving reusable software components from a software base.

This paper focuses on a technique for using specifications as search keys for component retrieval. Given our software base of Ada components, each with a corresponding formal specification, and given a software base query in the form of a specification, we would like to search the database to find the component(s) whose specification(s) best match the query specification. Fundamental to our approach is normalization of specifications. Using normal forms for the specifications reduces the variability in the representation and diminishes the effort required for the search.

Section 2 describes related work, rapid prototyping, CAPS, the form of our component specifications, and our process model for component storage and retrieval. In Section 3 we describe syntactic normalization and matching using PSDL specifications and in Section 4 we discuss semantic normalization and matching. We summarize in Section 5 and assess the progress of our system.

2: Background

2.1: Related work

Runciman and Toyn have developed a method of retrieving software components by polymorphic type [14]. A two phased approach to retrieving components via specifications developed at CMU [15] retrieves ML components (functions) with Lambda Prolog specifications by first matching on signature and then on function pre- and post-conditions. Another approach employing both syntax and semantics of a component but not necessarily its specification was developed by Wood and Sommerville [21]. They built a system that performs component retrieval using descriptor frames based on Schank's theory of conceptual dependency.

2.2: Rapid prototyping

A prototype is an executable model of a proposed software system that accurately reflects chosen aspects of the system, such as display formats, the values computed, or response times. Rapid prototyping is an iterative approach to software development that uses prototypes to help both the developers and their customers visualize the proposed system and predict its properties.

0073-1129-1/92 $3.00

Rapid prototyping may be used in conjunction with or as an alternative to the traditional software lifecycle. It may be used to rapidly construct and adapt software, validate and refine user requirements, or check the consistency of proposed designs. Our approach to rapid prototyping combines the power of high level specifications with a data base of reusable software components to help an engineer quickly build a prototype which will help clarify requirements and eliminate the large amount of wasted effort currently spent on developing software to meet incorrect or inappropriate specifications [13, 19].

Prototyping has gained importance in recent years because new technologies have made computer-aided prototyping feasible. These technologies have reduced the time and cost involved in producing a prototype, thus widening the gap between a software prototype and the cost of the final software system and increasing the potential leverage of prototyping. The new technologies, often manifested in CASE tools, are based on reusable code, computer-aided design, and automatic generation of programs.

2.3: The Computer Aided Prototyping System

The computer aided prototyping system (CAPS) is an integrated environment aimed at rapidly prototyping hard real-time embedded systems [11, 13]. This integrated set of software tools includes an execution support system, a rewrite system, a syntax directed editor with graphics capabilities, a software base, a design database, and a design management system.

Embodied within the CAPS software development approach is a systematic design method for rapid prototype construction. System or subsystem descriptions are stated at a problem-oriented, abstract level and iteratively refined into a hierarchically structured prototype using a uniform decomposition method that combines the advantages of data flow and control flow. At each level of the hierarchy, the designer focuses only on the details important at that level.

With respect to reusable component retrieval, the most important tool in CAPS is the software base management system (SBMS). As this paper describes in detail, the key to component storage and retrieval is the component's specification.

2.4: Component specification

The prototype system description language (PSDL) [12] forms the basis of CAPS. It serves as an executable prototyping language at a specification or design level and has special features for real-time system design. The PSDL model is based on data flow under real-time constraints and uses an enhanced data flow diagram that includes non-procedural control constraints and timing constraints.

PSDL provides two kinds of building blocks for prototypes: abstract data types and operators. Software systems are modeled as networks of operators communicating via data streams. The following is an example of a PSDL specification for an abstract data type component that implements a set and some of its operations.

```
type SET specification
    operator EMPTY specification
        output S1 : set end

    operator ADD specification
        input   ELEMENT : integer
                S1 : set
        output  S2 : set end

    operator IN specification
        input   ELEMENT : integer
                S1 : set
        output  RESULT : boolean end

    operator SUBSET specification
        input   S1 : set
                S2 : set
        output RESULT : boolean end

    operator EQUAL specification
        input   S1 : set
                S2 : set
        output  RESULT : boolean end

    keywords SET, INTEGER
    description {Implements a set of integers}
    axioms
    {obj SET is sort Set .
        protecting INT .
        op empty : -> Set .
        op add : Int Set -> Set .
        op in : Int Set -> Bool .
        op subset : Set Set -> Bool .
        op equal : Set Set -> Bool .
        vars s1 s2 : Set .
        vars e1 e2 : Int .
        eq in(e1, empty)  = false .
        eq in(e1, add(e2, s1)) = or(==(e1, e2), in(e1, s1)) .
        eq subset(empty, s1) = true .
        eq subset(add(e1, s1), s2) = and(in(e1, s2),
                                         subset(s1, s2)) .
        eq equal(s1, s2) = and(subset(s1, s2),
                               subset(s2, s1)) .
    endo}
end
```

The set package defines constructors (Empty, Add) and accessors (In, Subset, Equal) for a set of integers. Each operator description includes a specification which may optionally include inputs, outputs, exceptions, generic parameters, states and timing information. It is these *interface* characteristics that form the basis of *syntactic* normalization and matching, the first phase of the retrieval process.

One of the latter parts of a PSDL component

specification is the formal description of the component or *axioms*. PSDL uses axioms of several different forms. The axioms in this paper are written using OBJ3 [6]. The axioms express the *semantics* of the specification and will be the basis of semantic normalization and matching, the second phase of the retrieval process. Syntactic and semantic normalization and matching together provide the means for component storage and retrieval.

2.5: Process model for component storage and retrieval

Today there is much attention focussed on the nature of reusable software component databases. The most widely known Ada software bases are the Common Ada Missile Parts (CAMP)[22], the Ada Software Repository [3], and the Booch component collection [2]. There are many more besides these and all of their developers have given thought as to how to retrieve a desired component from the software base. Techniques that have been applied to the problem of component retrieval include browsers such as those found in object-oriented languages (e.g. Smalltalk, KEE and Eiffel), keyword search algorithms, multi-attribute search algorithms, and expert systems [18].

Our general methodology is to store components in an OODBMS and use PSDL specifications as the basis for retrieval. Each stored component consists of a PSDL specification, an Ada specification, and an Ada body. The syntax and semantics of the PSDL specification is used to direct the search for a component.

Figures 1 and 2 summarize the steps necessary to store components in the software base and to retrieve them using a given query specification. Components to be *stored* must first pass through syntactic and semantic normalization (see Figure 1). The normalization processes transform the component's PSDL specification to facilitate later matching.

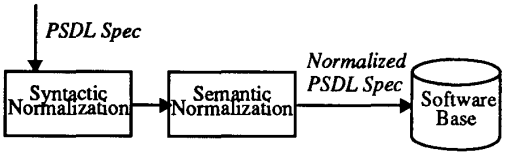

Fig. 1 Component Storage

Figure 2 shows the general process for component retrieval. A *query* for a library component is a PSDL specification. The query is syntactically and semantically normalized and then matched against stored specifications. Syntactic and semantic normalization may proceed in parallel but syntactic matching must take place before semantic matching. Syntactic matching is faster and partitions the software base quickly in order to narrow the list of possible candidates that the semantic matching algorithm must consider. Semantic matching may be time consuming and should be applied to as small a candidate list as possible without excluding potential matches.

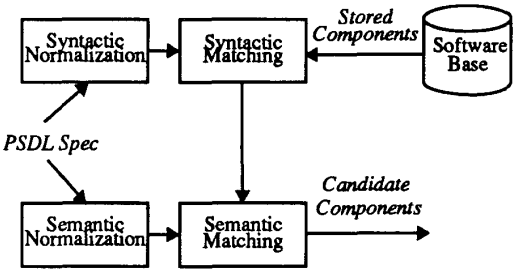

Fig. 2 Component Retrieval

Both syntactic and semantic normalization and matching are required to achieve the best performance from the system. The main benefit of syntactic matching is speed whereas the advantage of semantic matching is accuracy. We believe that accuracy is required in order to reduce the number of reusable components that a designer will have to evaluate before making a selection. Consider the example of trying to find an abstract data type for a set. The Booch component library [2] contains 34 different variations for implementing a set. The specifications for these set packages are quite similar but the implementations are different. Clearly we cannot rely on syntax alone to provide us a sufficiently fine grained search. Semantics are also required. The details of syntactic and semantic normalization and matching are addressed in sections 3 and 4.

3: Syntactic matching and normalization

The purpose of syntactic normalization is to derive information from the PSDL specification to define an ordering for stored component specifications. It is called *syntactic* normalization because the information used comes from the interface specification of the component. This part of the specification contains information on the inputs, outputs, states, and exceptions but contains no implementation details. Syntactic normalization gathers statistics from a query for use in matching.

Syntactic *matching* is the process of comparing the statistics derived from a query to those of stored components. The purpose of the matching process is to quickly eliminate those components that cannot possibly satisfy the requirement, leaving a candidate set of components for semantic matching. From an information retrieval perspective, syntactic matching provides high *recall* while semantic matching provides increased *precision*. Details of syntactic normalization and matching may be found in [16].

4: Semantic normalization and matching

As shown in the set example earlier, one of the attributes of a PSDL specification is an axiomatic description of the component. Both types of PSDL components (operators and abstract data types) may be described by algebraic axioms. It is likely, given a large software base, that a query based only on syntactic matching will find components that are not semantically relevant. We therefore perform normalization and matching on component semantics as well. The semantics of a component are described using OBJ3 [6, 20], an executable specification language.

This section describes our phased approach to specification matching and presents some of the details of OBJ3. Interface normalization and matching are then described, followed by explanations of *query by consistency* and normalization for theorem proving.

4.1: Overview

Our overall approach to reusable component retrieval is three-phased. The first phase, described above, focuses on the numbers and types of parameters within each operator in the PSDL portion of the query.

The second phase, called *query by consistency*, relies on the formal OBJ3 specification for each component. Query by consistency formulates example terms from a query's algebra and passes the terms as parameters to its operators. The set of outputs obtained is compared against the outputs from similar tests performed in the domain of a candidate component. This phase reduces further the set of candidate components, eliminating components which cannot possibly satisfy the query because of behavioral incompatibilities. Query by consistency requires a form of normalization we call interface normalization.

The final phase of the search process, based on theorem proving, attempts to find candidates that can be shown to satisfy the query, or to order the ones that partially satisfy the query if none of the candidates is completely satisfactory. This phase requires axiom normalization.

4.2: Representation of specifications

OBJ3 is the language we have chosen to augment PSDL to write our formal specifications. This section describes some of the important constructs of OBJ3. Figure 1 shows an example of an OBJ3 specification in the axioms portion of the PSDL specification.

OBJ3 is a functional programming language rigorously based on order sorted logic. The dominant construct is the module. Modules can be objects or theories. An object completely determines the behavior of a type or parameterized set of types and a theory partially constrains the behavior of a set of types. Both objects and theories are executable, but theories cannot contain built-in equations. We focus here on objects which consist of a signature and a set of axioms.

An OBJ3 definition of an abstract data type introduces a new set of values, which contains all the instances of the type. The principal sort (Order sorted logic uses the term "sort" rather than "type") of the abstract data type is the name of this set of values. The form of the signature, which defines the syntax of the object's interface, is a set of "op" definitions defining the name, domain sorts, and range sort of each operator (since OBJ3 is a functional programming language, all operators are functions). The sorts of the object defined in Figure 1 are {Set, Int, Bool}. An operation whose range is the same as the principal sort is called a *constructor*. An operation whose range is a sort other than the principal sort is called an *accessor*.

The axioms (or equations) portion of an object define the semantics of the object. Expressions are of the form

eq <Exp1> = <Exp2> *or*

cq <Exp1> = <Exp2> if <Bexp>

where both sides of each equation are well formed expressions with respect to the signature and previously declared variables. The axioms are written declaratively and interpreted operationally as rewrite rules.

Objects may import operations and sorts from other objects using the *protecting* statement. In the object defined in Figure 1, we import another object INT, which affords us the ability to use the operations defined on integers.

In our approach to semantic matching, the OBJ3 portion of the PSDL query is compared to OBJ3 specifications of stored components to identify components that can possibly satisfy the query. Because of the infinite variety possible in writing specifications, normal forms become an important means to diminish the effort applied to finding a match.

4.3: Interface normalization

The signature of an OBJ3 specification is an interface description. One of the first tasks required in searching for candidate components is to find a correspondence or mapping between the query and a stored component by comparing their interfaces. In order to simplify the mapping process, we normalize the interface, transforming it to a suitable representation for performing the mapping. This kind of normalization involves expansion and transformations.

Expansion in normalization was developed in the context of the Algebraic Specification Formalism (ASF) [1]. In this approach, a normal form is achieved when all imports to a specification have been eliminated and as many parameters as possible have been eliminated. ASF's textual normalization expands a module by fully incorporating the sorts and functions of imports and by binding parameters to the greatest extent possible. The purpose of this normalization in ASF is to assign a semantics to the complete specification and to each module within the specification. ASF also performs a renaming of operators

with the same name to avoid conflicts.

In the process of normalizing an OBJ3 interface description, we also expand the module. The expansion is necessary because the module will be considered an *atomic* unit during the matching process. Contrary to ASF however, we allow overloading of operator names. A detailed example of our expansion method may be found in [17].

Having performed expansion, the system constructs an alternative representation of the signature to simplify mapping. Since we use Prolog as the tool to find the mappings between a query and a candidate component, we transform each operation definition in the signature into a set of Prolog predicate expressions. To guide this transformation, it is necessary to have more information about the operations than is provided in the specification. We must also know which of the operations the user wants considered in the matching process

```
obj LIST-OF-BITUPLE is sort List .
    sort BiTuple .
    protecting NAT .
    protecting BOOL .
    op nil : -> List .
    op cons : BiTuple List -> List .
    op make : Nat Nat -> BiTuple .
    op length : List -> Nat .
    op head : List -> BiTuple .
    op tail : List -> List .
    op append : List List -> List .
    op reverse : List -> List .
    op member : BiTuple List -> Bool .
    op first : BiTuple -> Nat .
    op second : BiTuple -> Nat .
    ...
endo
```

Fig. 3 Interface Description for a List of BiTuple

For example, if the specification shown in Figure 3 were used as query to the software base, the user might not want all of the operations that come with the List object. A more general query with fewer "op" definitions would certainly offer better recall from the software base. Also, the user may have defined hidden or local operations in his object which are not necessarily required by the stored component. We therefore leave it up to the user to specify the operations he wishes to have considered. A specification used for query may have only a few of the operations identified, whereas a specification accompanying a component to be stored may have all operations identified. Figure 4 shows an example of the LIST-OF-BITUPLE module used as a query and Figure 5 shows it used as part of a component to be stored.

```
***(operations nil cons make append length)
obj LIST-OF-BITUPLE is sort List .
    sort BiTuple .
    protecting NAT .
    protecting BOOL .
    ...
endo
```

Fig.4 List of BiTuple as a Query

```
***(operations nil cons tail append reverse
    make length head first second member)
obj LIST-OF-BITUPLE is sort List .
    sort BiTuple .
    protecting NAT .
    protecting BOOL .
    ...
endo
```

Fig. 5 List of BiTuple for Storage

The specifications in Figures 4 and 5 have been augmented with OBJ3 comment blocks, ***(comment), to indicate the operations the user wants considered. From this information and that contained in the signature, the necessary Prolog predicate expressions may be generated. For each operation specified in the signature we define a corresponding "operation" predicate, and for each input parameter in the operation we define an "argument" predicate.

To find a matching candidate in Prolog, we combine the predicate expressions provided by the query to form a Prolog rule. To that rule, we also add additional predicate expressions to ensure that all bound operation names are unique and that for each operation, all parameter positions are unique. We use the predicate expressions provided by a candidate component as our database and then attempt to satisfy the query. A detailed example of our use of Prolog to perform the mapping task may be found in [17].

4.4: Query by consistency

Given one or more mappings between a query and a candidate component, we use *query by consistency* to check the semantics of the query against the semantics of the stored component.

Query by consistency creates a set of terms called a *test set* from the constructors of the sorts used in the query and uses those terms to generate a list of input-output pairs called an I/O list. The input part of each pair in the I/O List is submitted to the axioms for reduction (term rewriting) and the result is stored as the output part of the pair. We

perform the reductions in both the query and the stored component and then compare corresponding outputs in the respective I/O Lists. We use this comparison to compute a score of semantic similarity and rank-order the candidates.

The idea of using a test set is borrowed from Kapur and Zhang [8,9] who developed a refinement to an inductionless induction procedure called proof by consistency [7]. In proof by consistency using test sets, a canonical algebraic theory is augmented by an axiom to be proven (a conjecture) and a new extended canonical theory is incrementally computed. Whenever a new rule is generated during the process, the rule is checked against a test set to see if it reduces any of the irreducible ground constructor terms contained in the set. If the new rule can reduce a term in the test set, then the conjecture is not a theorem.

The test set is key to this method of proof by consistency. It is a finite set of terms that describes the equivalence classes of constructor ground terms. For example, the test set for integers with successor (suc) and predecessor (pre) constructors would be {0, suc(0), suc(suc(x)), pre(0), pre(pre(y))}.

We do not adhere strictly to Kapur and Zhang's notion of a test set but ours is similar. In our system, the test set constructed for a given query consists of terms derived from all of the operations whose range sorts are defined in the module as well as some terms derived from system defined sorts. An example of query by consistency will help clarify these concepts.

Example: Consider the example of a list of bituple shown in Figure 4. If the user were to submit that specification as a query, the system would generate the following test set:

```
Nat:     0
Nat:     succ(natconst1)
List:    nil
List:    cons(!!!, listconst1)
List:    append(listconst1, listconst2)
BiTuple: make(!!!, !!!)
```

The exclamation points in some of the test set terms are *placeholders*. They represent arguments that must be filled when using the term to build an I/O list input. A placeholder will be filled with a term having the appropriate sort.

As stated previously, these terms represent the equivalence classes of all terms that can be generated from the algebra defined in the module, limited by the user selected operations (nil cons make append length) and the predefined constructors for sort Nat. These terms will be used to build input terms in the following manner. We generate an initial I/O list consisting of a template for each user selected operation. The inputs in the initial I/O list are:

```
nil
cons(!!!, !!!)
make(!!!, !!!)
append(!!!, !!!)
length(!!!)
```

We then expand the I/O list by checking each term for placeholders. If a placeholder is encountered, we delete that term and replace it with a new set of terms, each containing a substitution for the placeholder taken from the test set. Care must be taken to avoid circularities. Expansion of the above initial I/O list resulted in 68 terms. Each term is comprised solely of operations or constant constructors (OBJ3 cannot perform reductions on terms containing variables). A sample of the terms generated follows:

```
nil
cons(make(0, 0), nil)
cons(make(0, 0), append(listconst1, listconst2))
cons(make(0, succ(natconst1)), nil)
make(0, 0)
make(0, succ(natconst1))
make(succ(natconst1), 0)
make(succ(natconst1), succ(natconst1))
append(nil, nil)
append(nil, append(listconst1, listconst2))
append(append(listconst1, listconst2), nil)
append(append(listconst1,listconst2),
       append(listconst1, listconst2))
length(nil)
length(append(listconst1, listconst2))
length(cons(make(natconst1, natconst1), listconst1))
```

Having created the input half of the I/O list, we submit the terms to the axioms of the query using the OBJ3 environment to determine output results. OBJ3 uses term rewriting to reduce the inputs to a normal form, that is, a form where no further reductions are possible.

The corresponding outputs to the above list of inputs are:

```
nil
cons(make(0, 0), nil)
cons(make(0, 0), append(listconst1, listconst2))
cons(make(0, succ(natconst1)), nil)
make(0, 0)
make(0, succ(natconst1))
make(succ(natconst1), 0)
make(succ(natconst1), succ(natconst1))
nil
append(listconst1, listconst2)
append(listconst1, listconst2)
append(append(listconst1,listconst2),
       append(listconst1, listconst2))
```

0
length(append(listconst1, listconst2))
sum(1, length(listconst1))

Note that many of the outputs are identical to the inputs. This will be the case when the input term is composed solely of constructor operations having no corresponding axioms, such as:

nil *and*
cons(make(0, 0), nil).

This is also the case when the term contains constants that cannot be reduced by axioms, such as:

length(listconst1).

We now have a complete I/O list in the domain of the query and can proceed with semantic matching.

Matching: Given a complete I/O list in the domain of the query, the system can proceed to check each of the candidates whose signature maps to the query signature. For each possible mapping for a candidate, the system transforms the inputs in the query I/O list to inputs in the domain of the candidate. The transformation process changes the names of operations and the order of parameters where necessary. The inputs are submitted to the candidate's axioms for reduction resulting in a corresponding list of outputs.

The final step is to compare the list of outputs in the query domain to the list of outputs in the candidate domain. Once again, a transformation must be made, this time on the *outputs* of the query, changing them to the domain of the candidate. At this point, a meaningful comparison can be made between the query outputs and the candidate component outputs.

The method used to compare the outputs is an inductionless induction proof method provided by OBJ3 [5]. Two terms consisting of operations on operations and constants can be checked for equality by submitting them to OBJ3 as follows:

term1 == term2

OBJ3 will reduce each of the terms and make transformations on the terms based on operation attributes (such as commutativity, associativity, etc.) to try to prove their equivalence. If it can prove they are equivalent, the result is **true**, otherwise the result is **false**. We use these true and false results to find the best map for a particular candidate and to ultimately rank-order a set of candidates. We may also use a threshold value to eliminate candidates with low scores.

4.5: Theorem proving and axiom normalization

The objective of the second phase of the component retrieval process, query by consistency, is to rank order and reduce further the set of candidate components that would have to be considered in phase three. Phase three involves theorem proving, a process that is potentially open-ended, so we would like as small a set of candidates as possible to check in this phase. In this phase, we focus on the axioms of the specification. To diminish the effort applied in theorem proving, a normal form for the axioms is warranted.

The form of theorem proving we use is inductionless induction, described in [5]. Because each formal specification consists of a set of axioms, the axioms may be treated as a theory. Given a set of axioms from a query and a set of axioms from a candidate stored component, we find the set of mappings between the query and the stored component specification. We use each possible mapping to express the axioms of the query in terms of the signature of the stored component specification. We then treat the axioms of the stored component specification as a theory and try to prove that each axiom from the query is satisfied in the theory.

The chosen proof technique treats the axioms of the stored component as rewrite rules, which are used to reduce both sides of each query axiom (equation) to normal form. If both sides of the equation reduce to the same term, then the query axiom is satisfied in the theory of the stored component. This proof procedure is sound and fast, but not complete. We plan to evaluate the effectiveness of such a weak procedure via experimental benchmarks when the implementation of phase three is complete.

If all axioms in the query are satisfied in the theory of the stored component specification, then we have proven that the stored component specification semantically matches the query. If some but not all of the axioms of the query are satisfied in the theory of the stored component, then the number of query axioms that are satisfied becomes a basis for ranking partial matches.

In the context of prototyping, it is feasible to combine the results of several components that partially satisfy a query to synthesize a component that completely satisfies the query. If we can find several components such that every component provides all of the constructor operations and each accessor operation is provided by at least one of the components, then we can satisfy the query using a record containing an instance of each representation, where different components are used to realize different accessors. This is acceptable in the context of prototyping because efficiency is not an overriding concern.

If the set of axioms in the theory is canonical, the chances for success in theorem proving are improved. A canonical set of axioms is both Church-Rosser and

terminating. We therefore normalize the axioms of a theory by performing Knuth-Bendix completion on the axioms to obtain the desired properties. This normalization is done just once for each component, at the time it is added to the software base.

4.6: Issues

Transformation of signatures to Prolog predicates is necessary to map a query signature to a candidate component signature. With some combinations, many mappings will be possible, but only one might be meaningful. This complicates the task of the overall query by consistency algorithm. For each candidate component, the algorithm must check every possible mapping. In the worst case, this task is exponential based on the number of operations with identical domain and range sorts. If we allow variables in stored components, which is the case when we store generic components, the problem is exacerbated. In practice, we hope that this will be a rare problem. We defer our judgement until we have performed more tests on this portion of the system.

Query by consistency has some limits. When an I/O list input is reduced in the query and component, the result is two terms that must be compared for equality, a problem known to be undecidable in the general case [10]. The inductionless induction method we are using is sound and fast, but not complete. We plan to evaluate the effectiveness of such a weak procedure via experimental benchmarks.

Another disadvantage is that the modules need to be well defined for term rewriting. Ideally, this means that they are Church-Rosser and terminating, that is, canonical. Experiments will indicate whether this is actually necessary. If so, an automatic Knuth-Bendix completion procedure may help. Goguen has stated, however, that users nearly always write specifications "that are easily seen to be canonical, because they just define primitive recursive functions over free constructors" [5]. We hope to use this observation to our advantage.

A third disadvantage, related to the second, is that there will be unusual situations that must be dealt with, such as rewriting that does not terminate, rewriting that results in errors, and terms whose comparison for equality is time consuming. Again, further experimentation will indicate the extent to which these problems will arise.

A final issue raises the question of practicality. Use of query by consistency requires that a user write a formal specification for the object sought. This may be beyond the capabilities of some users. With little training, however, the user could generate a *signature* for the object and proffer *example* terms rather than axioms (see Figure 6).

This obviates the need for a test set. In this case, we simply find a mapping using the signature and then use the left and right hand sides of the given "axioms" as the inputs and outputs in an I/O list. We perform the same transformations on the inputs to the domain of the candidate and perform the same check for equivalence on the corresponding outputs. This variation of query by consistency is a promising alternative when one does not have or cannot write a full formal specification. Eichmann [4] has also researched this idea, combining it with a faceted classification methodology.

```
...
axioms
{obj SET is sort Set .
    protecting INT .
    op empty : -> Set .
    op add : Int Set -> Set .
    op in : Int Set -> Bool .
    op subset : Set Set -> Bool .
    op equal : Set Set -> Bool .
    eq in(1, empty) = false .
    eq in(1, add(1, empty)) = true .
    eq subset(empty, empty) = true .
    eq subset(add(1, empty),
          add(1, add(2, empty))) = true .
    eq equal(empty, empty) = true .
    eq equal(empty, add(1, empty)) = false .
endo}
```

Fig. 6 Formal Specification for a Set with *Example* Axioms

5: Conclusion

We believe that retrieval of reusable components based on their formal specifications is both useful and feasible. Manual approaches do not scale up to large software bases, because the effort to find a component tends to increase with the number of components in the software base. Informal approaches to automatic retrieval, such as keyword search, can help to mitigate the size problem somewhat, but they are also limited in scale because the precision of a query is not very good: only a small fraction of the retrieved components is usually relevant to the problem, requiring a manual search in the final phase. In contrast, formal specification enables queries to achieve very high precision.

Query by specification does require the designer to formulate a formal specification of the properties of the desired software component, and this does require some effort. However, in the context of rapid prototyping and high-precision software development, such specifications must be developed anyway for purposes of documenting the required properties of proposed designs, and to support computer-aided verification, either via proofs or via automated testing. We believe that producing the specifications early in the project, rather than as an afterthought, has a low marginal cost, and may reduce the overall effort required for development.

Since theorem proving is known to be slow, many people

have held the opinion that retrieval based on formal specifications cannot be done within practical resource limits. In this paper we outline our approach to overcome this problem, based on a layered set of techniques for reducing the size of the set of candidate components.

Our layered approach can be summarized as follows. First, we partition the software base using an indexing structure based on signatures. This ensures that components whose types are not compatible with the query are not even considered. Second, we use test cases to quickly rule out the majority of the remaining components based on behavioral considerations. This leaves us with a set of plausible components that should be relatively small. Finally, we use a limited but fast method for theorem proving to attempt to conclusively and automatically demonstrate that one of the plausible components will in fact meet all of the requirements in the theory.

Final and conclusive demonstrations of the practical feasibility of this approach depend on experimental evaluations. We have implemented syntactic matching and query by consistency but have not yet implemented the theorem proving for phase 3 or query by consistency using examples. We plan to carry out experimental evaluations of our system over the next year.

REFERENCES

[1] J.A. Bergstra, J. Heering, and P. Klint, The Algebraic Specification Formalism ASF, Addison-Wesley, New York,1989.

[2] Grady Booch, Library of Reusable Ada Components, Wizard Software, Lakewood, CO, 1990, (303) 987-1874.

[3] Richard Conn, "The Ada Software Repository and Software Reusability", *Proceedings of the Fifth Annual Joint Conference on Ada Technology and Washington Ada Symposium*, 1987, pp. 45-53, (also in [8]).

[4] David Eichmann, "Selecting Reusable Components Using Algebraic Specifications", Second International Conference on Algebraic Methodology and Software Technology, Iowa City, IA, May 22-25, 1991, pp. 37-40.

[5] J. A. Goguen, "OBJ as a Theorem Prover with Applications to Hardware Verification", SRI International Report SRI-CSL-88-4R2, August 1988.

[6] J. A. Goguen, and Timothy Winkler, "Introducing OBJ3", SRI International Report SRI-CSL-88-9, August 1988.

[7] Deepak Kapur and D. Musser, "Proof by Consistency", Artificial Intelligence, v. 31, February 1987, pp. 125-157.

[8] Deepak Kapur, and Hantao Zhang, "An Overview of Rewrite Rule Laboratory", in Rewriting Techniques and Applications, ed. by N. Dershowitz, Springer-Verlag, New York, 1989, pp. 559-562.

[9] Deepak Kapur and Hantao Zhang, "RRL: Rewrite Rule Laboratory User's Manual", Department of Computer Science, State University of New York at Albany, May 1989.

[10] Donald E. Knuth and Peter B. Bendix, "Simple Word Problems in Universal Algebras", in Computational Problems in Abstract Algebras, John Leech, ed., Pergamon Press, 1967.

[11] Luqi, and Valdis Berzins, "Rapidly Prototyping Real Time Systems", *IEEE Software*, September 1988, pp. 25-36.

[12] Luqi, Valdis Berzins, and Raymond T. Yeh, "A Prototyping Language for Real-Time Software", *IEEE Transactions on Software Engineering*, Vol. 14, No. 10, October 1988, pp. 1409-1423.

[13] Luqi, and M. A. Ketabchi, "A Computer Aided Prototyping System", *IEEE Software*, March 1988, pp. 66-72.

[14] Colin Runciman, and Ian Toyn, "Retrieving re-usable software components by polymorphic type", in Proceedings of the International Conference on Functional Programming and Computer Architecture (FPCA'89), New Orleans, 1989, pp. 166-173.

[15] Eugene J. Rollins, and Jeanette M. Wing, "Specifications as Search Keys for SW Libraries: A Case Study using Lambda Prolog", Carnegie Mellon University, CMU-CS-90-159, 26 September 90.

[16] Robert Steigerwald, Luqi, and John K. McDowell, "A CASE Tool for Reusable Software Component Storage and Retrieval in Rapid Prototyping", Information and Software Technology, November 1991.

[17] Robert Steigerwald and Valdis Berzins, "Normal Forms for Algebraic Specifications of Reusable Ada Packages", Proceedings of Tri-Ada '91, ACM Press, San Jose, California, October 21-25, 1991.

[18] Will Tracz, Software Reuse: Emerging Technology, IEEE Computer Society Press, Washington D.C., 1988.

[18] Raymond T. Yeh, "Software Engineering", *IEEE Spectrum*, November 1983, pp. 91-94.

[20] Timothy Winkler, "Introducing OBJ3's New Features", SRI International Report (preliminary version provided by the author), March 1991.

[21] Murray Wood, and Ian Sommerville, "An Information Retrieval System for Software Components", SIGIR Forum, 22, 3&4, Spring/Summer, 1988, pp. 11-28.

[22] Air Force Armament Laboratory, Contract F08635-88-C-0002, CDRL No. A009, CAMP Parts Engineering System Catalog User's Guide, McDonnell Douglas Missile Systems Company, 30 November 1989.