1991

Normal Forms for Algebraic Specifications of Reusable Ada Packages

Steigerwald, Robert; Berzins, Valdis

Monterey, California: Naval Postgraduate School.

https://hdl.handle.net/10945/38288

Normal Forms for Algebraic Specifications of Reusable Ada Packages

Robert Steigerwald
Valdis Berzins

Computer Science Department
Naval Postgraduate School
Monterey, CA 93943

Abstract. This paper introduces the concept of normal forms for algebraic specifications of Ada packages defining abstract data types. The normal form is used in the process of reusable software component retrieval via formal specifications. We review the use of algebras for the specification of abstract data types. Then, using a concrete example, we define normal forms and present the details of algorithms to automate the normalization process.

I. Introduction

A. Proposed Solution to the Component Retrieval Problem. This paper addresses the problem of using algebraic specifications as a key for reusable software component retrieval. Given a database (software base) of software components, each with a corresponding formal algebraic 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.

The representation methodology we use for specifications is a combination of the Prototype System Description Language (PSDL) [LBY88] and OBJ3 [GW88]. PSDL is an executable specification language that allows one to model the interface and timing requirements of real-time systems. OBJ3 is also an executable specification language, used here to augment the PSDL description of a package or procedure by providing formal axioms which describe the semantics of components.

Our approach to reusable component retrieval is three-phased. The first phase focuses on the numbers and types of parameters within each operator in the PSDL portion of the query. This information is used to form a search key which partitions the software base, quickly ruling out those components which cannot possibly satisfy the query because of type incompatibilities. This phase, called the *syntactic search* phase, provides a set of components to the subsequent *semantic search* phases [SLM91]. We do not discuss syntactic search or syntactic normalization in this paper.

The second phase, called *query by consistency*, relies on the formal OBJ3 specification for each component. Query by consistency formulates example terms from a stored component'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 the query. 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.*

B. Related Work. In [RT89], the authors describe a method of retrieving software components by polymorphic type. A two phased approach to retrieving components via specifications is

1991 ACM 0-89791-445-7/91/1000-0248

described in [RW90]. That system, written in Lambda Prolog, 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 may be found in [WS88], which describes a system that performs component retrieval using descriptor frames based on Schank's theory of conceptual dependency.

C. Overview. Section II describes some of the details of OBJ3, the algebraic specification language we have chosen to write formal specifications. Section III describes interface normalization and how the normal form is used in query by consistency. Section IV describes axiom normalization and how it supports the theorem proving process in axiom matching. In section V, we provide an additional example of both forms of normalization and describe how matching takes place. Section VI contains our conclusions and some suggestions for future work.

II. Representation of Specifications

OBJ3 is the language we have chosen to augment PSDL to write our formal specifications. This section provides an example of a specification and describes some of the important constructs of OBJ3. Figure 1 shows an example of an OBJ3 specification which defines an abstract data type that can be used to keep track of values bound to variables.

OBJ3 is a functional programming language rigorously based on order sorted logic [GW88, Wink91]. 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. Objects are fully executable and theories are partially executable because the theory may not contain enough constraints to fully determine the values of some of the operations. We focus here on objects which consist of a signature and a set of axioms because our retrieval mechanism requires the specifications to be fully executable.

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* 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[1]. The sorts[2] of the object defined in Figure 1 are {Env, Item, Key}, with the principal sort Env, and two *parameterized sorts* Item and Key. 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*.

```
obj ENVIRONMENT[Item Key :: TRIV] is
    sort Env .
    protecting BOOL .
    op null : -> Env .
    op default : -> Elt.Item .
    op bind : Elt.Item Elt.Key Env -> Env .
    op lookup : Elt.Key Env -> Elt.Item .
    op combine : Env Env -> Env .
    var E1 E2 : Elt.Item .
    var K1 K2 : Elt.Key .
    var Env1 Env2 : Env .
    eq lookup(K1,null) = default .
    eq lookup(K1,bind(E1, K1, Env1)) = E1 .
    cq lookup(K1,bind(E1, K2, Env1)) =
            lookup(K1,Env1) if K1 =/= K2 .
    eq combine(null, Env1) = Env1 .
    eq combine(Env1, null) = Env1 .
    cq combine(bind(E1,K1,Env1),Env2) =
            combine(Env1,bind(E1,K1,Env2))
            if lookup(K1,Env2) == default .
    cq combine(bind(E1,K1,Env1),Env2) =
            combine(Env1,Env2)
            if lookup(K1,Env2) =/= default .
endo
```

Figure 1 - OBJ3 Specification for an Abstract Data Type

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

[1] Since OBJ3 is a functional programming language, all operators are functions.

[2] Order sorted logic uses the term "sort" rather than "type".

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 Bool, which affords us the ability to use the operations *and*, *or* and *not* (among others) in Boolean expressions.

It is easy to see parallels between OBJ3 objects and Ada packages. An OBJ3 signature is analogous to an Ada package specification and the axioms to a package body. The *protecting* statement is much like an Ada *with*. In our approach to reusable software component retrieval, each Ada package in the software base has a corresponding OBJ3 specification. To find a desired component, an OBJ3 *query* is compared to 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.

III. Normalizing Interface Descriptions

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, renaming, and transformations.

A. Expansion and Renaming. Expansion and renaming in normalization was developed in the context of the Algebraic Specification Formalism (ASF) [BHK89]. 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. In the process of normalizing, the algorithm *renames* sorts and functions to avoid conflicts; establishes the origin of each sort, function and variable, creating an attribute collocated with each definition; and binds formal with actual parameters.

In the process of normalizing an OBJ3 interface description, we also expand the module and perform renaming to avoid conflicts. The expansion is necessary because the module will be considered an atomic unit during the matching process. We illustrate this concept using an interface description for a list (see Figure 2) and one for a BiTuple (see Figure 3). (Note: The three dots that appear in many of the example specifications mean that there is more to the specification than is actually being shown.)

```
obj LIST[Item :: TRIV] is sort List .
    protecting NAT .
    protecting BOOL .
    op nil : -> List .
    op cons : Item List -> List .
    op length : List -> Nat .
    op head : List -> Item .
    op tail : List -> List .
    op append : List List -> List .
    op reverse : List -> List .
    op member : Item List -> Bool .
    . . .
endo
```

Figure 2 - Interface Description for a List

```
obj BITUPLE[C1 :: TRIV, C2 :: TRIV] is
    sort BiTuple .
    op make : Elt.C1 Elt.C2 -> BiTuple .
    op first : BiTuple -> Elt.C1 .
    op second : BiTuple -> Elt.C2 .
    . . .
endo
```

Figure 3 - Interface Description for a BiTuple

Suppose one used the List defined in Figure 2 in the following way:

```
obj LIST-OF-BITUPLE is
    protecting LIST[BITUPLE[NAT,NAT]] .
    op member : Nat List -> Nat .
    . . .
endo
```

The user has defined his own object which is composed of the List object and an object called BiTuple which defines a relation of 2 elements. The user has also defined a member function which returns the second argument of a tuple in the list given the first argument. The expanded version of the object is shown in Figure 4. It was necessary to rename[3] the imported member function and to instantiate the sort *Item* in object List as BiTuple and the elements of BiTuple as Nat.

```
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 member1 : BiTuple List -> Bool .
    op first : BiTuple -> Nat .
    op second : BiTuple -> Nat .
    op member : Nat List -> Nat .
    ...
endo
```

Figure 4 - Interface Description for a List of BiTuple

B. Transformation. Having performed expansion and renaming we now define 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 know which of the operations the user wants considered.

For example, if the specification shown in Figure 4 were used as query to the software base, the user may 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 he does not intend for the stored component to have. 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 5 shows an example of the LIST-OF-BITUPLE used as a query and Figure 6 shows it used as part of a component to be stored.

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

Figure 5 - List of BiTuple as a Query

```
***(operations nil cons tail append reverse
    make length head member1 first second
    member)
obj LIST-OF-BITUPLE is sort List .
    sort BiTuple .
    protecting NAT .
    protecting BOOL .
    ...
endo
```

Figure 6 - 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. The set of predicates for the specification in Figure 5 is:

```
operation(Sort1, 0, Op1Name)
operation(Sort1, 2, Op2Name)
```

[3] In OBJ3, as in Ada, the overloading of an operator name *is* permitted, however, to simplify matching in our system, we prefer to rename.

```
argument(Op2Name, Sort2, Op2Pos1)
argument(Op2Name, Sort1, Op2Pos2)
operation(Sort2, 2, Op3Name)
argument(Op3Name, nat, Op3Pos1)
argument(Op3Name, nat, Op3Pos2)
operation(nat, 2, Op4Name, 2)
argument(Op4Name, nat, Op4Pos1)
argument(Op4Name, Sort1, Op4Pos2)
operation(nat, 1, Op5Name)
argument(Op5Name, Sort1,Op5Pos1)
```

Each *operation* predicate expression has 3 arguments: a variable to bind to the range sort of a stored component's operation, the number of domain (input) parameters in the operation, and a variable to bind to the name of a stored component's operation. Each *argument* predicate expression has 3 arguments: a variable bound to an operation name, the sort of this particular parameter, which may be a constant or a variable, and the position of the parameter in the domain of the operation. The example predicates above contain many variables because the specification in Figure 5 is meant to be a query and we want our query to bind to the operation names and sorts of a stored component.

The choice of the arguments in the predicate expressions reflect some of our assumptions about what constitutes a match between specifications. For instance, the number of parameters present in the operations must match precisely even though we can conceive of possibilities where an operation with two variable parameters, for example, could match to an operation with two variable parameters and a constant parameter. A rule we use in finding a match is that all of the operations of the query must bind to unique operation in the component. This is based on the assumption that an engineer will not define identical semantics for any two operations in the same specification.

The order of the arguments in the predicate expressions is important for efficiency. Quintus Prolog© hashes on the first argument of a predicate expression when that argument is bound. Using the range sort of an operation as the first argument of the operation predicate partitions the operations into smaller sets. Once a particular range sort variable has been bound, the search for subsequent matches will be very fast. The first argument of the argument predicate is the name of the operation because this variable is always bound in the operation predicate that precedes it. Thus, the search for appropriate arguments is also fast.

The set of predicate expressions for the specification in Figure 6 is:

```
operation( list, 0, nil)
operation( list, 2, cons)
argument(cons, bituple, 1)
argument(cons, list, 2)
operation( bituple, 2, make)
argument(make, nat, 1)
argument(make, nat, 2)
operation(nat, 1, length)
argument(length, list, 1)
operation(bituple,1, head)
argument(head, list, 1)
operation(list, 1, tail)
argument(tail, list, 1)
operation(list, 2, append)
argument(append, list, 1)
argument(append, list, 2)
operation(list, 1, reverse)
argument(reverse, list, 1)
operation(bool, 2, member1)
argument(member1, bituple, 1)
argument(member1, list, 2)
operation(nat, 1, first)
argument(first, bituple, 1)
operation(nat, 1, second)
argument(second, bituple, 1)
operation(nat, 2, member)
argument(member, nat, 1)
argument(member, list, 2)
```

C. Using Interface Normalization for Matching

Expansion and renaming are required to make a component an atomic unit for both storage in the software base and for comparison with the query by consistency algorithm. Operation definition transformation to Prolog predicates is necessary to give us the means to map a query to a candidate stored component using Prolog. 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 predicate expressions to ensure that all bound operation names are unique and that for each operation, all parameter positions are unique. We use use the predicate expressions provided by a candidate component as our database and then attempt to satisfy the query.

We have written the program to map component signatures given correct Prolog predicate expressions. In the above example, the query in Figure 5 can map only one way to the component of Figure 6. It should be clear, however, that 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 the this portion of the system has been completed.

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 for the sorts being defined and uses the 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 pairs in the respective I/O Lists. We use this comparison to compute a score of semantic similarity and then eliminate components whose score does not exceed a certain threshold. This portion of our system is not yet implemented.

IV. Normalization for Theorem Proving

The objective of the second phase of the component retrieval process, query by consistency, is to 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 [Gogu88]. 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 from axioms from a candidate stored component, we find the set 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 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.

V. Example

To illustrate our normalization techniques, we offer an additional example based on the specification in Figure 1. Figure 7 shows the specification of Figure 1 with the generic parameters instantiated to Nat and augmented with a normalized interface description. Note that we are treating this specification as a query to the software base. Figure 8 shows a candidate stored component specification with two generic parameters, also containing a normalized interface description.

```
***(operations null bind default export
    lookup combine)
***(predicates
    operation(Sort1, 0, Op1Name)
    operation(Sort1, 3, Op2Name)
    argument(Op2Name, nat, Op2Pos1)
    argument(Op2Name, nat, Op2Pos2)
    argument(Op2Name, Sort1, Op2Pos3)
    operation(nat, 0, Op3Name)
    operation(nat, 2, Op4Name)
    argument(Op4Name, nat, Op4Pos1)
    argument(Op4Name, Sort1, Op4Pos2)
    operation(Sort1, 2, Op5Name)
    argument(Op5Name, Sort1, Op5Pos1)
    argument(Op5Name, Sort1, Op5Pos2))
obj ENVIRONMENT is sort Env .
    protecting BOOL .
    protecting NAT .
    op null : -> Env .
    op default : -> Nat .
    op bind : Nat Nat Env -> Env .
    op lookup : Nat Env -> Nat .
    op combine : Env Env -> Env .
    ...
endo
```

Figure 7 - OBJ3 Query for an Environment

```
***(operations create store exception
    retrieve join equal)
***(predicates
        operation(map, 0, create)
        operation(map, 3, store)
        argument(store, map, 1)
        argument(store, _, 2)
        argument(store, _, 3)
        operation(_, 0, exception)
        operation(_, 2, retrieve)
        argument(retrieve, _, 1)
        argument(retrieve, map, 2)
        operation(map, 2, join)
        argument(join, map, 1)
        argument(join, map, 2)
        operation(bool, 2, equal)
        argument(equal, map, 1)
        argument(equal, map, 2))
obj MAP[In Out :: TRIV] is sort Map .
    protecting BOOL .
    op create : -> Map .
    op store : Map Elt.In Elt.Out -> Map .
    op retrieve : Elt.In Map -> Elt.Out .
    op join : Map Map -> Map .
    op equal: Map Map -> Bool .
    op remove: Elt.In Map -> Map .
    op exception : -> Elt.Out .
    var D1 D2 : Elt.In .
    var R1 R2 : Elt.Out .
    var Map1 Map2 : Map .
    eq retrieve(D1,create) = exception .
    eq retrieve(D1,store(Map1, D1, R1)) = R1 .
    cq retrieve(D1,store(Map1, D2, R1)) =
       retrieve(D1,Map1) if D1 =/= D2 .
    eq join(create, Map1) = Map1 .
    eq join(Map1, create) = Map1 .
    eq join(store(Map1, D1, R1), Map2) =
       if (retrieve(D1, Map2) == exception)
       then join(Map1, store(Map2, D1, R1))
       else join(Map1, Map2) fi .
    eq equal(create, create) = true .
    eq  equal(create,store(Map1,D1,R1))=false.
    eq  equal(store(Map1,D1,R1),create)=false.
    eq equal(store(Map1, D1, R1), Map2) =
       retrieve(D1, Map2) == R1 and
       equal(Map1, remove(D1, Map2)) .
    eq remove(D1, create) = create .
    eq remove(D1,store(Map1,D1,R1)) = Map1 .
    cq remove(D1, store(Map1, D2, R2)) =
       store(remove(D1, Map1), D2, R2)
           if D1 =/= D2 .
endo
```

Figure 8 - OBJ3 Specification for a Map

Figure 9 shows the axioms of Figure 8 after axiom normalization[4].

```
obj MAP[In Out :: TRIV] is sort Map .
   op f1 : -> Bool .
   ...
   eq retrieve(D1,create) = exception .
   eq retrieve(D1,store(Map1, D1, R1)) = R1 .
   cq retrieve(D1,store(Map1, D2, R1)) =
        retrieve(D1,Map1) if D1 =/= D2 .
   eq join(create, Map1) = Map1 .
   eq join(Map1, create) = Map1 .
   eq join(store(Map1, D1, R1), Map2) =
        if (retrieve(D1, Map2) == exception)
        then join(Map1, store(Map2, D1, R1))
        else join(Map1, Map2) fi .
   eq equal(create, create) = true .
   eq equal(create, store(Map1,D1,R1)) = false .
   eq equal(store(Map1,D1,R1), create) = false .
   eq equal(store(Map1, D1, R1), Map2) =
        retrieve(D1, Map2) == R1 and
        equal(Map1, remove(D1, Map2)) .
   eq remove(D1, create) = create .
   eq remove(D1, store(Map1, D1, R1)) = Map1 .
   cq remove(D1, store(Map1, D2, R2)) =
        store(remove(D1, Map1), D2, R2)
     if D1 =/= D2 .
   cq store(remove(D1, Map1), D2, R2) =
        remove(D1, store(Map1, D2, R2))
        if D2 =/= D1 .
   eq retrieve(D1, Map2) == R1 and
        equal(Map1, remove(D1, Map2)) = f1 .
   eq equal(store(Map1, D1, R1), Map2) = f1 .
   eq f1 and retrieve(D1, Map2) == R1 = f1 .
   eq f1 and equal(Map1, remove(D1, Map2))
        = f1 .
endo
```

Figure 9 - OBJ3 Specification for a Map with Normalized Axioms

VI. 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 are currently in the process of implementing the techniques

[4] We used the automatic Knuth-Bendix completion feature of the Rewrite Rule Laboratory [KZ89] to normalize the axioms.

described in this paper, and plan to carry out such experimental evaluations within the next year.

REFERENCES

[BHK89] J.A. Bergstra, J. Heering, and P. Klint, The Algebraic Specification Formalism ASF, Addison-Wesley, New York,1989.

[Gogu88] J. A. Goguen, "OBJ as a Theorem Prover with Applications to Hardware Verification", SRI International Report SRI-CSL-88-4R2, August 1988.

[GW88] J. A. Goguen, and Timothy Winkler, "Introducing OBJ3", SRI International Report SRI-CSL-88-9, August 1988.

[KZ89] Kapur, D., and Zhang, H., "RRL: Rewrite Rule Laboratory User's Manual", Department of Computer Science, State University of New York at Albany, May 1989.

[LBY88] Luqi, Valdis Berzins, and Raymond T. Yeh, "A Prototyping Language for Real-Time Software", IEEE Transactions on Software Engineering, Vol. 14, No. 10, Oct 1988, pp. 1409-1423.

[RT89] 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.

[RW90] 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 Sep 90.

[SLM91] Robert A. Steigerwald, Luqi, and John K. McDowell, "A CASE Tool for Reusable Component Storage and Retrieval in Rapid Prototyping", in Proceedings of the Third International Conference on Software Engineering and Knowledge Engineering, Skokie, IL, June 1991.

[Wink91] Timothy Winkler, "Introducing OBJ3's New Features", SRI International Report (preliminary version provided by the author), March 1991.

[WS88] Murray Wood, and Ian Sommerville, "An Information Retrieval System for Software Components", *SIGIR Forum*, 22, 3&4, Spring/Summer, 1988, pp. 11-28.

BIOGRAPHIES

Robert A. Steigerwald is a PhD student at the Naval Postgraduate School, Monterey, California, and is currently performing research in the area of reusable software component retrieval. He is sponsored by the U.S. Air Force Academy and the Air Force Institute of Technology. Upon completion of his degree program, he will join the faculty at the Air Force Academy.

Steigerwald received his bachelor's degree in computer science in 1981 from the U.S. Air Force Academy, Colorado Springs, Colorado and his master's degree in computer science in 1985 from the University of Illinois, Urbana, Illinois.

Dr. Valdis Berzins is an Associate Professor of Computer Science at the Naval Postgraduate School. His research interests include software engineering and computer aided design. His recent work includes papers on software merging, specification languages, VLSI design, and engineering databases. He received B.S., M.S., E.E., and Ph.D. degrees from MIT, served as an Assistant Professor at the University of Texas, and as an Associate Professor the University of Minnesota. He has developed a number of specification languages and software tools. His current address is NPS 052, Monterey, CA 93943.