Solving Software Reuse Problems with Theorem Provers

Thomas Baar[1] and Bernd Fischer[2]

[1] Inst. f. Mathematik, HU Berlin
baar@mathematik.hu-berlin.de
[2] Abt. Softwaretechnologie, TU Braunschweig
fisch@ips.cs.tu-bs.de

Abstract. In NORA/HAMMR, we investigate the application of automated theorem provers to retrieve software components based on their formal specifications. The problem profile has major impacts on the problem solving process. Integration and preprocessing steps, e.g., simplification, become as important as the actual proving process.
NORA/HAMMR thus uses a pipeline of filters of increasing deductive strength. Only in the final filter provers are applied. Here, we use ILF to control competition between different systems. Experiments confirm this approach. With moderate timeouts we already achieve an overall recall of approximately 80%.

1 Introduction

Progress in automated deduction has made the application of automated theorem provers (ATPs) to problems in software engineering a more realistic idea than ever before. With NORA/HAMMR (cf. [5] for a detailed account) we investigate an application in software reuse, *deduction-based software component retrieval*. It uses formal specifications as component indexes and as queries, builds proof tasks from these, and checks the validity of the tasks using an ATP. A component is retrieved if the prover succeeds on the associated task—retrieval becomes a deductive problem.

Solutions of this deductive problem, however, are constrained by peculiarities of its software engineering roots which set it apart from other applications domains, e.g., mathematics:

- The users are no ATP experts; they are not even interested in successful proofs but only in retrieved components.
- Response times matter; from the user's point of view it is better to be fast than complete ("results-while-u-wait".)
- Every single user task spawns a large number of proof tasks.
- If a task is provable, its proof is rather simple but in most cases it is unprovable (i.e., no valid theorem) because it belongs to a non-match.

The different user and problem profiles have major impacts on a realistic implementation of deductive retrieval. First, the deductive component must be encapsulated completely. The "novice" users must be able to formulate their problems in their own, application-oriented language (e.g., NORA/HAMMR uses VDM-SL). Thus, an efficient and automatic construction of prover-specific tasks becomes an important part of the problem-solving process. Then, the time requirements and the large number of tasks render a naïve generate-and-test approach infeasible. Instead, more intelligent architectures are required which prevent the actual ATP from "drowning." Finally, simplification of the proof tasks and detection and removal of non-theorems can no longer be taken for granted and must be done explicitely.

These requirements prompt an open system architecture (cf. Section 3.1), in which different deductive components work in combination on a practical application which is too difficult for a single monolithic system. However, the single components still require a substantial amount of system tuning which must be done by an "expert user" or *reuse administrator*. For this process (cf. Section 5), *interactive* theorem proving systems with good presentation and prototyping facilities as for example the ILF-system proved to be suitable.

2 Application Background

Component retrieval is one of the technical key issues in software reuse: "You must find it before you can reuse it!"[1] A variety of different approaches has been investigated, deduction-based retrieval being the most ambitious (cf. [9] for an overview.) In contrast to the other approaches, it exploits exact semantic information about the components and retrieves proven matches only. Its basic idea is very simple.

1. Each component c is associated with a *contract*, a formal specification which captures the relevant behaviour in form of a pre- and postcondition, e.g.,

   ```
   run (l : list) r : list
   pre  true
   post exists l1 : list & l = r ⌒ l1 ∧ ordered(r)
      ∧ forall i : item, l2 : list & l = r ⌒ [i] ⌒ l2 ⇒ ¬ordered(r ⌒ [i])
   ```

   which computes the longest ordered initial subsegment (i.e., run) of a list.[2]
2. Contracts also serve as queries q, e.g.,

   ```
   segment (l : list) r : list
   pre  true
   post exists l1, l2 : list & l = l1 ⌒ r ⌒ l2
   ```

   can be used to retrieve any function which returns an arbitrary continuous sublist of the argument.

[1] *The First Golden Rule of Software Reuse*, attributed to W. Tracz.

[2] In VDM-SL, ⌒ denotes list concatenation, [ ] the empty list, [i] a singleton list with item i. & reads as "such that" and ordered is a user-defined predicate.

3. For each possible candidate, a proof task is constructed comprising the respective pre- and postconditions.
4. A component qualifies if an ATP can establish the validity of the associated task.

The exact form of the proof task determines the nature of the reuse. The most common form is *plug-in compatibility*

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

which supports black box reuse—retrieved components may be reused "as is", without further proviso or modification. Other notions of compatibility support white box reuse but then manual checks or code modifications are required in order to guarantee the applicability of the retrieved components.

3 The Deductive Infrastructure

3.1 System Architecture

The key problem in deduction-based software retrieval is to maintain a balance between fast responses and high recall (i.e., number of proofs found.) The large number of tasks makes it also a hard problem. Thus, a architecture is required which prevents the actual ATP from "drowning". NORA/HAMMR uses a pipeline of filters of increasing deductive strength in order to reduce the number of proof problems stepwise. Several prefilters based on signature matching and rewriting try to identify non-matches as fast and early as possible and only for the remaining proof problems a real ATP is started.

Yet, all experiments show that still no single ATP on its own is powerful enough to be "the" deductive component for all the tasks passing the prefilters. NORA/HAMMR thus integrates different ATPs, using the ILF-system [4] as an "ATP-scheduler" to control them. ILF provides easy access to the resources of an entire local computer network for time-consuming proof attempts and obviates the necessity to generate specific input-files for every ATP. This allows us to use different *methods* for the same problem in parallel. Currently, we use resolution (OTTER and SPASS) and tableau style systems (SETHEO.) Even a proper combination of methods following the TECHS-approach [3] is supported. Further parallelization is achieved along another dimension. NORA/HAMMR can generate different *variants* of the same problem, e.g., using different axiom sets which are handled by ILF in the same way.

Figure 1 shows the resulting system architecture. Users communicate only with NORA/HAMMR, using a simple graphical user interface described in [5]. The tasks are piped through the different pre-processing stages provided by NORA/HAMMR. At the end of the pipeline, ILF takes over control and dispatches the tasks to the ATPs. Since the users need no proofs, ILF just returns whether a proof has been found at all, and NORA/HAMMR eventually displays the component.

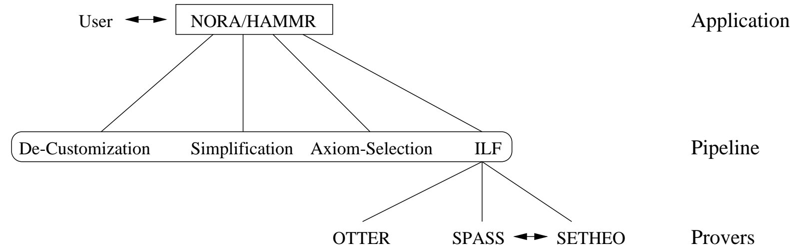

Fig. 1. System architecture

3.2 De-Customization

VDM-SL offers a wide variety of syntactic constructs, e.g., *let*-expressions, pattern matching, built-in datatypes, dynamic types using type invariants and many more. The process to cut this down is called de-customization. It translates the proofs tasks into LPF, the logic of partial functions [1] which we use as core language. De-customization replaces binding expressions on the term level (e.g., *let*- or *cases*-expressions) by standard quantifiers such that non-deterministic expression evaluation (due to VDM-SL's loose semantics) and undefined expressions (due to partial functions) are mapped correctly (cf. [8].) It also eliminates dynamic types and replaces them with their static super-types by relativization with the type invariants, similar to the standard relativization technique [10].

A second step takes care of the partial functions and translates LPF into FOL, following [7]. The translation is provability-preserving, i.e., $\vdash_{\text{LPF}} \varphi \iff \vdash_{\text{FOL}} \varphi'$ holds. It uses a set of signed functions to map any LPF-formula which contains an undefined subterm to an unprovable FOL-formula. E.g., the LPF-formula $\forall l : \mathit{List} \cdot \mathit{hd}\ l = \mathit{hd}\ l$ which has the truth value *undefined* becomes $\forall l : \mathit{List} \cdot l \neq [\,] \wedge \mathit{hd}\ l = \mathit{hd}\ l$. Since the quantifiers in LPF range only over proper (i.e., defined) values, we can optimize the handling of formulas and terms which contain no occurrences of partial functions.

The original translation by Jones and Middelburg uses *infinitary* logic to deal with recursively defined datatypes. Since we translate only into pure FOL but do not apply proper inductive provers, we need first-order approximations for those datatypes. This approximation comprises two steps.

In the first step, the free generation property of the datatype is encoded by additional first-order axioms, similar to [6]. In detail, we have to encode (*i*) the constructor property of the constructor functions (i.e., that terms with different top-level constructors are never equal), (*ii*) the surjectivity of the constructors wrt. the datatype domain (i.e., that the top-level function symbol of each element in the domain is one of the constructor functions), and (*iii*) the freeness or injectivity of the constructor functions (i.e., if two terms with the same top-level constructor are equal then their respective arguments are equal, too). Although these axioms do not capture the finite generation property, they work quite well in practice. For example, in the usual theory of lists which is freely generated

by *nil* and *cons*, the three properties give rise to the following axioms[3] $(i)\ \forall i : item,\ l : list \cdot nil \neq cons(i,l)$, $(ii)\ \forall l : list \cdot l = nil \vee \exists i : item,\ m : list \cdot l = cons(i,m)$, and $(iii)\ \forall i,j : item,\ l,m : list \cdot cons(i,l) = cons(j,m) \Rightarrow i = j \wedge l = m$.

However, we can even improve this and incorporate cardinality information which we can infer from the constructors and the signature information contained in the theory database. If a sort is *freely* generated by at least two constructors and all argument domains are guaranteed to be non-empty (e.g., because the signature contains constants of the necessary types), then we know that it contains at least two *different* elements. In the list example, we can thus add a fourth axiom $(iv)\ \forall l : list \cdot \exists m : list \cdot l \neq m$.

A second step deals with the induction scheme which follows from a datatype definition. Obviously, it cannot be encoded by first-order axioms. However, the special nature of our proof tasks allows the very powerful heuristic to use the formal parameter(s) of candidate component as induction variable(s) and to instantiate the induction scheme appropriately.

3.3 Simplification

Unlike the problems in benchmark collections as the TPTP [14], proof tasks in applications are generated automatically and thus not simplified. E.g., in our case they may still contain the propositional constants *true* and *false* from the original contracts or redundant equations which may be used to simplify the task. Hence, rigorous simplification is a necessary first step.

In NORA/HAMMR, we use a rewrite-based simplification procedure, and since we are working with extensions of FOL, the applied set of rewrite rules is two-tiered. The core tier deals with the FOL operators and equality. It eliminates the propositional constants, rewrites the tasks into conjunctive normal form and then further into anti-prenex form to minimize quantifier scopes.

The custom tier deals with all other symbols. It can also be separated into two subsets. One subset contains all rules which can be extracted from "suitable" axioms and lemmas in the database, i.e., universally closed unit literals, equations, and implications. Unit literals are rewritten into *true* or *false*, depending on their sign. For equations and implications we only check whether they decrease the size of the terms but do not use a proper termination ordering. The other subset follows from the generator information for datatypes. Of course, the constructor property and injectivity of the constructor functions induce the usual rewrite rules. The surjectivity gives rise to a *witness rule*, e.g., $\exists x : List \cdot x = t \rightsquigarrow true$ (provided that the bound variable x does not occur free in t.) Similarly, the cardinality information can be turned into rewrite rules. Note that both rules consider the quantifier as an ordinary operator symbol.

[3] The necessary sort information can easily be obtained from the function specifications in the theory database (cf. Section 3.5.)

3.4 Rejection

Simplification can also be used in a rejection filter: if a proof task $\mathcal{G}$ can be simplified to *false*, the candidate may obviously be rejected. Unfortunately, only very few of the inherent inconsistencies can already be detected by the simplifications so far. For rejection purposes it is necessary to make much more of them explicit. To this end, we can again exploit the generator information for datatypes and use the surjectivity of the constructor functions to "unroll" sorted quantifiers, e.g., $\forall l : \mathit{list} \cdot H[l]$ becomes $H[\mathit{nil}] \wedge \forall i : \mathit{item}, l : \mathit{list} \cdot H[\mathit{cons}(i,l)]$. By repeated unrolling and re-simplification we are then able to detect almost half of the mismatches.

Even though this rewrite-based simplification is a good low-cost rejection filter, it is still too coarse and more methods to show $\mathcal{A} \not\models \mathcal{G}$ formally are necessary. The obvious approach is to negate the goal and to check $\mathcal{A} \vdash \neg\mathcal{G}$. However, this is only a sufficient and not a necessary condition and in many cases we have that $\mathcal{A} \not\models \mathcal{G}$ and $\mathcal{A} \not\models \neg\mathcal{G}$ both hold.

Another approach is to look for explicit countermodels, i.e., structures in which the axioms $\mathcal{A}$ hold but not $\mathcal{G}$. We have experimented with model checking techniques (cf. [13]) but since $\mathcal{A}$ includes the theory of lists, we can only approximate the necessary finite structures and the approach becomes unsound. However, as humans we can spot the countermodels easily because usually only a small part of the structure is required. Moreover, this part is even quite similar for most tasks. Hence, in order to show $\mathcal{A} \not\models \mathcal{G}$, we formalize the countermodel by additional axioms CM and try to deduce the negated goal, i.e., we have to solve the task $\mathcal{A} \cup \mathit{CM} \vdash \neg\mathcal{G}$. This approach relies of course on the fact that the extension CM is consistent with the original axioms $\mathcal{A}$. However, this cannot be proven automatically but must be shown manually by the reuse administrator.

3.5 Axiom Selection

The proof tasks contain a variety of extra-logical symbols which need to be axiomatized by the reuse administrator. NORA/HAMMR provides a theory description kernel which resembles in some ways a logical framework, e.g., Isabelle [11]. The main difference is that it does not support the specification of new logics but only of conservative or inductive extensions of order-sorted FOL or *theories*. The application of such a dedicated theory description language is nevertheless worthwhile because it explicitly captures meta-information which is essential for many specialized techniques and which cannot easily extracted automatically from a flat list of FOL-formulas.

A theory description comprises a set of sort, function, and predicate declarations together with axioms, lemmas, and rules which describe properties of the declared symbols. Theories are hierarchically ordered by the extension relation in the same way modules are ordered by the import relation. The example theory `TList`

```
theory TList = FOL +
  classes CListEq :: CEq
```

```
types "List" :: CListEq;
      "Nil"  <  "List"
```

directly extends the base theory `FOL`. It introduces the class (i.e. collection of sorts) `CListEq` of list sorts with equality as a subclass (i.e. subcollection) of the general equality class `CEq`. `CListEq` comprises the sort `List` and a `Nil`-subsort.

Based on these domains, predicates and functions can be declared. The theory kernel supports different operator fixities and priorities as well as variable arity operators. For the list example, typical declarations are

```
consts "nil" : "Nil"                      (0);
       "#"   : "[Item; List] => List"     (infix 2 45);
       "^"   : "[List; List] => List"     (infix 2 45);
       "mem" : "[Item; List] => o"        (2)
```

which introduce a `nil`-constant, two binary infix operators `#` (cons) and `^` (append) with priority 45 and a nonfix binary predicate `mem` (membership), respectively.

Properties of these symbols can be specified in different ways. As usual, arbitrary FOL-formulas can be used but the kernel allows a distinction between proper axioms and lemmas where it is assumed (but not checked) that the lemmas are inductive consequences of the axioms, e.g.,

```
axioms
  memDef "forall I:Item . forall L:List .
            mem(I,L) <-> exists L1:List . exists L2:List . L = L1 ^ (I # L2)"
lemmas
  memNil "forall I:Item . ~ mem(I, nil)"
```

The kernel also provides explicit notations for properties which are exploited by other steps, e.g., associative-commutative operators or freely generated datatypes:

```
"List" freely generated by "nil", "#";
```

The large number of axioms and lemmas contained in a theory database requires a reduction mechanism which selects only those which are necessary to find a proof at all or are likely to shorten it and omits all those which only increase the search space.

In NORA/HAMMR, we use signature-based heuristics similar to that of Reif and Schellhorn [12]. Their basic assumption is that rules are redundant if they contain no symbols which occur in the problem, or more precisely, if they are defined in redundant theories. A theory is redundant if it introduces only symbols not occuring in the problem and is not referred (directly or indirectly) by other non-redundant theories.

Due to the distinction between axioms and lemmas the strategy of Reif and Schellhorn can be modified in several ways, e.g., (*i*) select only axioms, (*ii*) additionally, select lemmas if they contain only symbols which occur in the original problem, (*iii*) additionally, select lemmas if they contain at least one symbol which occurs in the original problem but no symbol from non-redundant theories, or (*iv*) select all axioms and lemmas from non-redundant theories. NORA/HAMMR currently implements the variants (*i*) and (*iv*).

4 Experiments

We used a library of 119 specifications of list processing functions. Approximately 75 of them describe actual functions (e.g., *tail*, *rotate*, or *delete_minimal*) while the rest simulates queries. We thus included under-determined specifications (e.g., the result is an arbitrary front segment of the argument list) as well as specifications which do not refer to the arguments (e.g., the result is not empty). We then cross-matched each specification against the entire library, using plugin-compatibility as match relation. This yielded a total of 14161 proof tasks where 1839 or 13.0% were valid.

The theory database used in the experiments comprises 65 theories, in which 24 different function and predicate symbols are axiomatized. The axiomatization consists of 38 core axioms and approximately 100 additional lemmas which are (first-order or inductive) consequences of the axioms.

We then used the rewrite-based methods (cf. Section 3.3) to detect and filter out obvious (mis)matches. We thus ruled out up to 6663 (47.1%) of the tasks as invalid and another 858 (6.1% or 46.1% of the valid problems) as trivial.

In a first experiment, we used the axiom selection mechanisms to generate three different variants of the proof tasks. SPASS was able to solve between 933[4] and 1089 of the matches (50.7%–59.2%) within one second, depending on the variant. With an increased time-out of 60 secs., the numbers grew to 67.9%–71.8%. As expected, competition between the variants significantly increased the recall, by approx. 7.5%. For short timeouts we even observed a “superlinear” growth. E.g., for a timeout of 10 secs., competition between all three variants solved 3.2% more problems than the best variant with a timeout of 30 secs. At the same time, the total elapsed runtime dropped by approx. 6%.

In a second experiment, we tested competition between the different provers OTTER, SETHEO, and SPASS. For a small but representative subset we achieved even better results—up to 56% compared to the best single system. Remarkably, none of the provers is “subsumed” by another as each solved at least one problem exclusively.

5 Reuse Administration using ILF

NORA/HAMMR provides some general preprocessing methods, e.g., axiom selection and rewriting mechanisms, and offers, in connection with ILF, an open system architecture which allows for the easy integration of further deductive components. However, their combination results in an accepted retrieval tool only after some domain-specific tuning of the entire system.

Since we consider the ATPs essentially as black boxes, we concentrate on *problem tuning*, e.g., through additional lemmas or development of better simplification methods. This requires an experimental testbed which offers

- translation of the proof tasks generated by the application system into a human readable form,

[4] All numbers include the trivial 858 matches detected by simplification.

- translation of example proofs found by an ATP into a human readable form,
- prototyping of user-defined methods which exploit the task structure, and
- good experimental support to gather statistical data and evaluate the methods.

Our experience has shown that ILF is an excellent testbed and, especially, that the combination of its presentation and prototyping facilities is very useful. The former allows the detection of simplification potential, the latter allows the exploitation of this potential. If a prototyped method turns out to be useful in the experiments, it can be integrated into the system. This feedback from ILF to NORA/HAMMR improves its overall performance.

However, "novice" users of NORA/HAMMR never interact with ILF—its application as experimental testbed is restricted to the reuse administrator. His skills must be exploited to achieve better results when the automated methods and their combinations are exhausted.

We used reuse administration to develop better rejection methods. A special property of the generated unprovable tasks is that in most cases only a few additional countermodel axioms given by the reuse administrator allow a formal refutation of the actual goal by an ATP (cf. Section 3.4.) Fortunately, the same set of axioms allows to dis-proof a large number of tasks. After inspection of some failed dis-proof attemps, it turned out that the necessary axioms are rather simple and consistent with the other theory of lists, e.g., $a > b$ or $memberP(cons(b, cons(a, nil)), b)$ for some new constants a and b.

Through proof task inspection we discovered that some complicated subformulas occured in many goals, sometimes even more than once, e.g., $\exists l : list \cdot app(l, cons(x, nil)) = y$. Such formulas can be replaced by simpler terms (e.g., $last(x, y)$) before the ATP is started if the appropriate axioms as $\forall x \forall y \cdot last(x, y) \leftrightarrow \exists z \cdot app(z, cons(x, nil)) = y$) are added to the task. Because the axioms are conservative extensions, this *definitorial folding* does not change the semantics of a theory.

Both methods (i.e., countermodel axiomatization and folding) can be combined, if suitable lemmas for the defined predicates are added. This combination improves the results considerably—almost 95 % of the non-matches which remain after rewrite-based rejection can be dis-proved if the tasks are simplified according the sketched approach.

6 Conclusions

In this paper, we described the application of ATPs to solve a problem in software reuse, the retrieval of components based on their formal specifications. Paradoxically, the key success factor of our system NORA/HAMMR is that it defers the application of ATPs as far as possible.

The problem profile makes it necessary to invest much effort in preprocessing steps, e.g., logic conversion, simplification, or detection of non-theorems. These steps require domain-specific (i.e., depending on the particular component library) tuning. Here, we use the presentation and prototyping facilities of ILF.

Experiences gained with this interactive use of ILF can then be fed back into NORA/HAMMR and used to optimize the application of automated systems.

On the actual deductive level, our main method of attack is *competition*, both between different task variants and between different ATPs. Here, we use the ILF-system to control the provers. Our results show this attack is successful: competition increases the recall rates considerably, by up to 50% compared to single systems. Currently, we thus achieve an overall recall of approximately 80% with moderate timeouts which indicates that deduction-based retrieval may become feasible with the next hardware generations.

References

[1] H. Barringer, J. H. Cheng, and C. B. Jones. "A Logic Covering Undefinedness in Program Proofs". *Acta Informatica*, **21**(3):251–269, October 1984.

[2] W. Bibel and P. H. Schmitt, (eds.). *Automated Deduction - A Basis for Applications.* Kluwer, Dordrecht, 1998. To Appear.

[3] J. Denzinger and D. Fuchs. "Enhancing conventional search systems with multi-agent techniques: a case study". In *Proc. Intl. Conf. on Multi Agent Systems (ICMAS'98)*, 1998. To Appear.

[4] B. I. Dahn, J. Gehne, T. Honigmann, and A. Wolf. "Integration of Automated and Interactive Theorem Proving in ILF". In *Proc. CADE-14*, *LNAI* 1249, pp. 57–60, Springer, 1997.

[5] B. Fischer, J. M. P. Schumann, and G. Snelting. "Deduction-Based Software Component Retrieval". In Bibel and Schmitt [2]. To Appear.

[6] J. Harrison. "Inductive definitions: automation and application". In *Proc. 8th Intl. Workshop on Higher Order Logic Theorem Proving and Its Applications*, *LNCS* 971, pp. 200–213. Springer, 1995.

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

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

[9] A. Mili, R. Mili, and R. Mittermeir. "A Survey of Software Reuse Libraries". *Annals of Software Engineering*, 1998. To appear.

[10] A. Oberschelp. "Untersuchungen zur mehrsortigen Quantorenlogik". *Mathematische Annalen*, **145**:297–333, 1962.

[11] L. C. Paulson. *Isabelle: A Generic Theorem Prover*, *LNCS* 828. Springer, 1994.

[12] W. Reif and G. Schellhorn. "Theorem Proving in Large Theories". In Bibel and Schmitt [2]. To Appear.

[13] J. M. P. Schumann and B. Fischer. "NORA/HAMMR: Making Deduction-Based Software Component Retrieval Practical". In *Proc. 12th Intl. Conf. Automated Software Engineering*, pp. 246–254, Lake Tahoe, November 1997.

[14] G. Sutcliffe, C. B. Suttner, and T. Yemenis. "The TPTP Problem Library". In *Proc. CADE-12*, *LNCS* 814, pp. 252–266. Springer, 1994.