VCR: A VDM-based software component retrieval tool

B. Fischer M. Kievernagel W. Struckmann

TU Braunschweig, Abteilung für Softwaretechnologie
Gaußstraße 17, D-38092 Braunschweig, Germany
{fisch,mkiever,struck}@ips.cs.tu-bs.de

Abstract

We present a tool which allows implicit VDM specifications to be used as search keys for the retrieval of software components. A preprocessing phase utilizes signature matching to filter promising candidates out of a component library. The actual specification matching phase builds proof obligations from the specifications of key and candidates and feeds them into a theorem prover. Validated obligations denote matching components. First experiments clearly demonstrate the feasibility of this approach. We thus get a high-precision retrieval tool which helps programmers in locating components which exactly match their needs.

Keywords: *formal methods, software component retrieval, signature matching, specification matching, theorem proving, model searching.*

1 Introduction

Effective software component retrieval methods play a key role in reuse. Most methods grew out of classical information retrieval (e. g. [11, 8]) but recently semantic-based methods have gained more attention.

As opposed to the former, which rely on an external classification scheme, the latter use semantic information which is intrinsic to the components like for example type schemes [12, 14] or axiomatic specifications [13]. Such semantic-based methods also allow the automatic addition of components to a library and do not require the overhead of manual classification of most text-based methods.

Using type schemes as search keys (also called *signature matching*) is well understood, but its application has been restricted to functional languages. Work on specifications as search keys is, however, much rarer. In this paper we show how implicit VDM specifications may be used as search keys for the retrieval of software components. Figure 1 sketches our general concept. It is based on a large software component library. The components are implemented in some imperative language and annotated with implicit VDM specifications.

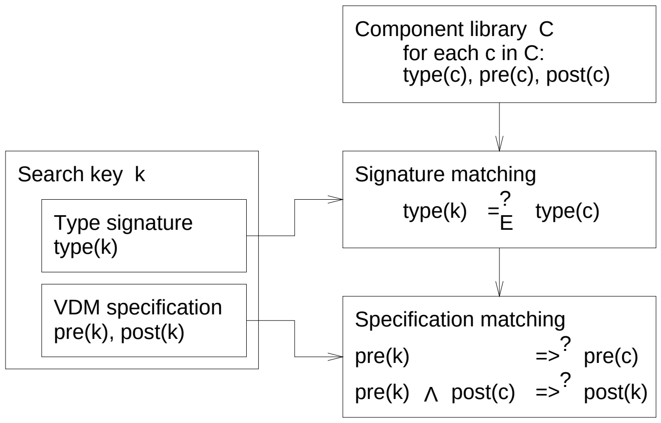


Figure 1: Search concept

Our test library [7] consists of about 50 modules implementing several variants of typical abstract data types like stacks, queues, graphs, and trees using generic items. It provides approx. 1000 procedures with 120 different type signatures. Since the four most frequent signatures already cover a third of the procedures, signature matching alone is not sufficient to identify a procedure adequately.

Remaining components pass through further filters which match them against parts of the key. Figure 1 depicts our specification matching program as second filter. It builds proof obligations from the pre- and postconditions of the search key and the components and feeds them into a theorem prover. Each validated pair denotes a match.

The successive filtering of components offers two main advantages. It allows free combination of different retrieval methods—including text-based methods. 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.

This concept has been implemented in VCR—a VDM-based Component Retrieval tool. It is a high-precision retrieval tool which helps programmers in locating components which exactly match their needs. VCR has been developed in the NORA/HAMMR-project, which is part of the inference-based software development environment NORA[16, 3, 6].

2 Search keys and signature matching

Figure 1 has already sketched the main parts of VCR. Since specification matching and library organization will be discussed in subsequent sections, we will concentrate here on search keys and signature matching.

The search keys, through which a user mainly communicates with VCR, consist of a type signature and a specification, 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 is essentially a Modula-2 type definition extended by type variables (`I` and `S`) to abstract naming of types. The specification is written in VDM-SL [2], but some naming conventions are applied to refer to parameters and result.

The main characteristic of signature matching is the equivalence E on types. Some equational theories have been investigated for functional languages [12, 14, 13] to increase recall, i. e. the number of interesting objects found (see also [15]). These include for example axioms to handle currying and different argument orders. We extended this to cover imperative features like `VAR`-parameters as well. Thus, VCR is able to match the procedure `Push( VAR st:STACK; it:ITEM )` against the sample key.

Besides its use as a preprocessor signature matching is also necessary in order to identify corresponding variables in the specifications. The actual specification of `Push` looks like this:

```
operations Push  ( st : Stack; it : Item ) res : Stack
           pre   true
           post  res = [it] ^ st
```

Hence, the names st, it and res have to be bound to s, i and x in the key, respectively.

The proof obligations $\mathsf{pre(k)} \Rightarrow \mathsf{pre(c)}$ and $\mathsf{pre(k)} \wedge \mathsf{post(c)} \Rightarrow \mathsf{post(k)}$ express that a candidate component c matches the given search key k iff it simultaneously has a weaker precondition and a stronger postcondition than the key. The special form of the second condition additionally allows to match components realizing a less partial function than the key. In other words, any found component is ready "to be plugged in" since it requires less than specified but grants more.

3 Specification matching

In this section we will explain how VCR checks a specification against the filtered library components. The crucial step during this phase is to employ some kind of proof procedure to check the resulting proof obligations. We decided not to hard-wire a special proof procedure for VDM but to integrate the general purpose theorem prover OTTER[10] and the associated model finder `anldp`[9]. OTTER is based on the resolution principle and can handle formulas of the non-sorted first order fragment of predicate calculus with equality. 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.

We divided specification matching into four major steps as shown in figure 2. The translation step produces the sequence of proof obligations from the key and the type matched components and turns these into OTTER syntax. It is based on a multiple entry parser for VDM-SL, i.e. it is also able to parse single conditions.

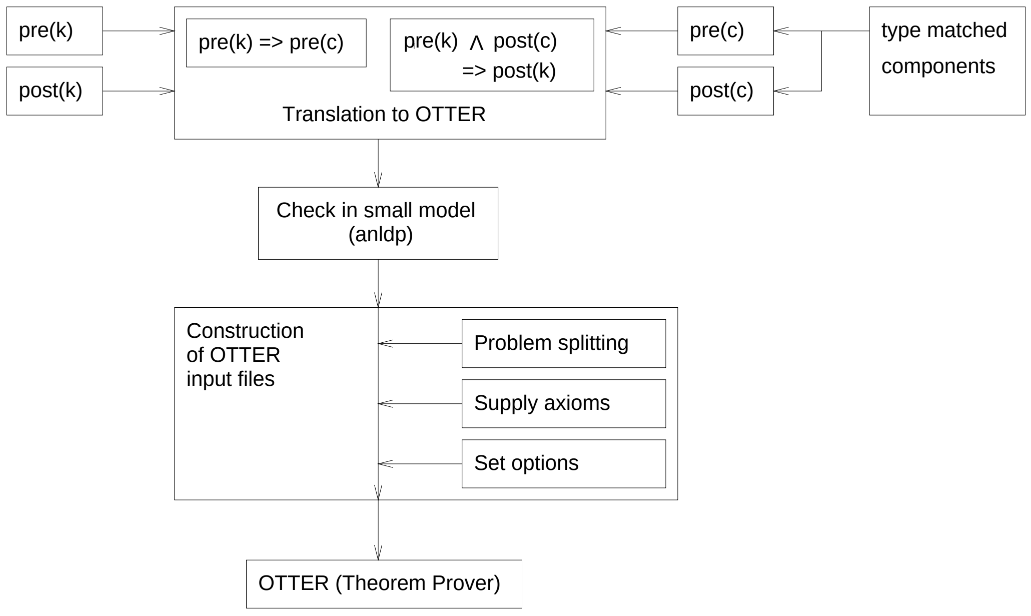


Figure 2: Specification matching

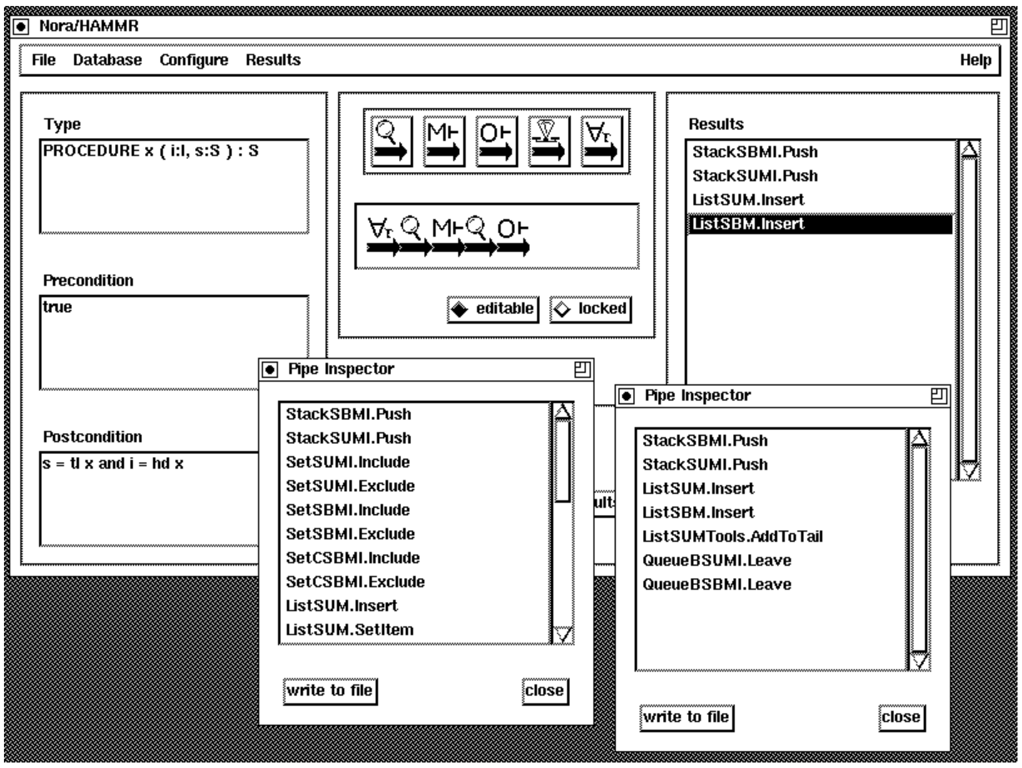
The second step purges obligations which can easily be refuted by checking their validity in a small fragment of the VDM-axiomatization. Its basic idea is to check whether all assignments of small integers and small lists, resp., to program variables evaluate the obligations to `true`. Obviously, this is a prerequisite for the obligations to be provable in the full theory. The filter thus extends each obligation by an axiom set for integers and lists which has exactly one finite model. `anldp` then tries to find this designated model using a modified first-order Davis-Putnam procedure (i. e. enumeration of all finite models.) If this fails the component is rejected. Otherwise, since this filter is not sound, the validity in the full theory still has to be tested.

The third step creates the problem description for OTTER and the full theory. First the problems are split into independent subproblems in order to reduce the search space. This is done by transforming the formula into disjunctive normal form and combining every set of disjunctions with common variables into one subproblem. Another reduction of the search space stems from the fact that we do not supply each problem with the complete axiomatization of the full theory, but axiomatize the problems independently, i.e. link them dynamically with an appropriate set of axioms. The complete axiomatization consists of about 120 axioms and lemmata and is based on [1] for sequences and on our earlier work in automatic program verification [5, 4] for arithmetic. The problems presented in this paper needed up to 25 axioms from this set.

To complete the problem descriptions some options for OTTER have to be set accordingly. Those regarding inference rules, especially the handling of equality, vary with the chosen axiomatization, while some limits, for example the number of demodulations per inference, vary with respective size properties of the problem.

Finally, OTTER is run on the generated problems and its output is analyzed. An obligation denotes a match iff each of its split parts have been proven. The matched components—final or intermediate results— are represented by their location (module) and their name.

Our graphical user interface (see figure 3) reflects the idea of successive filtering. Additionally, inspectors grant easy access to components (VDM-SL specification and Modula-2 code) in intermediate results. This filter-inspector-chain may easily be customized by the user through an icon pad. The configuration displayed below corresponds to the sequence of filters described in this paper and the picture is taken from experiment 1 of the next section. 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.



Type
PROCEDURE x (i:I, s:S) : S

Precondition
true

Postcondition
s = tl x and i = hd x

Results
StackSBMI.Push
StackSUMI.Push
ListSUM.Insert
ListSBM.Insert

Pipe Inspector
StackSBMI.Push
StackSUMI.Push
SetSUMI.Include
SetSUMI.Exclude
SetSBMI.Include
SetSBMI.Exclude
SetCSBMI.Include
SetCSBMI.Exclude
ListSUM.Insert
ListSUM.SetItem

Pipe Inspector
StackSBMI.Push
StackSUMI.Push
ListSUM.Insert
ListSBM.Insert
ListSUMTools.AddToTail
QueueBSUMI.Leave
QueueBSBMI.Leave

Figure 3: Graphical user interface

The objective of the graphical user interface is to hide all details of prover usage and VCR's internal structure from a user. The knowledge necessary for its use is thus restricted to the specification language (VDM-SL) and the target language (Modula-2), which are prerequisites, anyway.

4 Practical experience

The experiments reported here are based on about half of Lins' library. We have specified all list-like structures from singly-linked lists to priority queues and deques and the basic set-like data types set and bag. All specifications are based on the structure of the Modula-2 implementation. We represent singly-linked structures and arrays as VDM-sequences. The elementary types and the other structured types are represented in VDM by their nearest counterpart.

The experiments described below used a rather liberal type equality relation for signature matching. It creates a name binding as long as all parts of the type key can also be found in the components' type. Type matched components are purged automatically whenever the name binding is insufficient to build the proof obligations due to missing names in the key.

As already noted, the model checker tests the obligations in finite models. We have experimented with different sizes and have obtained good results using a fragment of the full theory only containing the objects `nul` and `one = suc(nul)` as integers, `nil` and `lnul = cons(nul,nil)` as lists and `inc` denoting illegal terms.

We selected axioms and options for OTTER in a very strict way. In our experience larger axiom sets and options led to search spaces in which OTTER was easily lost. Consequently, a proof for valid obligations may fail. But if this happens the user can always rerun this filter with more generous options.

The following table displays the filtering effect of the three phases of VCR. 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 a short time limit for a valid proof obligation (`np+`) resp. an invalid proof obligation (`np-`).

#	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-`

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

Except for OTTER, there are no runtimes included in this table because some syntax transformations and parts of the process control are not yet fully automated. However, the signature matching filter takes about 2–3 seconds for the whole library depending on the generality of the type key and the number of matching components. The model checker takes about 2–4 seconds per proof obligation varying with their complexity. This time could be drastically reduced by a specialized program because `anldp` needs 1.75 seconds to find the intended model from the axioms of the small theory alone.

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.

We will now show the specification of the procedure `Split` and the given search key as an example of a more complex retrieval experiment with an under-specified key.

$$\begin{array}{l}
\textit{Split(AList : List, AnItem : Item, ToList : List) OutLists : List} \times \textit{List} \\
\mathsf{post}\ ((\forall i \in \mathsf{inds}\ AList \cdot AList(i) \neq AnItem) \\
\qquad\qquad \Rightarrow OutLists = \mathsf{mk\text{-}}(AList, [])) \\
\quad \wedge (\exists i \in \mathsf{inds}\ AList \cdot (\forall j \in \mathsf{inds}\ AList \cdot (j < i \Rightarrow AList(j) \neq AnItem) \\
\qquad\qquad\qquad\qquad\qquad\qquad \wedge\ (j = i \Rightarrow AList(i) = AnItem)) \\
\qquad\qquad \Rightarrow OutLists = \mathsf{mk\text{-}}(AList(1, \ldots, i{-}1), AList(i, \ldots, \mathsf{len}\ AList)))
\end{array}$$

The procedure `Split` leaves the input lists intact if the given item is not found in *AList* and otherwise splits *AList* at the leftmost occurrence of *AnItem* whereas the search key only specifies the result for the case that *AnItem* is present in *AList*.

$$\begin{array}{l}
\texttt{PROCEDURE x (VAR l1 : LIST; it : ITEM; VAR l2 : LIST)} \\
\mathsf{post}\ \exists m \in \mathsf{inds}\ l1 \cdot (\forall n \in \mathsf{inds}\ l1 \cdot (m > n \Rightarrow l1(n) \neq it) \\
\qquad\qquad\qquad\qquad\qquad \wedge (m = n \Rightarrow l1(m) = it)) \\
\qquad\qquad \Rightarrow (l1 = x1\ \hat{}\ x3 \wedge \mathsf{len}\ x1 = m - 1)
\end{array}$$

The construction phase of the specification matcher selects 12 axioms, which are given in addition to the obligation to OTTER. Half of the axioms are not necessary to complete the proof and especially one of them leads to a much increased search space. If the axioms are selected by hand, OTTER is able to find a proof within two seconds.

Finally, we want to comment on the adequacy of the found procedures. In experiments 2–5 all relevant procedures are found though some of them cannot be proved automatically to match the requirements, because specification matching is restricted of course by the undecidability of the problem in general and by the proving power of OTTER in particular. For the other experiments there are additional procedures in the double-ended-queue (deque) modules. These are not found by the signature matcher because the implementation uses an extra enumeration parameter to distinguish operations at the front and the back of a deque.

The following table gives the classical retrieval measures of recall ($\mathcal{R}$ = retrieved relevant components / all relevant components) and precision ($\mathcal{P}$ = retrieved relevant components / all retrieved components) for the experiments at each filter (average-2 is computed only from non-empty results.)

problem	sig. match		model check		spec. match		best filter	
	$\mathcal{R}$	$\mathcal{P}$	$\mathcal{R}$	$\mathcal{P}$	$\mathcal{R}$	$\mathcal{P}$	$\mathcal{R}$	$\mathcal{P}$
1	0.67	0.16	0.67	0.57	0.67	1	0.67	1
2	1	1	0	–	0	–	1	1
3	1	1	1	1	1	1	1	1
4	1	1	1	1	0	–	1	1
5	1	1	1	1	1	1	1	1
6	0.75	0.12	0.75	1	0.75	1	0.75	1
7	0	0	0	0	0	–	0	0
average	0.77	0.61	0.63	0.65	0.49	0.57	0.77	0.86
average-2	0.90	0.71	0.88	0.91	0.85	1	0.90	1

The table shows that when signature matching produced a result of low precision the model check and specification matching filters increased the precision to 1. And for two-thirds of the experiments VCR succeeded in proving retrieved components to fulfill the requirements.

5 Conclusions

Our first practical experience confirmed our approach. VCR is able to locate software components via matching of implicit VDM specifications. The computational effort, however, is high. But due to the concept of successive filters, VCR is able to present acceptable intermediate results in short time. A specialized replacement of `anldp` will even lead to better results. Another point where improvement is surely possible is the selection of axioms. Parts of an axiomatization can be used to rewrite obligations into a normalized form, which will only need a reduced axiom set to be proven.

Once VCR's integration is completed we will assess its effect on reuse in a large a programming project. We expect to find some positive effects especially regarding the "fitness" of retrieved components and consequently a further reduced need of testing.

Acknowledgements

G. Snelting originally proposed the idea of NORA/HAMMR. C. Lindig designed parts of the user interface, J. Rhiemeier implemented the VDM-SL parser, P. Heise and C. v. Grone specified parts of Lins' library, and E. Gode implemented the first version of the signature matching.

M. Kievernagel and B. Fischer were supported by DFG, grants Sn11/1-2 and Sn11/2-2.

References

[1] J. C. Bicarregui, J. S. Fitzgerald, P. A. Lindsay, R. Moore, and B. Ritchie. *Proof in VDM: A Practitioner's Guide.* FACIT series. Springer, Berlin, 1993.

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

[3] F.-J. Grosch and G. Snelting. Polymorphic components for monomorphic languages. In R. Prieto-Diaz and W. B. Frakes, editors, *Proc. of the 2nd International Workshop on Software Reusability*, pages 47–55, Lucca, Italy, Mar. 1993. IEEE Computer Society Press.

[4] B. Hohlfeld and W. Struckmann. *Einführung in die Programmverifikation.* Reihe Informatik. BI Wissenschaftsverlag, Mannheim/Leipzig/Wien/Zürich, 1992.

[5] M. Kievernagel. Auswahl und Installation eines Beweissystems. Master's thesis, Technical University of Braunschweig, Germany, Feb. 1990.

[6] M. Krone and G. Snelting. On the inference of configuration structures from source code. In *Proc. 16th ICSE*, pages 49–57. IEEE Computer Society Press, May 1994.

[7] C. Lins. *The Modula-2 Software Component Library.* Springer Compass International. Springer, Berlin, 1989.

[8] Y. S. Maarek, D. M. Berry, and G. E. Kaiser. An information retrieval approach for automatically constructing software libraries. *IEEE TOSE*, SE-17(8):800–813, 1991.

[9] 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.

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

[11] R. Prieto-Diaz. Classifying software for reusability. *IEEE Software*, 4(1), Jan. 1987.

[12] M. Rittri. Retrieving library identifiers via equational matching of types. In M. E. Stickel, editor, *Proc. 10th CADE*, *LNCS* 449. Springer, July 1990.

[13] E. J. Rollins and J. M. Wing. Specifications as search keys for software libraries. In K. Furukawa, editor, *Proc. 8th ICLP*, pages 173–187, Paris, June 24-28 1991. MIT Press.

[14] C. Runciman and I. Toyn. Retrieving re-usable software components by polymorphic type. In *Proc. 4th FPCA*, 1989.

[15] G. Salton and M. J. McGill. *Introduction to Modern Information Retrieval.* McGraw-Hill, New York, 1983.

[16] G. Snelting, B. Fischer, F.-J. Grosch, M. Kievernagel, and A. Zeller. Die inferenzbasierte Softwareentwicklungsumgebung NORA. *Informatik—Forschung und Entwicklung*, 9(3):116–131, Aug. 1994.