Deduction-Based Software Component Retrieval

B. Fischer and **M. Kievernagel** and **G. Snelting**
TU Braunschweig, Abteilung für Softwaretechnologie
Gaußstraße 17, D-38092 Braunschweig, Germany
{fisch,mkiever,snelting}@ips.cs.tu-bs.de

Abstract

We present a retrieval approach which allows pre- and postconditions of software components to be used as search keys. A component qualifies, if it has a weaker precondition and a stronger postcondition than the search key. In contrast to previous work, our tool NORA/HAMMR allows for configurable chains of deduction-based filters such as signature matchers, model checkers — which will be our main subject here —, and resolution provers; the latter can be run with dynamically adjusted axiom sets and inference rules. Hence, instead of feeding the search key and all components' specifications to a theorem prover in a batch-like fashion, NORA/HAMMR allows for incremental narrowing of the search space along the filter chain, and interactive inspection of intermediate results.

Classification: software component retrieval, formal methods, specification matching, model finding, theorem proving.

1 Introduction

The basic idea of deduction-based software component retrieval is very simple:

1. for each component C in the library, provide a formal specification in form of pre- and postcondition $(pre_C, post_C)$,
2. allow pre- and postconditions $(pre, post)$ as search keys,
3. a component qualifies, if $pre \Rightarrow pre_C \wedge post_C \Rightarrow post$.

This approach has been proposed several times (e.g. [Rollins and Wing, 1991],[Manhart and Meggendorfer, 1991]), but without convincing success. First, some people state that formal specifications are too difficult to use as search keys for ordinary programmers. Furthermore, the approach turned out to produce proof obligations which sometimes cannot be handled even with todays most sophisticated theorem provers. Technically, both weaknesses stem from a batch-oriented view of software component retrieval: in previous approaches, a complete specification must be supplied, which will be matched against all components (this includes proving the above two obligations); finally, the results are presented to the user.

In order to overcome acceptance problems and insufficient proving power, we propose a more incremental and interactive retrieval approach. Instead of feeding the complete search key into the retrieval system at once, the user is allowed to incrementally sharpen the postcondition (and weaken the precondition). Furthermore, search keys are not processed by an all-purpose theorem prover, but by a chain of filters of increasing power.

The successive filtering of components offers two main advantages. It allows free combination of different retrieval methods — including text-based or concept-based methods [Lindig, 1995]. Moreover, since intermediate results can be inspected at every stage, the overall running time is not critical to the performance of the tool. As we will show, results of acceptable precision are ready for inspection early in the process.
A typical filter chain consists of the following phases:

1. signature matching,
2. model checking,
3. theorem prover.

After signature matching (which aims at high recall and not at high precision) a lot of components has still survived, as the signature alone does not describe the component precisely enough. The second step checks the proof obligations in some small model (small integers and short lists), which is already a rather sharp filter. Only for the few remaining candidates, a theorem prover (OTTER[McCune, 1994b] or SETHEO[Letz *et al.*, 1992]) is invoked; in order to reduce the search space, NORA/HAMMR tries to select a minimal set of axioms.

In this paper, we describe some details of our approach, especially the application of the model finder `anldp` in our model checking filter. We conclude with our experiences with NORA/HAMMR in making first experiments. Our test library [Lins, 1989] consists of about 50 Modula-2-modules implementing several variants of abstract data types like stacks, queues, graphs, and trees using generic items. It provides approx. 1000 procedures with 120 different type signatures. A substantial part of these procedures has been specified manually in VDM.

2 Search keys and signature matching

The search keys, through which a user mainly communicates with NORA/HAMMR, consist of a type signature and a VDM part, as the example of a push operation for stacks shows:

```
PROCEDURE x( i:I, s:S ) : S
pre    true
post   s = tl x and i = hd x
```

The type signature encapsulates all language-specific aspects like the kind of the target object (in this example `PROCEDURE`) or the names of the basic types such as `INTEGER`. For convenience, we use a syntax which is oriented at the target language. In the case of Modula-2 we have just extended procedure types by type variables (`I` and `S`) to search for a class of signatures and to abstract naming of types. The VDM part is written in VDM-SL [Dawes, 1991], but some naming conventions are applied to refer to parameters and result.

In NORA/HAMMR, signature matching acts as the first filter in the chain. Its main characteristic is an equivalence E on types. For functional languages, E typically includes axioms to handle currying, pairing, extra arguments or different argument orders [Rittri, 1990]. Our current implementation — which aims at procedural target languages — applies order-sorted AC1-unification for parameter lists in order to abstract the order of parameters. We will also add a "result currying" axiom

```
PROCEDURE p( x, VAR y:Y )
    = PROCEDURE p( x, y:Y ) : Y
```

to handle the equivalence of VAR-parameters and return values. Thus, NORA/HAMMR matches the desired procedure even if it is implemented as

```
PROCEDURE StackSBMI.push
      ( VAR st:STACK; it:ITEM )
```

due to an application of the commutativity and result currying axioms. As expected, the intermediate result, a set of components with suitable type, is of poor precision. It not only contains `StackSBMI.push` but 87 more procedures of Lins' library. Precision drops even further if the library includes e. g. some mathematical routines since `I` and `S` may be bound to `REAL` and thus the key matches all binary operators.

3 Checking the proof obligations

We decided not to hard-wire a special proof procedure for VDM but to integrate the general purpose theorem prover OTTER[McCune, 1994b] and the associated model finder `anldp`[McCune, 1994a]. This design eases experimentation with the prover and also allows us to replace it, either by a more advanced one or even by a specially tailored proof procedure.

The second filter purges obligations which can easily be refuted by checking their validity in a small fragment of the VDM-axiomatization. Its basic idea is to check whether all assignments of small integers and small lists, resp., to program variables evaluate the obligations to `true`. Obviously, this is a prerequisite for the obligations to be provable in the full theory. We will show in the next section how we use `anldp` in this filter.

The third filter tries to prove the remaining obligations using OTTER and an axiomatization of the full theory. The whole axiomatization mainly covers the first-order properties of equality, sequences and integer arithmetic; it consists of about 120 axioms and lemmata.

The search space for the prover is reduced by splitting independent parts of a problem into subproblems. This is done by transforming the formula into disjunctive normal form and combining every set of disjunctions with common variables into one subproblem.

A further reduction of the search space is achieved by axiomatizing each subproblem of an obligation independently, i.e. linking it dynamically with an appropriate set of axioms. The selection of axioms is based on the symbols used in the problem. Axioms defining the required domains are always given. For additional auxiliary symbols the definitions in elementary terms and lemmata stating relations between them are added to the axiom set.

Some parts of the axiomatization can be used to transform respective parts of a problem in a normalized form. For example, all propositions using integer ordering relations ($<, \leq, >, \geq$) can also be expressed using only '$<$'. This normalizing part of the axiom set is always applied in a preprocessing phase and never given to the prover.

Our graphical user interface (see figure 1) reflects the idea of successive filtering. Additionally, inspectors grant easy access to any intermediate results. This filter-inspector-chain may easily be customized by the user through an icon pad. The configuration displayed below corresponds to the chain of filters described in this paper. The left part of the window is used to enter the three parts of the search key while the right part displays the final retrieval results.

4 Model checking

We will now give a detailed description of the model checking filter and its use of `anldp`. To illustrate the ideas we use the first experiment from Table 1 below,

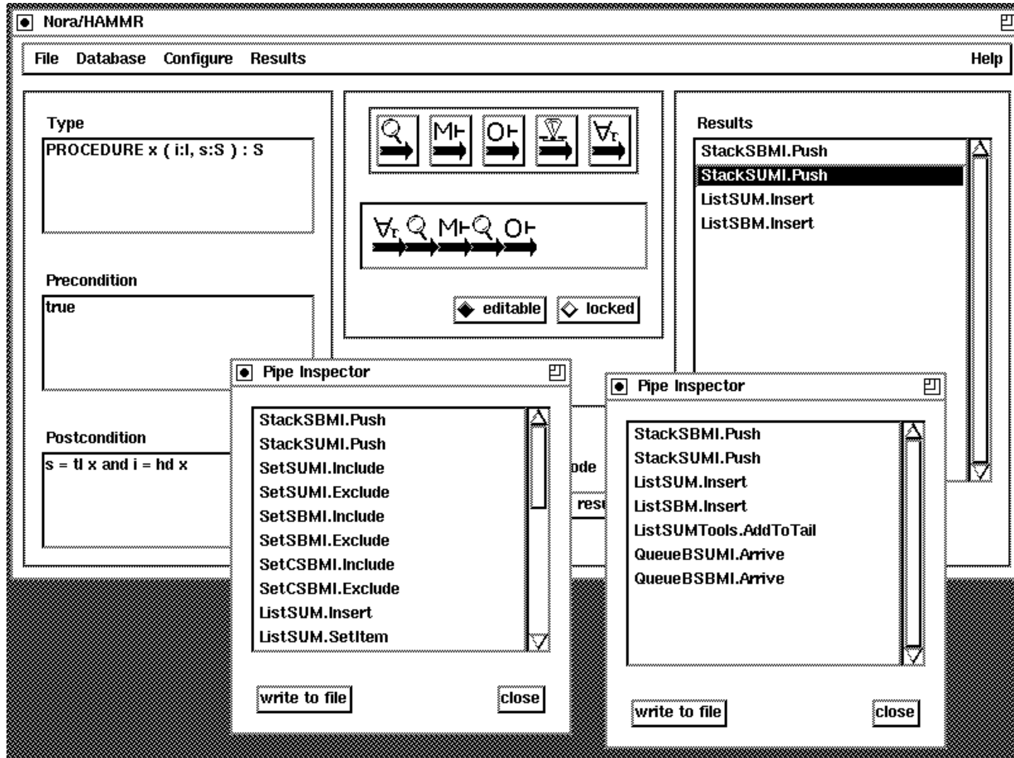

Figure 1: Graphical user interface

which is based on the example search key for the push-operation given in section 2.

First we will look at the intermediate result after signature matching. It consists of 25 procedures from 14 modules and contains the relevant procedures `Push` and `Insert` from the stack resp. singly-linked list modules. Also matched are procedures for head-assignment and tail-insertion in the singly-linked list modules, insertion and head-assignment in the doubly-linked list modules, inclusion and exclusion in the set modules and insertion and item-deletion in several kinds of queue modules.

Currently, the procedures from the doubly-linked list modules and the priority queue modules are regarded as unsuitable to build a proof obligation, because their specifications use parts of the resp. datatype which have no counterpart in the key and thus cannot be bound. This removes 6 components, leaving 19 procedures from 12 modules. Alternatively, the unbound parts could be included in proof obligations as free variables. Then a constructive proof method is required, but it would enable NORA/HAMMR to inform the user about correct instantiations of the unbound parts of a component.

For the remaining components the model checking filter checks if the resulting obligations are valid for small integers and small lists. The method we use is based on an axiom set for integers and lists which is restricted to finite domains and total functions and predicates. It defines exactly one finite model. The filter adds an obligation to the model definition and then runs the model finding program `anldp` on it to see whether it can still find the designated model. `anldp` is based on a first-order variant of the Davis-Putnam procedure (i.e. exhaustive enumeration of all finite models.) If `anldp` fails the component is rejected, because the corresponding proof obligation contradicts the model definition. Otherwise, the component matches, but the validity in the full theory still has to be tested.

The domain sizes of the model influence the precision of the method. The larger they are, the more the filter behaves like a prover for the full theory. We have experimented with different sizes and have obtained good results using a fragment of the full theory containing only the objects `nul` and `suc(nul)` as integers, `nil` and `cons(nul,nil)` as lists and `inc` denoting illegal terms.

We will show now some parts of the model definition. These first two parts define the objects of the sorts `nat` and `seq` and restrict the domains by introducing fixpoints in the constructor functions:

```
nat(nul).
nat(suc(nul)).
suc(nul) != nul.
suc(suc(nul)) = suc(nul).

seq(nil).
seq(cons(nul,nil)).
cons(nul,nil) != nil.
cons(nul,cons(nul,nil))
```

```
        = cons(nul,nil).
```

Other functions are defined as usual except that the domain limitations have to be taken into account:

```
hd(cons(nul,nil)) = nul.
tl(cons(nul,nil)) = nil.
len(nil) = nul.
len(cons(nul,nil)) = suc(nul).

concat(nil,nil) = nil.
concat(nil,cons(nul,nil))
      = cons(nul,nil).
concat(cons(nul,nil),nil)
      = cons(nul,nil).
concat(cons(nul,nil),cons(nul,nil))
      = cons(nul,nil).
```

In order to remove any incomplete definitions an object `inc` (inconsistent) is introduced. It is used to turn partial functions into total functions. The following definitions are necessary to make the `cons`-function total:

```
cons(nil,x) = inc.
cons(cons(x,y),z) = inc.
cons(suc(nul),y) = inc.
-nat(x) | cons(y,x) = inc.
cons(inc,x) = inc.
cons(x,inc) = inc.
```

The computation of a basic model by `anldp` which is sufficient for many obligations needs 0.50 sec. The complete model for lists and basic integer arithmetic needs 1.56 sec to be computed.

The result of the computation is the obvious model. `anldp` uses internal object names which can be assigned to constants (here: 0 = `nul`, 1 = `nil` and 2 = `inc`.) The other objects are assigned by `anldp` while the model is constructed. In the following part of the model `anldp` has assigned 3 to `suc(nul)` and 4 to `cons(nul,nil)`:

```
nat:  0 1 2 3 4    seq:  0 1 2 3 4
---------------    ---------------
      T F F T F          F T F F T

suc:  0 1 2 3 4    len:  0 1 2 3 4
---------------    ---------------
      3 2 2 3 2          2 0 2 2 3
```

Returning to our retrieval example there are effectively three possible results for an obligation:

- It is valid and the model is found.
- It is not valid but the model is found.
- It is not valid and the model is not found.

Considering the list domain (`nil` and `cons(nul,nil)`) you can expect that model checking will be able to distinguish sequence insertions from deletions or changes but not the location where an insertion takes place.

An example for a component which is filtered out by model checking is an assignment to the head of a list. The following proof obligation (Otter format) for `ListSBM/SUM.SetItem` is checked in less than 2 secs (with no model found):

```
formula_list(usable).
  (all st1 all it all st2
   ((seq(st1) & nat(it) & seq(st2)) ->
     ((st2 = cons(it,tl(st1)))
      ->
     ((tl(st2) = st1) & (hd(st2) = it))))).
end_of_list.
```

For the next two examples a model was found also within two seconds. The first is a valid obligation while the second is not valid in the full theory (giving only the component specification):

```
StackSBMI/SUMI.Push:
  (st2 = cons(it,st1)).

QueueSBMI/SUMI.Arrive:
  (st2 = concat(st1,cons(it,st1))).
```

The complete list of the seven matched components is found in picture 1 in the second inspector window. Thus model checking has eliminated most of the irrelevant components in this experiment.

5 Preliminary experiences

The experiments reported here are based on about half of Lins' library. For efficiency reasons, the informations necessary for NORA/HAMMR are compiled from the library and stored in a database. Each entry contains the type signature and pre- and postconditions for the matching process and references to the different definitions of a component. Also, some retrieval relevant intermediate results like the binding of names generated by the type matcher are stored there. The retrieval process is considerably sped-up by an indexing scheme which is based on the principal operators in the type signature. This means that a large part of the components with incompatible type is not even accessed.

Table 1 displays the filtering effect of the three phases of NORA/HAMMR. The left column gives a short description of the search key. The columns for the type matcher and the model checker give counts for the successfully matched procedures and the modules in which these are contained. The last column gives the results of the respective OTTER runs which are either a successful proof (runtime in seconds[1]) or there was no proof within

[1] All times were measured on a SPARC ELC-10.

#	description	sig. match	model check	OTTER runs
1	Insert at head of seq.	25/14	7/4	$4 \times 2s/3 \times$ np-
2	Seq. split at element	1/1	—	np+ $(48s)$
3	Seq. split at position	1/1	1/1	$1s$
4	Member?-predicate	3/3	3/3	$3 \times$ np+
5	Position of element in seq.	9/9	9/9	$9 \times 1s$
6	Remove from front of seq.	51/20	6/3	$6 \times 2s$
7	Remove from back of seq.	51/20	6/3	$6 \times$ np-

Table 1: Experimental results

a short time limit for a valid proof obligation (np+) resp. an invalid one (np-).

Most search keys only produce easy proof obligations which OTTER proves in a few seconds each. Experiment 4 creates obligations for which it fails to find a proof but at least the model checker successfully shows their validity in the small theory. Another search key that caused some problems for OTTER and the model checker is the "element-split" of experiment 2. anldp fails here, because it cannot handle skolem functions of arity larger than four. OTTER is able to find a proof of the resulting obligation but clearly exceeds the given time limit. Interestingly, we got the best results when OTTER was allowed to choose its parameters itself (OTTER's so-called automode).

For the above experiments, overall recall was 0.49 and overall precision was 0.86. As expected, precision is very high. The rather poor recall comes from signature matching: at the moment, the equivalence E on signatures is too restrictive, excluding some relevant components. Additional axioms for E will increase recall and decrease precision of signature matching — but overall precision is maintained by the model checker and theorem prover.

6 Conclusions

Due to the concept of successive filters, our retrieval system NORA/HAMMR is able to present acceptable intermediate results in short time. A specialized replacement of anldp will even lead to better results. We will also experiment with other filters based on unsound proving methods. The replacement of OTTER by other theorem provers is another possibility for improvement. Recent experiments with the SETHEO prover showed that SETHEO is at least as suitable as OTTER.

Acknowledgements

NORA/HAMMR is part of the inference-based software evelopment environment NORA[2]. Ch. Lindig developed the graphical user interface for NORA/HAMMR. M. Kievernagel and B. Fischer were supported by DFG, grants Sn11/1-2, Sn11/2-2, Sn11/3-1 and Sn11/4-1.

[2]NORA is a play by the Norwegian writer H. IBSEN, hence NORA is no real acronym. HAMMR stands for "higly adaptive multi-method retrieval"

References

[Dawes, 1991] John Dawes. *The VDM-SL Reference Guide.* Pitman, London, 1991.

[Letz *et al.*, 1992] R. Letz, J. Schumann, S. Bayerl, and W. Bibel. Setheo: A high performance theorem prover. *Journal of Automated Reasoning*, 2(8):183–212, 1992.

[Lindig, 1995] C. Lindig. Concept-based component retrieval. In *Proc. IJCAI Workshop on Reuse of Proofs, Plans and Programs*, 1995. to appear.

[Lins, 1989] Charles Lins. *The Modula-2 Software Component Library.* Springer Compass International. Springer Verlag, New York Berlin Heidelberg, 1989.

[Manhart and Meggendorfer, 1991] P. Manhart and S. Meggendorfer. A knowledge and deduction based software retrieval tool. In *Proc. 4th International Symposium on Artificial Intelligence*, pages 29–36, 1991.

[McCune, 1994a] W. W. McCune. A Davis-Putnam program and its application to finite first-order model search: Quasigroup existence problems. Technical report, Argonne National Laboratory, 1994. Draft.

[McCune, 1994b] W. W. McCune. Otter 3.0 user's guide. Argonne National Laboratory Report ANL-94/6, 1994.

[Rittri, 1990] Mikael Rittri. Retrieving library identifiers via equational matching of types. In Mark E. Stickel, editor, *Proc. 10th International Conference on Automated Deduction*, volume 449 of *Lecture Notes in Computer Science.* Springer-Verlag, July 1990.

[Rollins and Wing, 1991] Eugene J. Rollins and Jeannette M. Wing. Specifications as search keys for software libraries. In Koichi Furukawa, editor, *Proc. of the Eighth International Conference and Symposium on Logic Programming*, pages 173–187, Paris, June 24-28 1991. MIT Press.