# NORA/HAMMR: Making Deduction-Based Software Component Retrieval Practical*†

Bernd Fischer

Abt. Softwaretechnologie, TU Braunschweig, D-38092 Braunschweig
fisch@ips.cs.tu-bs.de

Johann Schumann

Automated Reasoning, Institut für Informatik, TU München, D-80290 München
schumann@informatik.tu-muenchen.de

## Abstract

Deduction-based software component retrieval uses pre- and postconditions as indexes and search keys and an automated theorem prover (ATP) to check whether a component matches. This idea is very simple but the vast number of arising proof tasks makes a practical implementation very hard. We thus pass the components through a chain of filters of increasing deductive power. In this chain, rejection filters based on signature matching and model checking techniques are used to rule out non-matches as early as possible and to prevent the subsequent ATP from "drowning." Hence, intermediate results of reasonable precision are available at (almost) any time of the retrieval process. The final ATP step then works as a confirmation filter to lift the precision of the answer set. We implemented a chain which runs fully automatically and uses MACE for model checking and SETHEO as ATP and evaluated it over a medium-sized collection of components. The results confirm the practicality of our approach.

## 1 Introduction

Reuse of approved software components has been identified as one of the key factors for successful software engineering projects. Although the reuse process also covers many non-technical aspects [ZS95], retrieving appropriate software components from a reuse library is a central task. This is best captured by Tracz's *First Golden Rule for Software Reuse*: "You must find it before you can reuse it!"

Most earlier software component retrieval (SCR) methods (e.g., [MS89]) grew out of classical information retrieval for unstructured texts. However, since software components are highly structured, more specialized approaches may lead to better results. In this paper we will concentrate on a deduction-based approach where we use pre- and postconditions as the components' indexes and as search keys. A component matches a search key if the involved pre- and postconditions satisfy a well-defined logical relation, e.g., if the component has a weaker precondition and a stronger postcondition than the search key. From this matching relation a proof task is constructed and an ATP is used to establish (or disprove) the match.

This approach has been proposed before (e.g., [KRT87, Per87, RW91, MM91, MMM94, MW95]) but without convincing success because essential user requirements have been neglected. In this

---

*NORA is no real acronym, HAMMR is a highly adaptive multi-method retrieval tool.

paper we follow a more user-oriented approach and describe steps for making deduction-based SCR practical.

## 2 The User's Point of View

Most earlier work focussed on the technical aspects of deduction-based SCR. The users had to write complete specifications in the ATP input language and even had to supply useful lemmata. The provers were run in a batch mode, checking the whole library before any results were presented. Runtimes and problems of scale were ignored.

This view led to severe acceptance problems as the users are software engineers and no ATP experts. Their main requirements are that the tool is easy to use, fully automatized, fast, and customizable, and hides all evidence of automated theorem proving.[1]

Hiding the ATP has some consequences. The input language must be a fully-flavored specification language and not pure first order logic (FOL). But then the automatic construction of the actual proof tasks becomes itself a major task.

Short response times are also essential as the *Fourth Reuse Truism* demands that "you must find it faster than you can rebuild it!"[Kru92]. However, due to the computational complexity of ATP, truly interactive ("sub-second") behavior is still far out of reach. Instead, *anytime behavior* is acceptable: intermediate results of sufficient precision must be available to the user at (almost) any time during the retrieval process. Retrieval may then be guided further with feedback from the user who may for example strengthen the search key incrementally.

Ideally, the tool doesn't constrain user feedback to the queries but allows a customization of the complete retrieval process. This includes the selection of an appropriate match relation from a given list of choices as well as some tuning of the deductive mechanism (e.g., time limits or model sizes). But it is important to ensure that the tool still runs fully automatically and produces useful results even without customization.

In exchange for these constraints, deduction-based SCR offers the unique feature that completeness and even soundness are not absolutely vital. Incomplete and unsound deduction methods only reduce *recall* ("do we get all matching components?") and *precision* ("do we get the right components?").

## 3 System Architecture

In order to meet the user requirements we implemented NORA/HAMMR as a *filter pipeline* through which the candidates are fed. This pipeline typically starts with *signature matching filters*. They check whether candidate and query have "compatible" calling conventions (i.e., types or signatures). The notion of compatibility is specified by an equational theory $E$; the filter then applies $E$-matching or $E$-unification of the signature terms. Typical theories include axioms to handle associativity and commutativity of parameter lists and records, currying (for functional languages), pointer types and VAR-parameters (for imperative languages), coercion rules and more (see [Fis97] for a detailed discussion).

Then, *rejection filters* try to eliminate non-matches as fast as possible. This is a crucial step to prevent the ATP from "drowning" as there are many more non-matching than matching candidates. We currently apply model generation techniques to check the validity of the tasks in suitable finite models. However, both precision and recall may decrease because this approach is neither sound nor complete.

Finally, *confirmation filters* check the validity of the remaining proof tasks and thus lift the precision of the result to 100%. Here, we apply SETHEO, a high-performance prover based on the model elimination calculus. Both filter classes will be described below in more detail.

---

[1]In a real-life setting, a single reuse administrator may be required who knows the applied deduction methods and "tunes" each component before it is entered into the library (e.g., by giving ATP settings and lemmata).
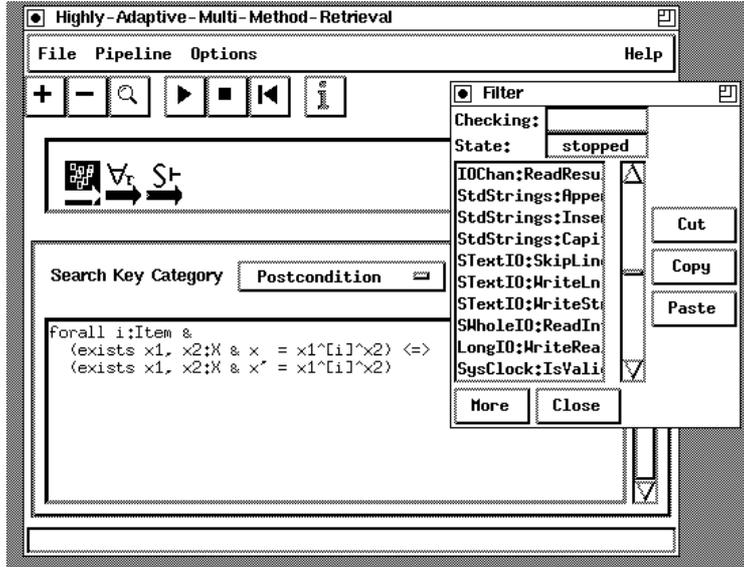
Figure 1: Graphical user interface

The graphical user interface reflects the idea of successive filtering. The pipeline may easily be customized through an icon pad; each filter icon also hides a specialized filter control window which allows some fine-tuning of the filters. Additional inspectors display intermediate results and grant easy access to the components. They also allow to save intermediate results in a file such that they may easily be used as libraries for subsequent retrieval runs. The objective of the GUI is precisely to hide all evidence of ATP usage. Hence, the knowledge necessary to use NORA/HAMMR as a *tool* is restricted to VDM-SL [Daw91] which we use as our input language and to the target language which is required for signature matching.

# 4 Proof Tasks and Reuse

The overall structure of the generated proof tasks depends on the definition of the *match relation* which is used in a deduction-based SCR tool. Thus it ultimately depends on the kind of reuse which the tool aims at.

Most often, deduction-based SCR is configured to ensure *plug-in compatibility* of the retrieved components: $c$ matches if it has a weaker precondition and a stronger postcondition than the search key $q$. This is usually (cf. e.g., [MW95]) formalized as $(pre_q \Rightarrow pre_c) \wedge (post_c \Rightarrow post_q)$[2]. However, this is not adequate for partial functions. If $q$ is a partial function (e.g. *tail*) and $c$ its total completion (e.g., $c(nil)$ returns *nil*) then we want $c$ to match $q$ even if its "completed" result does not fit the original specification. It is thus necessary to restrict the implication between the postconditions on the domain given by $pre_q$. We thus work with proof tasks of the form

$$(pre_q \Rightarrow pre_c) \wedge (pre_q \wedge post_c \Rightarrow post_q) \tag{1}$$

which are similar to [MW95]'s "guarded plug-in match" except for our use of the stronger (via the first implication) precondition from the query. Plug-in compatibility supports safe reuse. The retrieved components may be considered as black boxes and may be reused "as is", without further proviso or modification of the component.

---

[2]Actually, the proof tasks are universally closed wrt. the formal input and output parameters of the component and the query and also contain equations relating the parameters. Likewise, the pre- and postconditions are of course logical functions of the respective parameters. However, to improve the legibility, we use this traditionally abbreviated formulations.

Sometimes plug-in compatibility is not applicable because the users don't want to specify any precondition but are willing to accept whatever comes, as long as their postconditions are met. In that case, (1) simplifies to

$$pre_c \wedge post_c \Rightarrow post_q \qquad (2)$$

or *conditional compatibility*. However, reuse now becomes potentially unsafe because any client still has to satisfy the open obligation $pre_q$.

Sometimes (2) might be too strong, and retrieves no components, although the library contains "almost" matches, e.g., partial functions. To additionally retrieve such components, *partial compatibility* may be used:

$$pre_c \wedge pre_q \wedge post_c \Rightarrow post_q \qquad (3)$$

Thus, a component is retrieved if it computes the right results on the common domain. If, however, the domains of $c$ and $q$ are disjoint, $pre_c$ and $pre_q$ are never true at the same time and thus (3) will become vacuously true. But usually $q$ and $c$ then also work on different types and $c$ should already be rejected by signature matching. If $c$ has an empty domain or is not implementable (i.e., $post_c$ never becomes true), (3) will again become vacuously true and $c$ will be retrieved for any query. However, this should not happen in a well-designed library.

Obviously, reuse based on partial compatibility is unsafe because the retrieved components are not guaranteed to work on the entire required domain. But they might be good starting points for desired more general implementations. Hence, the components must be considered as "white boxes"—their code needs a closer inspection.

As an example, let us consider the following VDM-SL specifications:[3]

| | | | |
|---|---|---|---|
| $rotate(l : List)\ l' : List$ | | $shuffle(x : X)\ x' : X$ | |
| pre | true | pre | true |
| post | $(l = []\ \Rightarrow l' = [])\wedge$ | post | $\forall i : Item \cdot (\exists x_1, x_2 : X \cdot x = x_1{}^\wedge[i]{}^\wedge x_2 \Leftrightarrow$ |
| | $(l \neq []\ \Rightarrow l' = (\mathsf{tl}\ l){}^\wedge[\mathsf{hd}\ l])$ | | $\exists x_1, x_2 : X \cdot x' = x_1{}^\wedge[i]{}^\wedge x_2)$ |

Let us further assume that we use plug-in-compatibility as match relation, *rotate* as candidate $c$ and *shuffle* as query $q$. Then several steps are necessary to construct a sorted FOL proof task. First, the formal parameters must be identified, in this case $l = x$ and $l' = x'$.[4] Then, VDM's underlying three-valued logic LPF must be translated into FOL. This essentially requires the explicit insertion of additional preconditions into the proof task to prevent reasoning from undefined terms as well as a translation of the connectives which takes care of the missing law of the excluded middle [JM94, Mid93]. In our example, this results in the proof task

$$\forall l, l', x, x' : List \cdot (l = x \wedge l' = x' \wedge \mathsf{true} \Rightarrow \mathsf{true})$$
$$\wedge \quad (l = x \wedge l' = x' \wedge (l = []\ \Rightarrow l' = []) \wedge (l \neq []\ \Rightarrow (l \neq []\ \Rightarrow l' = (\mathsf{tl}\ l){}^\wedge[\mathsf{hd}\ l]))$$
$$\Rightarrow (\forall i : Item \cdot (\exists x_1, x_2 : X \cdot x = x_1{}^\wedge[i]{}^\wedge x_2 \Leftrightarrow \exists x_1, x_2 : X \cdot x' = x_1{}^\wedge[i]{}^\wedge x_2)))$$

## 5   Rejecting Non-Matches

Detecting and rejecting non-matching components as fast and early as possible is probably the most important single step in making deduction-based SCR practical—there are simply many more non-matching than matching components. Unfortunately, most ATPs are not suited for this task. They exhaustively search for a proof (or refutation) of a conjecture but are practically unable to conclude that it is *not* valid (or contradictory). Therefore, other techniques have to be used to implement rejection filters.

Generally, we may reject a component $c$ if we find a "counterexample" for its associated proof task $\mathcal{T}_c$ because it then cannot be valid. Model generators for FOL like Finder [Sla94] or MACE

---

[3]Here, $^\wedge$ means the concatenation of lists, $[]$ the empty list, $[i]$ a singleton list with item $i$, and $\mathsf{hd}$ and $\mathsf{tl}$ the functions head and tail, respectively.

[4]This identification is, however, not always a simple renaming substitution as VDM-SL allows pattern matching and complex data types.

[McC94] try to find such counterexamples (which are simply interpretations under which $\mathcal{T}_c$ evaluates to *false*) by systematically checking all possible interpretations. This obviously terminates only if all involved domains are finite, as for example in finite group theory or hardware verification problems. On the other hand, their highly efficient implementation (usually using BDD-based Davis-Putnam decision procedures) would make model generators ideal candidates for fast rejection filters,

However, most domains in our application are not finite but unbounded, e.g., numbers or lists. If we want to use model generation techniques for our purpose, we must map these infinite domains onto finite representations, either by *abstraction* or by *approximation.*

## 5.1   Mapping by Abstraction

One approach to establish this mapping uses techniques from abstract interpretation [CC77] where the infinite domain is partioned into a small finite number of sets which are called abstract domains. For each function $f$ an abstract counterpart $\bar{f}$ is constructed such that $f$ and $\bar{f}$ commute with the abstraction function $\alpha$ between original and abstract domains, i.e., $\alpha(f(x)) = \bar{f}(\alpha(x))$. E.g., we may partition the domain of integers into three abstract domains $\{0\}$, $\{x \mid x > 0\}$ and $\{x \mid x < 0\}$, called *zero, pos* and *neg*. Then, all operations for integers must be abstracted accordingly. For example, for the multiplication $\times$, we get the abstract multiplication $\bar{\times}$ which actually mirrors the "sign rule": $neg\,\bar{\times}\,pos = pos\,\bar{\times}\,neg = neg$.

Abstract model checking [Jac94] then represents the abstract domains by single model elements and tries to find an abstract countermodel, using an axiomatization of the abstract functions and predicates with a standard FOL model generator. There is, however, a problem. While abstract interpretation may escape to a larger "abstract" domain of truth values in order to make the predicates commute with the abstraction function, standard FOL model generators require the exact concrete domain of *true* and *false* and thus a consistent abstraction may become impossible. E.g., when we try to abstract the ordering on the numbers, *less(zero, pos)* is valid but we cannot assign a single truth value to *less(pos, pos)* because two arbitrary positive numbers may be ordered either way.

So, while there are some predicates which allow exact abstractions, we have to approximate others. Since we want to use abstract model checking as a rejection filter, we have to make our choices such that the filter produces as few false counterexamples as possible: spurious matches are handled by the subsequent confirmation filter but improperly rejected components are lost forever.

## 5.2   Mapping by Approximation

The second approach to map an infinite domain onto a finite one is done by approximation. From the infinite domain, we select a number of values which seem to be "crucial" for the module's behavior. E.g., for lists, one usually picks the empty list [] and small lists with one or two elements (e.g., $[a], [a, b]$). Then, we search for a model. This approach mimicks the manual checking for matches: if one has to find a matching component, one first make checks with the empty list and one or two small lists. If this does not succeed, the component cannot be selected. Otherwise, additional checks have to be applied. This approach, however, is neither sound nor complete. There exist invalid formulas for which a model can be found in a finitely approximated domain (e.g., $\forall X : List \cdot \exists i : Item \cdot X = i^{\wedge} X$), and vice versa (e.g., $\exists X, Y, Z : List \cdot X \neq Y \wedge Y \neq Z \wedge Z \neq X$ which has a model only in domains with at least three distinct elements).

While the second case is not too harmful for our application—the performance of the filter just decreases (i.e., more proof tasks can pass), the first one is dangerous: proof tasks describing valid matches might be lost.

The experiments we describe in this paper are based on this approach. For our prototype implementation we used the model generator MACE [McC94]. The proof task (already converted into sorted clausal normal form) and the descriptions of the approximated domains (for an example see Section 7.2 below) are fed into MACE. It reports whether a counterexample could be found or

not, or if the formula itself is already invalid. Only if no counterexample has been found, the proof task passes this filter.

# 6 SETHEO as Confirmation Filter

For the final stage of our filter chain the high-performance theorem prover SETHEO is used. SETHEO is a complete and sound prover for unsorted first-order logic based on the Model Elimination calculus. It accepts formulas in clausal normal form and tries to refute the formula by constructing a closed tableau (a tree of clauses). Completeness is accomplished by limiting the depth of the search space (e.g., with a bound on the size or depth of the tableau) and performing iterative deepening over this bound. In the context of this paper, SETHEO can be seen as a black box which returns "proof found" or "failed to find proof" after the given time-limit. Hence, no further details about SETHEO are given in this paper. For a description of the system and its features see e.g. [LS+92, LMG94, GL+94, MI+97].

With SETHEO's soundness, we obtain a confirmation filter which guarantees that proof tasks which pass it successfully actually select matching components. Due to our hard time constraints, however, means must be taken not to decrease the recall in an unacceptable way. In the following, we describe how SETHEO has to be adapted in order to be integrated into NORA/HAMMR. We discuss important issues like handling of inductive problems, sorts and equality, and the selection of axioms and parameter settings.

## 6.1 Inductive Problems

Whenever recursive specifications are given or recursively defined data structures are used (e.g., lists) many of the proof tasks can be solved by *induction* only. SETHEO itself cannot handle induction and our severe time-constraints don't allow us to use an inductive theorem prover (e.g., [BvH+90]). Therefore, we approximate induction by splitting up the problem into several cases. For example, for a query and candidate with the signature $l : List$, and the corresponding proof task of the form $\forall l : List \cdot \mathcal{F}(l)$ we obtain the following cases: $l = [] \Rightarrow \mathcal{F}(l), \forall i : Item \cdot l = [i] \Rightarrow \mathcal{F}(l)$, and $\forall i : Item, l_0 : List \cdot \mathcal{F}(l_0) \wedge l = [i]^\wedge l_0 \Rightarrow \mathcal{F}(l)$.[5] After rewriting the formula accordingly, we get three independent first order proof tasks which then must be processed by SETHEO. This approach can be implemented efficiently. However, we cannot solve every inductive problem.

## 6.2 Equality

All proof tasks heavily rely upon equations. This is due to the VDM-SL specification style and the construction of the proof tasks. While some equations just equate the formal parameters of the query and the library module, others carry information about the modules' behavior. Therefore, efficient means for handling equalities must be provided.

We currently provide two variants: the naïve approach by adding the corresponding axioms of equality (reflexivity, symmetry, transitivity, and substitution axioms), and the compilation approach used within E-SETHEO [MI+97].

## 6.3 Sorts

All proof tasks are sorted. The sorts are imposed from the VDM-SL specifications of the modules and are structured in a hierarchical way. All sorts are static and there is only limited overloading of function symbols. Therefore, the approach to *compile* the sort information into the terms of the formula can be used. Then, the determination of the sort of a term and checking, if the sorts of

---

[5]Although it would be sufficient to have cases 1 and 3 only, we also generate case 2, since many specifications are valid for non-empty lists only. For those specifications, case 1 would be a trivial proof task which does not contribute to filtering.

two terms are compatible is handled by the usual unification. Thus there is no need to modify the prover (here SETHEO) and the overall loss of efficiency is minimal.[6] Our current prototype uses the tool ProSpec (developed within Protein [BF94]).

## 6.4   Selection of Axioms

Each proof task has to contain—besides the theorem and the hypotheses—the features of each data type (e.g., *List, nat*) as a set of *axioms*. Automated theorem provers, however, are extremely sensitive w.r.t. the number and structure of the axioms added to the formula. Adding a single (unnecessary) axiom can increase the run-time of the prover by magnitudes, thus decreasing recall in an unacceptable way. In general, selecting the optimal subset of axioms is a very hard problem and has not been solved in a satisfactory way yet. Our hard time-constraints furthermore won't allow us to use time-consuming selection techniques. In our prototype, we therefore use a simple strategy:

1. select only theories for the data types (e.g., list, nat, boolean) occurring in the proof task,

2. within such theories, only select clauses which have function symbols in common with the proof task, and

3. leave out particular clauses and axioms which are known to increase the search space substantially (e.g., long clauses, Non-Horn clauses).

Although this approach is not complete, we use it, since our aim is to solve as many obvious and simple proof tasks (i.e., those which don't use many axioms or have a complex proof) within short limits of run-time.

## 6.5   Control

Once started, the theorem prover has only a few seconds of run time to search for a proof. This requires that the parameters which control and influence the search (e.g., way of iterative deepening, subgoal reordering) are set in an optimal way for the given proof task. However, such a global setting does not exist for our application domain. In order to obtain optimal efficiency combined with short answer times, *parallel competition* over parameters is used. The basic ideas has been developed for SiCoTHEO [Sch96] and could be adapted easily: on all available processors (e.g., a network of workstations), a copy of SETHEO is started to process the entire given proof task. On each processor, a different setting of parameters is used. The process which first finds a proof "wins" and aborts the other processors.

## 7   Experimental Results

All experiments were carried out over a database of 55 list specifications which were modified to have the type *list → list* in order to please our still very simple signature matching filter. Approximately 40 of these specifications describe actual list processing functions (e.g., *tail* or *rotate*) while the rest simulates queries. We thus included underdetermined specifications (e.g., the result is an arbitrary front segment of the argument list) as well as specifications which don't refer to the arguments (e.g., the result is not empty). For simplicity, we formulated the specifications such that all preconditions were *true* and the postconditions only used VDM-SL's built-in sequences.

In order to simulate a realistic number of queries we then cross-matched each specification against the entire library, using plug-in compatibility as match relation. This yielded a total of 3025 proof tasks where 375 or 12.4% were valid.

---

[6]If the sort hierarchy is a tree (which it is in our case), the number of unification steps increase linearly with the depth of the sort hierarchy. In principle, any DAG structured sort hierarchy can be compiled into the terms, but in the general case, exponentially many variables might have to be used.

## 7.1 Evaluation of Filters

Information retrieval methods are evaluated [SM83] by the two criteria precision and recall. Both are calculated from the set $REL$ of *relevant* components which satisfy the given match relation wrt. to the query and $RET$, the set of *retrieved* components which actually pass the filter. The *precision*

$$p = \frac{\mid REL \cap RET \mid}{\mid RET \mid}$$

is defined as the relative number of hits in the response while the *recall*

$$r = \frac{\mid REL \cap RET \mid}{\mid REL \mid}$$

measures the system's relative ability to retrieve relevant components. Ideally, both numbers would be 1 (i.e. the system retrieves all and only matching components) but in practice they are antagonistic: a higher precision is usually paid for with a lower recall.

We also need some metrics to evaluate the filtering effect. To this end we define the *fallout*

$$f = \frac{\mid RET \backslash REL \mid}{\mid \mathcal{L} \backslash REL \mid}$$

(where $\mathcal{L}$ is the entire library) as the fraction of non-matching components which pass the filter as well as the *reduction* which is just the relative number of refuted components. Finally, we define the *relative defect ratio* by

$$dr = \frac{\mid REL \backslash RET \mid}{\mid \mathcal{L} \backslash REL \mid} \cdot \frac{\mid \mathcal{L} \mid}{\mid REL \mid}$$

as the relative number of rejected matching components in relation to the precision of the filter's input. Thus, a relative defect ratio greater than 1 indicates that the filter's ability to reject only irrelevant components is even worse than a purely random choice.

## 7.2 Rejecting Tasks with Model Generation

For the rejection filter, we currently use three different models as shown in Table 1. They are defined by giving the (finite) domains for all data types (in our case *List, Item*), and the definition tables for all functions. Places marked by $^\dagger$ represent arbitrary choices for the function definitions. These had to be introduced to ensure the finiteness of the model and since MACE cannot handle partial functions. These models also differ slightly in the number of their elements but due to the large number of variables in the proof tasks we are generally confined to very small models.

Our experiments with MACE, however, revealed that the restrictions and peculiarities are not too serious. As shown in Table 2, the model checking filter (with a time-limit of 20 seconds) is able to recover at least 75% of the relevant components, regardless of the particular model. The large standard deviation, however, indicates that the filter's behavior is far from uniform and that it may perform poor for some queries.

Unfortunately, the filter is still too coarse. While each model increases the precision of its answer (compared to the the original 12.4% "precision" of the library) significantly, it still lets too many non-matches pass. The values for fallout indicate that the results in average contain up to 55% of the original non-matching components. Similarly, the overall reduction of approx. 40-50% is at the lower end of our expectations. However, the relative defect ratios show that model checking with any model is at least twice as good as blind guessing.

## 7.3 SETHEO as the Confirmation Filter

All experiments with SETHEO have been carried out on a sun Ultra-SPARC-2. We used parallel competition with 4 processes exploring different ways of handling equality. As the basis for experiments, we used the library mentioned above. Due to technical reasons, however, we had to restrict

8

## Model A   Model B   Model C

*List*:  $[], l \equiv [i]$     *List*:  $[], l_1 \equiv [i],$     *List*:  $[], l_1 \equiv [i_1],$
                                   $l_2 \equiv [i]^\wedge l_1$              $l_2 \equiv [i_2]$

*Item*:  $i$      *Item*:  $i$      *Item*:  $i_1, i_2$

| function | $[]$ | $l$ |
|---|---|---|
| $\texttt{cons}(i, \_)$ | $l$ | $l^\dagger$ |
| $\texttt{tail}(\_)$ | $[]$ | $[]$ |
| $\texttt{head}(\_)$ | $i$ | $i$ |
| $\texttt{app}([], \_)$ | $[]$ | $l$ |
| $\texttt{app}(l, \_)$ | $l$ | $l^\dagger$ |

| function | $[]$ | $l_1$ | $l_2$ |
|---|---|---|---|
| $\texttt{cons}(i, \_)$ | $l_1$ | $l_2$ | $l_2^\dagger$ |
| $\texttt{tail}(\_)$ | $[]$ | $[]$ | $l_1$ |
| $\texttt{head}(\_)$ | $i$ | $i$ | $i$ |
| $\texttt{app}([], \_)$ | $[]$ | $l_1$ | $l_2$ |
| $\texttt{app}(l_1, \_)$ | $l_1$ | $l_2$ | $l_2^\dagger$ |
| $\texttt{app}(l_2, \_)$ | $l_2$ | $l_2^\dagger$ | $l_2^\dagger$ |

| function | $[]$ | $l_1$ | $l_2$ |
|---|---|---|---|
| $\texttt{cons}(i_1, \_)$ | $l_1$ | $l_1^\dagger$ | $l_1^\dagger$ |
| $\texttt{cons}(i_2, \_)$ | $l_2$ | $l_2^\dagger$ | $l_2^\dagger$ |
| $\texttt{tail}(\_)$ | $[]$ | $[]$ | $l_2$ |
| $\texttt{head}(\_)$ | $i_1^\dagger$ | $i_1$ | $i_2$ |
| $\texttt{app}([], \_)$ | $[]$ | $l_1$ | $l_2$ |
| $\texttt{app}(l_1, \_)$ | $l_1$ | $l_1^\dagger$ | $l_1^\dagger$ |
| $\texttt{app}(l_2, \_)$ | $l_2$ | $l_2^\dagger$ | $l_2^\dagger$ |

Table 1: Models used for the experiments

| Model | A | B | C |
|---|---|---|---|
| $|List| + |Item|$ | 2+1 | 3+1 | 3+2 |
| recall $r$ | 74.7% | 76.5% | 81.3% |
| $\sigma_r$ | 0.25 | 0.26 | 0.25 |
| precision $p$ | 18.5% | 19.6% | 16.5% |
| $\sigma_p$ | 0.21 | 0.19 | 0.16 |
| precision increase | 1.5 | 1.6 | 1.3 |
| fallout | 42.8% | 41.0% | 55.5% |
| reduction | 50.1% | 51.7% | 39.0% |
| defect ratio $dr$ | 0.51 | 0.45 | 0.48 |

Table 2: Results of model checking

the number of modules to 49. This resulted in a total of 2401 proof tasks where 204 or 8.5% were solvable and thus represent matches.

In our first set of experiments we tried to retrieve identical modules from the library (i.e., $c \equiv q$). The resulting 49 proof tasks are relatively simple and no induction or axioms are needed to prove them. As expected, SETHEO could show all of them within a time-limit of 20 seconds CPU-time. Whereas the mean run-time was less than 1s, several proof tasks needed up to 13 seconds.

Then we tried to retrieve matching, but non-identical components. Our experimental basis contains 155 such cases. First, these proof tasks were tried without induction. Here, SETHEO was able to solve 46 proof tasks with a standard set of axioms. The rate of recall could be increased drastically, when our approximation of induction was used. With the same set of axioms, 70 proof tasks could be solved. Due to the increased size of the formulas (esp. in the step case), more overflow errors occurred. Nevertheless, with case splitting we have been able to retrieve 18 matches more than without case splitting. Due to the different structure of the search space, 6 tasks could be shown only without case splitting, making the simple mode interesting for parallel competition. In order to obtain the overall recall of the SETHEO confirmation filter, we have to combine the data of both sets of experiments. From a total of $204 = 49 + 155$ possible matches, SETHEO could retrieve 125 (49 identical modules, 70 non-identical with case splitting and 6 without case splitting) modules. This yields an overall recall of 61.2%. However, the standard deviation is relatively high, revealing quite different retrieval results for the various queries. Since SETHEO's proof procedure

is sound, all solved proof tasks correspond to matches, hence the precision is 100%.

The experiments showed that parallel competition with several variants (case splitting, set of axioms, handling of equality) is essential to obtain short answer times. In the experiments described in this paper, we only performed parallel competition on the equality handling (axiomatic approach and E-SETHEO with 2 variants each). It showed that none of the methods is a clear winner. Proof tasks with a rich and complex structure of terms result in large sets of long clauses, if processed by E-SETHEO's preprocessing module. Such huge formulas often just take too long (w.r.t. to our time limits) to be handled by SETHEO. If additionally, only few equational transformations are needed during the proof, the naïve approach certainly needs fewer resources. In many cases, however, complex equational transformations are necessary which clearly favors the use of E-SETHEO.

# 8   Conclusions and Further Work

In this paper, we have presented NORA/HAMMR, a deduction-based software component retrieval tool. Our goal was to show that such a tool is not only theoretically possible but practical with state-of-the-art theorem provers. We thus designed it as a user-configurable pipeline of different filters. Rejection filters are in charge of reducing the number of non-matching query-component-pairs as soon and good as possible. In this paper, we have studied an approach which uses model generation techniques for this purpose. Our experiments with MACE showed that this approach, although neither sound nor complete, returns reasonable results. The final stage of the filter pipeline is always a confirmation filter which ensures that the selected components are really matches. Here, we have used the automated theorem prover SETHEO. Even with a short time-limit of 20 seconds, an overall recall of more than 60% was obtained.

We have evaluated our approach with a reasonable large number of experiments. The results obtained so are very encouraging. Nevertheless, many improvements still have to be made before NORA/HAMMR can really be used in industry. Due to the hard time-constraints ("results while-u-wait"), the reduction of proof-tasks, both in complexity and number is of central importance. Here, powerful rejection filters must ensure that only a few proof tasks remain to be processed by the automated theorem prover. However, our current model-checking filter rejects too much valid matches due to the necessary approximate abstractions. We are thus trying to model exact predicate abstractions with Belnap's four-valued logic [Bel77] which extends the three-valued LPF consistently. A translation into FOL which reflects the explicit falsehood conditions of Belnap's logic then yields a sound rejection filter.

Future work will include experiments with specialized decision procedures for the different theories and dis-proving techniques. Additionally, knowledge-based filters and heuristics will help to reduce the number of tasks to be handled by the confirmation filter. All these filters will be configurable and allow inspection of the behavior of the filter pipeline during each stage of the retrieval.

Current high-performance automated theorem provers are certainly usable as confirmation filters. Much work, however, is still necessary to adapt the ATPs for such kinds of proof tasks. In particular, the requirement of full automatization and the strong time-limits must be obeyed carefully. Key issues in this respect are the handling of inductive proofs, and the selection of appropriate axioms. Here, powerful heuristics as well as additional information, placed in the data base together with the components (e.g., tactics, lemmas, induction schemes) will be helpful.

This application of automated theorem proving carries the unique feature that soundness and completeness are not absolutely vital—unsound and incomplete methods only reduce the precision and recall of the retrieval tool. This allows interesting and promising deduction techniques (e.g., approximating proofs by filter chains or iteration) to be explored and will help to automate software engineering a little further.

# References

[Bel77]     N. D. Belnap. "A useful four-valued logic". In G. Epstein and J. M. Dunn, (eds.), *Modern Uses of Multiple-Valued Logic*, pp. 8–37. D. Reidel, Dordrecht, Netherlands, 1977.

[BF94]      P. Baumgartner and U. Furbach. "PROTEIN: A PROver with a Theory Extension INterface". In Bundy [Bun94], pp. 769–773.

[Bun94]     A. Bundy, (ed.). *Proc. 12th Intl. Conf. Automated Deduction, Lecture Notes in Artifical Intelligence* **814**, Nancy, June/July 1994. Springer.

[BvH+90]    A. Bundy, F. van Harmelen, C. Horn, and A. Smaill. "The Oyster-Clam System". In M. E. Stickel, (ed.), *Proc. 10th Intl. Conf. Automated Deduction, Lecture Notes in Computer Science* **449**, pp. 647–648, Kaiserslautern, July 1990. Springer.

[CC77]      P. M. Cousot and R. Cousot. "Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints". In *Proc. 4th Annual ACM Symp. Principles of Programming Languages*, pp. 238–252, Los Angeles, California, January 1977. ACM Press.

[Daw91]     J. Dawes. *The VDM-SL Reference Guide*. Pitman, London, 1991.

[Fis97]     B. Fischer. *A systematic approach to type-based software component retrieval*. PhD thesis, TU Braunschweig, 1997. In preparation.

[GL+94]     C. Goller, R. Letz, K. Mayr, and J. M. P. Schumann. "SETHEO V3.2: Recent Developments—System Abstract". In Bundy [Bun94], pp. 778–782.

[Jac94]     D. Jackson. "Abstract Model Checking of Infinite Specifications". In M. Naftalin, T. Denvir, and M. Bertran, (eds.), *Proc. 2nd Intl. Symp. Formal Methods Europe, Lecture Notes in Computer Science* **873**, pp. 519–531, Barcelona, October 1994. Springer.

[JM94]      C. B. Jones and K. Middelburg. "A Typed Logic of Partial Functions Reconstructed Classically". *Acta Informatica*, **31**(5):399–430, 1994.

[KRT87]     S. Katz, C. A. Richter, and K. S. The. "PARIS: A System for Reusing Partially Interpreted Schemas". In *Proc. 9th Intl. Conf. Software Engineering*, pp. 377–385, Montery, CA, March 1987. IEEE Computer Society Press.

[Kru92]     C. W. Krueger. "Software Reuse". *ACM Computing Surveys*, **24**(2):131–183, June 1992.

[LMG94]     R. Letz, K. Mayr, and C. Goller. "Controlled Integration of the Cut Rule into Connection Tableau Calculi". *J. Automated Reasoning*, **13**:297–337, 1994.

[LS+92]     R. Letz, J. Schumann, S. Bayerl, and W. Bibel. "SETHEO: A High-Perfomance Theorem Prover". *J. Automated Reasoning*, **8**(2):183–212, 1992.

[McC94]     W. W. McCune. A Davis-Putnam Program and its Application to Finite First-Order Model Search: Quasigroup Existence Problems. Technical report, Argonne, IL, USA, 1994.

[MI⁺97]   M. Moser, O. Ibens, R. Letz, J. Steinbach, C. Goller, J. Schumann, and K. Mayr. "The Model Elimination Provers SETHEO and E-SETHEO". *J. Automated Reasoning*, 1997. To appear.

[Mid93]   K. Middelburg. *Logic and Specification — Extending VDM-SL for advanced formal specification*. Computer Science: Research and Practice. Chapman & Hall, 1993.

[MM91]   P. Manhart and S. Meggendorfer. "A knowledge and deduction based software retrieval tool". In *Proc. 4th Intl. Symp. on Artificial Intelligence*, pp. 29–36, 1991.

[MMM94]  A. Mili, R. Mili, and R. Mittermeir. "Storing and Retrieving Software Components: A Refinement-Based System". In B. Fadini, (ed.), *Proc. 16th Intl. Conf. Software Engineering*, pp. 91–102, Sorrento, Italy, May 1994. IEEE Computer Society Press.

[MS89]   Y. S. Maarek and F. A. Smadja. "Full Text Indexing Based on Lexical Relations An Application: Software Libraries". In *Proc. 12th Annual Intl. ACM SIGIR Conf. on Research and Development in Information Retrieval*, pp. 198–206, 1989.

[MW95]   A. Moorman Zaremski and J. M. Wing. "Specification Matching of Software Components". In G. E. Kaiser, (ed.), *Proc. 3rd ACM SIGSOFT Symp. Foundations of Software Engineering*, pp. 6–17, Washington, DC, October 1995. ACM Press.

[Per87]   D. E. Perry. "The Inscape Environment". In *Proc. 11th Intl. Conf. Software Engineering*, pp. 2–12. IEEE Computer Society Press, May 1987.

[RW91]   E. J. Rollins and J. M. Wing. "Specifications as Search Keys for Software Libraries". In K. Furukawa, (ed.), *Proc. 8th Intl. Conf. Symp. Logic Programming*, pp. 173–187, Paris, June 24-28 1991. MIT Press.

[Sch96]   J. M. P. Schumann. "SiCoTHEO: simple competitive parallel theorem provers". In M. A. McRobbie and J. K. Slaney, (eds.), *Proc. 13th Intl. Conf. Automated Deduction*, *Lecture Notes in Artifical Intelligence* **1104**, pp. 240–244. Springer, July/August 1996.

[Sla94]   J. Slaney. "FINDER: Finite domain enumerator". In Bundy [Bun94], pp. 798–801.

[SM83]   G. Salton and M. J. McGill. *Introduction to Modern Information Retrieval*. McGraw-Hill, New York, 1983.

[ZS95]   M. Zand and M. Samadzadeh. "Software Reuse: Current Status and Trends". *J. Systems and Software*, **30**(3):167–170, September 1995.