Journal of Universal Computer Science, vol. 5, no. 3 (1999), 52-72
submitted: 1/11/98, accepted: 8/3/99, appeared: 28/3/99

Integrating Deduction Techniques in a Software Reuse Application

Thomas Baar
Institut für Mathematik, Humboldt-Universität zu Berlin
baar@mathematik.hu-berlin.de

Bernd Fischer
RIACS, NASA Ames Research Center
fisch@ptolemy.arc.nasa.gov

Dirk Fuchs
Fachbereich Informatik, Universität Kaiserslautern
dfuchs@informatik.uni-kl.de

Abstract: We investigate the application of automated deduction techniques to retrieve software components based on their formal specifications. The application profile has major impacts on the problem solving process and requires an open system architecture in which different deductive engines work in combination because the proof problems are too difficult for a single monolithic system. We describe our system architecture, a pipeline of filters of increasing deductive strength, and concentrate on the final filter, in which theorem provers are applied. Here, we use the ILF-system as a control and integration shell to combine different provers. We support two different combination styles, competition and cooperation. Experiments confirm our 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. [Fischer, Schumann, and Snelting 98] 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 or program verification:

- 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 often 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 engine 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, its efficient translation and the automatic construction of prover-specific tasks become important parts 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". Simplification of the proof tasks and detection and removal of non-theorems can no longer be taken for granted and must be done explicitly.

These requirements prompt an open system architecture (cf. [Section 3.1]), in which different deductive engines work in combination on a practical application which is too difficult for a single monolithic system. We support two different combination styles, competition and proper cooperation following the TECHS concept [see Fuchs and Denzinger 97; Denzinger and Fuchs 98]. Both styles increase the success rate and require merely small internal changes in the engines itself. However, the single engines still require a substantial amount of system tuning which must be done by an "expert user" or *reuse administrator.* For this process (cf. [Section 6]), *interactive* theorem proving systems with good presentation and prototyping facilities as for example the ILF-system [Dahn et al. 97] 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 [see Moorman Zaremski and Wing 97; Cheng and Jeng 92; Schumann and Fischer 97] being the most ambitious (cf. [Krueger 92] for a general overview of software reuse and [Mili, Mili, and Mittermeir 98] for library issues). In contrast to the other approaches, it exploits exact semantic information on 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 behavior in form of a pre- and postcondition, e.g.,

   $\mathsf{run}\ (\mathsf{l} : \mathsf{list})\ \mathsf{r} : \mathsf{list}$
   $\mathsf{pre}\ \ \mathsf{true}$
   $\mathsf{post}\ \mathsf{exists}\ \mathsf{l1} : \mathsf{list}\ \&\ \mathsf{l} = \mathsf{r} \frown \mathsf{l1}\ \wedge\ \mathsf{ordered(r)}$
   $\quad \wedge\ \mathsf{forall}\ \mathsf{i} : \mathsf{item},\ \mathsf{l2} : \mathsf{list}\ \&\ \mathsf{l} = \mathsf{r} \frown [\mathsf{i}] \frown \mathsf{l2}\ \Rightarrow \neg\mathsf{ordered}(\mathsf{r} \frown [\mathsf{i}])$

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

   $\mathsf{segment}\ (\mathsf{l} : \mathsf{list})\ \mathsf{r} : \mathsf{list}$
   $\mathsf{pre}\ \ \mathsf{true}$
   $\mathsf{post}\ \mathsf{exists}\ \mathsf{l1},\ \mathsf{l2} : \mathsf{list}\ \&\ \mathsf{l} = \mathsf{l1} \frown \mathsf{r} \frown \mathsf{l2}$

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

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

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

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 this 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 engine for all the tasks passing the prefilters. NORA/HAMMR thus integrates different ATPs, using the ILF-system (cf. [Section 3.3]) as an "ATP-scheduler" to control them. ILF allows us to use different provers for the same problem in parallel. Currently, we use resolution- (OTTER and SPASS) and tableau-style systems (SETHEO). Even a proper cooperation of provers following the TECHS approach (cf. [Section 4]) is supported, e.g., between SPASS, DISCOUNT and SETHEO. 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.

The resulting system architecture is shown in [Fig. 1]. Users communicate only with NORA/HAMMR, using a graphical interface described in [Fischer, Schumann, and Snelting 98]. The tasks are piped through the 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.

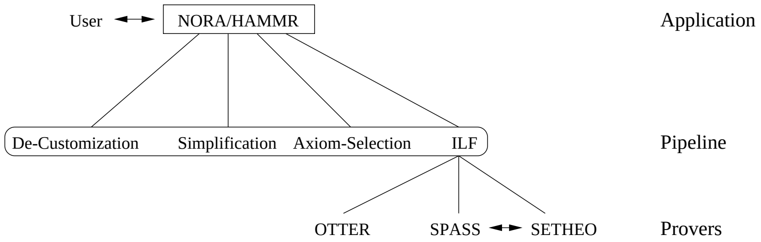

Figure 1: System architecture

3.2 Pipeline Elements

3.2.1 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 variety down is called de-customization. It translates the proofs tasks into LPF, the logic of partial functions [see Barringer, Cheng, and Jones 84] 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 [see Middelburg 93]. 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 [Oberschelp 62].

A second step takes care of the partial functions and translates LPF into FOL, following [Jones and Middelburg 94]. The translation is provability-preserving, i.e., $\vdash_{\mathrm{LPF}} \varphi \iff \vdash_{\mathrm{FOL}} \varphi'$ 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 $\forall l : List \cdot hd\ l = hd\ l$ which has the truth value *undefined* becomes $\forall l : List \cdot l \neq [\,] \wedge hd\ l = hd\ l$ which is unprovable in FOL. 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 [Harrison 95]. 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 a candidate component as induction variable(s) and to instantiate the induction scheme appropriately.

3.2.2 Simplification

Unlike the problems in benchmark collections as the TPTP [Sutcliffe, Suttner, and Yemenis 94], 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 equivalences. Unit literals are rewritten into *true* or *false*, depending on their sign. For equations and equivalences 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.

3.2.3 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

[3] The necessary sort information can easily be obtained from the function specifications in the theory database (cf. [Section 3.2.4]).

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., $\forall l : list \cdot H[l]$ becomes $H[nil] \wedge \forall i : item, l : list \cdot H[cons(i, l)]$. 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 [see Schumann and Fischer 97] 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 $\mathcal{A} \cup CM \vdash \neg\mathcal{G}$. 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.

3.2.4 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 [Paulson 94]. 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, L:List .
            mem(I,L) <-> exists L1:List, 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 operator or freely generated datatype:

```
"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 in [Reif and Schellhorn 98]. 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 occurring 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*).

3.3 ILF

The last element in the processing pipeline is the ILF-system (Integrating Logical Functions). It can, roughly speaking, be regarded as an integration and control shell for working with different ATPs.

The main input-output of ILF behavior is the solving of a first-order proof problem by giving the answers 'yes' or 'no' within a specific time limit. Due to the undecidability of the first-order logic, the answer 'no' does not mean that the goal does not follow from the theory. This is exactly the kind of behavior that all ATPs show if the search time for a proof attempt is restricted.

Internally, ILF converts the original proof problem into variants suitable for the supported state-of-the-art theorem provers, e.g. SETHEO, SPASS, OTTER, DISCOUNT, and launches these provers in parallel on a local computer network. If one of the provers can solve the problem within the time limit, ILF gives the answer 'yes', otherwise 'no'.

The sketched restricted functionality makes it easy to incorporate ILF into other systems because it can work as a shell script. However, originally ILF was developed for interactive work. In order to gain acceptance, it provides much more functionality than only saying 'yes' or 'no' and—as we can see in [Section 6]—this functionality is also very useful for our retrieval application.

The most striking difference to other existing verification tools is the presentation of theories and proofs from several automated provers in a human readable form. The possibility to inspect the feedback of the automated provers easily is extremely important in order to detect inconsistencies in the used theory. The detection of really abstruse proofs by a prover is an indication that the theory is inconsistent and every goal could be proved.

Another useful advantage is that ILF is highly customizable. It is possible to configurate ILF in such a way that it controls the proof attempts of all integrated theorem provers for hundreds of tasks and evaluates all results. This makes it feasible to obtain statistical data for a specific class of proof tasks.

4 Cooperation between ATPs

After we described the pipelining process and the ILF tool, which manages the ATPs for the emerging proof problems, we now investigate cooperation between ATPs. Our experiments (cf. [Section 5.2]) show that the success rate of ATPs can be increased if the provers work in a cooperative style.

We describe the underlying concept for cooperation in detail because the principles may also be interesting for other applications in which systems should interact and cooperate with each other. The general idea of our underlying TECHS (TEams for Cooperative Heterogeneous Search) approach is to interchange selected clauses between heterogeneous provers in regular time intervals.

4.1 Basics of TECHS

The TECHS approach requires several different provers running in parallel on different computing nodes. In our context of component retrieval problems proof problems are specified in first-order logic (with equality). The provers employ either calculi which are complete for first-order logic (*universal provers*) or calculi which are complete for a sub-logic of first-order logic (*specialized provers*). We allow for the use of saturation-based calculi, like resolution and superposition, and of analytic or tableau-style calculi, like the connection tableau calculus. We assume that unique numbers are assigned to the provers.

As already mentioned, the provers exchange information in regular time intervals. Thus, the working scheme of each prover is characterized by certain phases similar to the TEAMWORK approach [Denzinger 95]. The sequence of phases is

$P_{init}, P_w^0, P_c^0, P_w^1, P_c^1, \ldots$, i.e. after an initialization phase (P_{init}), working (P_w^i) and cooperation phases (P_c^i) alternate each other.

In the *initialization phase* P_{init} each prover j obtains the initial clause set $\mathcal{C}$ to be refuted and forms an initial search state S_j^0 out of this state. In each *working phase* P_w^i a prover j starts with an initial search state S_j^i and transforms this search state with its inference rules into successor states $S_j^i = S_{j,0}^i \vdash \ldots \vdash S_{j,n_j^i}^i$.

In a *cooperation phase* P_c^i, the provers exchange clauses which are proven to be logic consequences of $\mathcal{C}$. The information exchange takes place *synchronously*. Each prover j can extract clauses from the search states $\{S_{j,0}^i, \ldots, S_{j,n_j^i}^i\}$ enumerated in the working phase P_w^i. Note that—especially for an analytic prover—this is more sensible than simply extracting clauses from $S_{j,n_j^i}^i$. Since such provers usually perform a search with backtracking it is sensible to consider also experiences made during the search. Obviously, a prover should not communicate too many clauses to others. This would entail negative consequences for both saturation-based and analytic provers. A receiving saturation-based prover like SPASS must process all these clauses (i.e. perform all possible inferences involving them) which can take a rather long time. Moreover, since these clauses persist in the search state they might slow down the prover because they can be used in many inferences in future. An analytic prover like SETHEO can also not profit from too many clauses since they increase the branching rate of the search tree too much. Furthermore, it is wise to send only a small number of clauses so as to decrease the amount of communication. Hence, we decided to exchange only a subset of the clauses which is selected by using so-called *referees*.

All in all, in a cooperation phase P_c^i following activities are performed by each prover j:

- Extraction of clauses from the search states $\{S_{j,0}^i, \ldots, S_{j,n_j^i}^i\}$ of working phase P_w^i.
- Selection of a fixed number of clauses via referees and exchange of these clauses with selected clauses of other provers.
- Building a new start state S_j^{i+1} for working phase P_w^{i+1} considering the clauses received from other provers and the current search state $S_{j,n_j^i}^i$.

Note that a team based on the TECHS approach can easily be integrated into the ILF-system. It is only necessary that ILF launches the provers and gives them information on the proof problem and their cooperation partners. After that, the provers can cooperate independently of ILF.

4.2 General TECHS Architecture

When selecting clauses from a prover i which should be sent to prover j, it is sensible to utilize as much knowledge as possible in the selection process so as to perform an "optimal" selection regarding i and j. Further, the selection and transmission of clauses should be very efficient in order to reduce the overhead caused by the cooperation. However, these are conflicting goals: if we for example merely use *local knowledge* for selecting clauses, i.e. knowledge about the current search state of the sender and its history, we can select and transfer clauses very

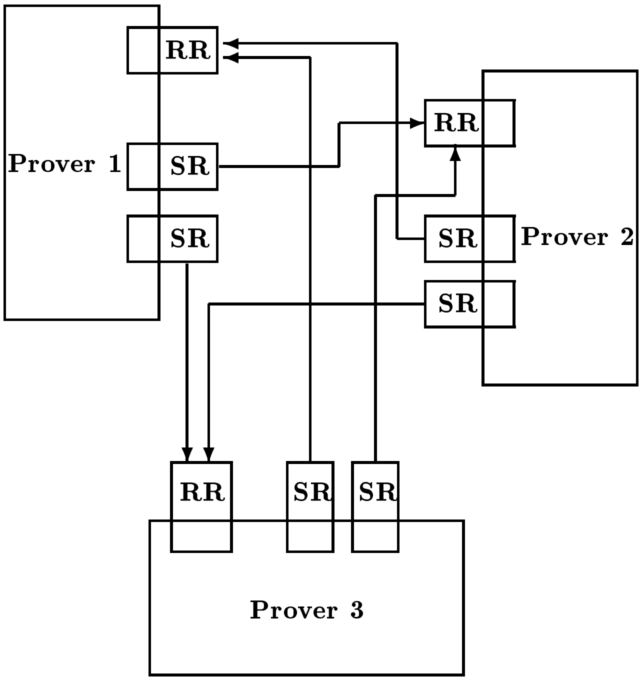

Figure 2: TECHS architecture for 3 provers

efficiently. We can select clauses at the site of the sender and send these clauses via broadcast to all receivers. However, concrete *needs* of receiving provers are not considered. The other extreme is to select clauses according to *global knowledge* about the sender and the receiver. If we have so much knowledge it is possible to perform an optimal selection regarding i and j. But this kind of selection requires the highest (communication) costs because the complete search states of sender and receiver (and probably their history) must be simultaneously evaluated.

Thus, we have chosen an approach that realizes a compromise between the use of local and global knowledge or success-driven and demand-driven selection. We employ an individual *send-referee* for each receiver of clauses at the sender site which selects clauses from the sender. Moreover, each receiver employs an additional *receive-referee* which filters some clauses from the set of clauses it receives from the send-referees of other provers. This receive-referee takes needs of the receiver into account. Thus, on the one hand send- and receive-referee can use knowledge about the sender and the receiver of clauses, respectively. On the other hand, the selection process is rather efficient: In the case that our team consists of n provers, $n-1$ selections take place at each sender site, then $n-1$ (rather small) sets of clauses must be transferred to the receiving provers, and then one additional selection out of the incoming clauses must be performed.

The architecture of a system based on TECHS is depicted in [Fig. 2]. The send-referees (SR) and receive-referees (RR) are displayed half inside and half outside the provers because they can be realized either as parts of the provers or as independent processes. Realizing referees as parts of the provers requires more implementational effort but allows to have access to internal data of the provers. Since this allows the development of more powerful referees we decided to choose this alternative for our experiments.

In order to exchange clauses between referees it is important that they employ a communication language that both send- and receive-referees understand. Consider the situation that a send-referee obtains clauses in format i and its associated receive-referee has to give some clauses to a prover which employs format j. Then, it is possible to implement receive-referees that understand many different formats of clauses and can transform them into the relevant format j. Another possibility is to employ a specific transfer language: Send-referees transfer their selected clauses from format i into this transfer language, receive-referees then transform received clauses into format j. The main disadvantage of the latter method is that two transformations are needed. However, it has the advantage that a new prover can simply be added to the cooperative system because its send- and receive-referee must only understand two clause formats. We used this second alternative and employed the syntax format of the *DFG Schwerpunkt Deduktion* [see Hähnle, Kerber, and Weidenbach 96] for exchanging clauses.

4.3 Achieving Cooperation between Provers

In the following, we describe the activities which are necessary for achieving cooperation between theorem provers. These are the extraction of clauses, the selection with send- and receive-referees, and the integration of extracted clauses.

4.3.1 Extraction of Clauses

The first step of an exchange of information is the extraction of clauses from the search states $\{S^i_{j,0}, \ldots, S^i_{j,n^i_j}\}$ a prover j has enumerated in the working phase P^i_w. We distinguish between saturation-based and analytic provers.

The extraction of valid clauses from search states of saturation-based provers is very easy because search states are sets of clauses. Even more, a theorem prover j can take all interesting clauses to be selected from the *actual search state* S^i_{j,n^i_j} at the beginning of cooperation phase P^i_c. This is because the provers only derive new clauses, delete unnecessary clauses, or simplify clauses. Hence, the sequence of enumerated search states (clause sets) is not needed but it is sufficient to consider the most recently derived clause set.

The analytic provers that we consider are based on the connection tableau calculus and conduct a search with iterative deepening and backtracking in the search tree $\mathcal{T}$ of all connection tableaux. A single search state is only one of these tableaux. Hence, it is only possible to extract information on *one* proof attempt from a search state. If the proof attempt represented by the search state does not lead to a proof it might be that all clauses which can be extracted from it are unnecessary. Then, send-referees might have to choose from a pool consisting of unnecessary clauses only. This shows that clauses must be extracted from *all* search states enumerated during a working phase, i.e. they must essentially be extracted from the search tree of all tableaux.

Because of the fact that the search tree $\mathcal{T}$ of all tableaux is in general not given explicitly, clauses must be extracted from the enumerated tableaux during the search process. A possible method to do this is to utilize *bottom-up lemma mechanisms* [see Letz, Mayr, and Goller 94] of connection tableau-based theorem provers. In addition, top-down lemmas, so-called subgoal clauses [see Fuchs 98b], might be extracted from a tableau. Before a send-referee is applied to the set of extracted lemmas it is reasonable to compute a *minimal* lemma set by eliminating subsumed clauses.

4.3.2 Realization of Send-Referees

A send-referee in a system based on the TECHS approach consists of a pair (S, φ) of a *filter predicate* S and a *selection function* φ. The prover that receives the results of the send-referee will get those clauses in the cooperation phases that pass through the filter and that are selected by φ.

The filter predicate S limits the set of clauses that are eligible for transmission. Typically, clauses are filtered out that are redundant w.r.t. the receivers. In the following those are clauses that a receiver cannot use in its inference mechanism (e.g. non-unit clauses are redundant for an equational prover).

The selection function φ can employ several judgment functions $\psi_1, \ldots, \psi_n$. These functions ψ_i associate a natural number with each clause C, and C is considered the better the higher the value $\psi_i(C)$ is. φ eventually selects the clauses with the best judgments.

The judgment functions that we have used consider on the one hand *general syntactic features of clauses.* E.g., they prefer general clauses which contain many variables and only few function symbols. These clauses are useful for saturation-based provers because they might often take part in contracting inferences (subsumption, rewriting). They are often also useful for analytic provers because open tableau branches can be closed. On the other hand we employ functions that *judge the derivation history of a clause* w.r.t. its possible usefulness for the receiver. E.g., if the receiver of a clause is an analytic prover and the sender a saturation-based prover, we consider whether the clause is the result of an equality inference like superposition. Such clauses are especially useful for analytic provers since they have difficulties with equality handling. Similar criteria can be developed for other combinations of sender and receiver.

Due to lack of space we cannot describe the referees in detail. For exact realizations of send-referees for saturation-based provers we refer to [Fuchs and Denzinger 97]. Exact descriptions of send-referees for analytic provers can be found in [Fuchs 98a; Fuchs 98b].

4.3.3 Realization of Receive-Referees

A receive-referee is a selection function φ which also uses judgment functions for measuring the quality of clauses. Receive-referees can only select clauses from the set of clauses they have received from the send-referees of other provers. Thus, they have to choose from a small set of clauses and can therefore employ more sophisticated (and time-consuming) criteria than a send-referee. Nevertheless, the knowledge about the receiver is somewhat limited and can only facilitate the selection a little bit. (In order to go beyond the limitation, for each clause the whole future proof and the role of the clause in it has to be computed, which is not feasible.) Since the receive-referee can only estimate which consequences the integration of certain clauses will have we must employ heuristic criteria again.

Our first judgment function judges whether a received clause may be *part of a proof* that can quickly be found. E.g., for superposition-based provers we judge whether many short clauses can be derived from a clause received and clauses from the actual clause set. In our approach, analytic provers employ receive-referees only if they receive clauses from saturation-based provers. There, we judge whether a received clause might contribute to close tableaux derived during the proof search. We simply compute the difference of the number of tableau branches which can be closed by a received clause and the number of branches that remain open. A second judgment function prefers clauses which need not be part of a proof but are nevertheless *able to decrease the search effort* in future. We use this function for saturation-based provers only. Our function produces higher values if many contracting inferences are possible with a clause. Again, exact definitions can be found in [Fuchs and Denzinger 97].

4.3.4 Integration of Clauses

The integration of clauses into the search state of a saturation-based prover is very simple since it works on sets of clauses [see Fuchs 98c]. Thus, a clause can be integrated by adding it to the current clause set and performing all inferences possible with it.

For the integration of clauses into the search state of an analytic prover there exist two possibilities. The first is to add the new clauses to the old axioms and to perform a re-start of the proof run. The second is to additionally avoid a re-start and to continue the search at the current choicepoint. The first variant has the disadvantage that the whole search of an analytic prover is lost and must possibly be repeated. However, the variant has the advantage that it can be implemented very easily. Moreover, in contrast to the second variant it guarantees that during the iterative deepening process a proof can be found in a minimal segment of the search space. Since the segments usually grow exponentially we have chosen the simple first variant.

4.4 Related Work

There are some related approaches that try to employ several provers for solving proof problems. Many of them are pure parallelization approaches and do not achieve cooperation by information exchange. For instance, the theorem prover PARTHEO [Letz and Schumann 90] is a parallel prover for the analytic connection tableau calculus. There exist also parallel versions of provers for saturation-based calculi as, e.g., PaReDuX [Bündgen, Göbel, and Küchlin 94].

Cooperation of provers by information exchange was so far mainly considered for homogeneous systems, i.e. systems where all provers employ the same calculus. For instance, the approaches TEAMWORK [Denzinger 95] and clause diffusion [Bonacina and Hsiang 95] are suitable for coupling different instances of a saturation-based prover. The prover CPTHEO achieves cooperation between different instances of a connection-tableau-based prover [see Fuchs and Wolf 98].

The closest relationship to our approach has the work done in [Sutcliffe 92] where a heterogeneous cooperation concept for automated deduction was proposed. The central concept was a distributed implementation of a shared memory (a so-called tuple-space), into which each agent wrote all formulas it generated. In contrast to our approach, no selection process was involved and no existing provers could be used. Therefore, the published experiments were not convincing.

5 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.

Although our experimental setup represents the most serious attempt to deduction-based software component retrieval so far, the library is still rather simple. The resulting proof tasks, however, are already rather hard, compared to standard theorem proving benchmarks, e.g., the TPTP. The tasks are not biased towards any particular prover because the component specifications have been completed before the experimental evaluation started; however, the applied logics (i.e., many-sorted first-order logic with equality) and the specification style should favor provers with built-in support for equality and sorts.

We expect that our approach easily scales up to other container datatypes, e.g., sets, bags, or graphs; obviously, more complicated specifications will lead to more complicated proof tasks. As all related work in deduction-based component retrieval, our approach is also essentially confined to components which can specified in a pure *input/output*-style, as for example functions, procedures, or methods; support for more elaborate component models (e.g. JavaBeans [Javasoft 98]) is currently planned.

5.1 Competition Experiments

With competition we denote that the ATPs work in parallel on basically the same problem but do not exchange information. We can distinguish two different competition modes which work along independent dimensions:

- variant competition: multiple identical instances of a single prover work on different task formulations of a problem.
- system competition: different provers or different instances of a prover (e.g., using different strategies or control parameters) work on the same proof task.

In NORA/HAMMR, we have experimented with both competition modes.

5.1.1 Variant Competition

Different preprocessing steps, e.g., simplification or axiom selection can give rise to quite different search spaces. Variant competition can then be used to exploit these differences.

In a first experiment we used OTTER to check four increasingly simplified variants of the proof tasks. All variants are obtained by application of the rewrite-based simplification procedure mentioned in [Section 3.2.2]. The respective results for different timeouts [4] are shown in [Tab. 1]. Here, *base* denotes the unsimplified base case, *FOL* the usual first-order simplifications, *subst* the additional substitution of equations $x = t$ throughout the task and *full* the application of the full custom tier of simplifications.

$T_{\max}$ (secs.)	base	FOL	subst	full	comp bf	comp all
1	749	711	729	952	970	979
10	1152	1147	1144	1247	1303	1319
30	1229	1215	1207	1282	1348	1363
60	1270	1263	1236	1293	1385	1397
90	1274	1267	1236	1293	1387	1399

Table 1: Results of variant competition experiments—simplification

This experiment clearly demonstrates the benefits of variant competition. The different competitive runs solve 7.27% and 8.20% more tasks than the best single variant; at the same time the elapsed real time drops by 13.52% to 15.80 %, resp., provided that each variant runs on an own processor. However, even for a single processor an improvement can be achieved, provided that the competition is restricted to the most complementary variants. E.g., for a timeout of 30 seconds *comp bf* solves 4.25% more tasks than the single best variant (*full*) within 60 seconds. Simultaneously, the elapsed total proof time drops by 1.73% which represents a *superlinear* speed-up.

In a second experiment, we used SPASS to check the effect of the different axiom selection mechanisms. The respective results for different timeouts[5] are shown in [Tab. 2]. Here, *core* and *lemmas* refer to variants where only axioms and axioms and additional lemmas, resp., have been selected, using the techniques described above while *full* refers to the base case containing the entire theory database. The *comp*-entries again refer to the different variant competitions.

As expected, the smaller search spaces induced by the *core* and *lemmas* selection mechanisms lead to a significantly (approx. 40%–45%) higher number of fast proofs which is especially important for our application. On the long run, however, too few lemmas are as bad as too much and both variants (and even their competition) solve dramatically fewer tasks than the *lemmas* variant. While this behavior is in accordance with the observations of [Reif and Schellhorn 98], the respective competition entries show that—unlike in the experiments of [Reif and Schellhorn 98]—in our case the *lemmas* variant does not completely subsume the other variants. However, the achieved improvements are rather small and in this case do not really warrant the higher computational efforts of the competition.

[4] All results were obtained using OTTER V3.0.5 in auto2-mode on a SUN UltraSPARC 170.

[5] All results were obtained using SPASS V0.90 on a SUN UltraSPARC 170.

$T_{\max}$ (secs.)	core	lemmas	full	comp cl	comp cf	comp lf	comp all
1	1040	1099	751	1149	1040	1099	1149
10	1162	1422	986	1453	1211	1423	1454
30	1171	1533	1063	1547	1249	1534	1548
60	1178	1561	1102	1576	1272	1568	1583
90	1180	1565	1142	1580	1292	1575	1590

Table 2: Results of variant competition experiments—lemma selection

5.1.2 System Competition Controlled by ILF

The experiments for system competition are based on a representative subset of the original library. We selected 24 components; in the resulting 576 tasks, the preprocessing methods integrated in NORA/HAMMR identified 23 provable and 336 unprovable tasks, which can be simplified to *true* and *false*, respectively.

For the remaining 217 tasks, the provers OTTER, SPASS, and SETHEO were started, each with a timeout of 240 secs. For SPASS and SETHEO we used different type-encoding techniques provided by ILF. Such techniques are needed to transform formulas from sorted logic into an unsorted logic.

	OTTER	$\text{SETHEO}_{\text{sub}}$	$\text{SETHEO}_{\text{te}}$	$\text{SPASS}_{\text{rel}}$	SPASS_{te}	comp
OTTER	**46**	25	14	4	12	–
$\text{SETHEO}_{\text{sub}}$	3	**24**	6	1	3	–
$\text{SETHEO}_{\text{te}}$	9	23	**41**	3	6	–
$\text{SPASS}_{\text{rel}}$	21	40	25	**63**	17	–
SPASS_{te}	13	26	12	1	**47**	–
comp	–	–	–	–	–	**70**

Table 3: Results for provable tasks within ILF

The results are shown in [Tab. 3]; SPASS_{te} and $\text{SETHEO}_{\text{te}}$ denote variants where a simple term-encoding technique was used (this was also used for OTTER), $\text{SPASS}_{\text{rel}}$ denotes a variant with standard predicate relativization technique, and $\text{SETHEO}_{\text{sub}}$ a more complicated term-encoding technique where the type-subtype relation is coded by a term-instance relation on the codeterms for types. Each prover (variant) is compared with every other one. For instance, the first row shows that OTTER solved 46 proof tasks, and of these 25 could not be solved by $\text{SETHEO}_{\text{sub}}$, 14 not by $\text{SETHEO}_{\text{te}}$, 4 not by $\text{SPASS}_{\text{rel}}$, and 12 not by SPASS_{te}. If all provers are run competetively, a total of 70 tasks can be solved, i.e., compared to the results of the best ATP, the recall rate can be increased significantly by 11%. As a further remarkable point, we observe that no prover variant is subsumed by another variant. Even for SPASS which has built-in support for sorts, the term encoding yields an additional proof.

However, in order to show formally for every case, whether the query matches or not, the remaining 147 tasks have to be shown unprovable. We thus started all provers on the negated goals and obtained even better results as in the "affirmative" case—competition can solve 56% more tasks than the best single system.

	OTTER	SETHEO$_{sub}$	SETHEO$_{te}$	SPASS$_{rel}$	SPASS$_{te}$	comp
OTTER	**41**	31	27	16	24	–
SETHEO$_{sub}$	7	**17**	7	9	12	–
SETHEO$_{te}$	13	17	**27**	12	14	–
SPASS$_{rel}$	14	31	24	**39**	14	–
SPASS$_{te}$	9	21	13	1	**26**	–
comp	–	–	–	–	–	**64**

Table 4: Results for unprovable tasks within ILF

5.2 Cooperation Experiments

We performed our experimental studies in the light of the same problems as in [Section 5.1.2]. All in all, we tackled 81 provable problems.

For our experimental study regarding cooperative provers we restricted ourselves so far to the universal provers SPASS and SETHEO, and the specialized equational prover DISCOUNT. Due to preliminary experimental results we restricted the information exchange of the cooperative team. We allowed for a bidirectional clause exchange between SPASS and DISCOUNT and a uni-directional clause transmission from SPASS to SETHEO. However, we forbid DISCOUNT and SETHEO to exchange clauses.

Cooperative runs were performed as follows. In each initialization phase all provers obtained the results of SPASS' normal form translator FLOTTER as initial clause set. Sorts were encoded with the simple term encoding mechanism from [Section 5.1.2]. Since SETHEO is not able to utilize built-ins for equality the usual equality axioms were added to its clause set. Since DISCOUNT is an equational prover we deleted non-unit clauses from its clause set. Positive equations were used as axioms, negative (in-)equations as proof goals. When using SETHEO in ILF it usually obtains clauses from a different normal form translator. It turned out, however, that for cooperation purposes the use of identical normal forms is unavoidable. If the provers work on different kinds of normal forms (including different signatures due to different Skolemization procedures) SETHEO is in general not able to close many tableau branches with the help of SPASS lemmas because unification failures arise immediately. It is to be emphasized that SETHEO's performance (when working alone) was not decreased when employing the FLOTTER normal form. We let the provers cooperate every 5 seconds.

$T_{\max}$ (secs.)	SPASS	SETHEO	DISCOUNT	competitive	TECHS
10	37	39	0	48	49
30	47	39	0	54	57
60	48	45	0	55	58
120	50	48	0	56	63
solvability	62%	59%	0%	69%	78%

Table 5: Experiments with cooperating theorem provers

The referees were parameterized in the following way. In a cooperation phase, SPASS selected 40 clauses for SETHEO, the receive referee of SETHEO selected 30 of these clauses. SPASS and DISCOUNT selected 10 clauses for each other via send-referees, 5 of these clauses were finally selected by their receive-referees. Note that these parameters are not the result of exhaustive studies.

Results can be found in [Tab. 5]. They are given in form of the number of solved problems after 10, 30, 60, and 120 seconds. Results of SPASS, SETHEO, and DISCOUNT are displayed in columns 2–4. Note that DISCOUNT is not able to solve a problem since all proof problems contain non-unit clauses which are needed in a proof. Column 5 shows the results of a competitive 3-prover team consisting of SPASS, SETHEO, and DISCOUNT. Finally, column 6 gives the results of an analogous cooperative 3-prover team.

The results show the high potential of cooperation. The number of solved problems can be increased. After 120 seconds our cooperative team can solve 10% more problems than the competitive team and even 17% more problems than the best single prover SPASS. In addition, run times can significantly be decreased. E.g., the cooperative team is able to solve more problems within the timeout of 30 seconds than the competitive team within the timeout of 120 seconds. Note that DISCOUNT, although it cannot solve a problem when working alone, is important in the cooperative team. When working with a team consisting of SPASS and SETHEO only, merely 61 problems can be solved. This shows that our concept is well-suited for integrating specialized provers. Since only a very small number of clauses is selected and exchanged, the overhead caused by the communication is rather small. The results reveal that the gains provided by exchanging clauses are much larger than the communication overhead.

6 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 engines. However, their combination results in a useful 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,
- 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 3.2.3]). Fortunately, the same set of axioms allows to dis-prove a large number of tasks. After inspection of some failed dis-proof attempts, it turned out that the necessary axioms are rather simple and consistent with the given 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 occurred 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.

7 Conclusions

In this paper, we describe the integration of several deduction techniques and ATPs to tackle 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 and it turned out that pure integration of several systems is too coarse for our purpose. In such a complicated technique as automated deduction the dependencies between the used approaches are still mostly unclear from the theoretical point of view. So we have to exploit statistical information about success rates to minimize unwanted effects caused by a bad combination of potentially good systems. 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 whole system.

On the actual deductive level, our main methods of attack are *cooperation* between different ATPs and *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. These results are further improved by proper cooperation of the ATPs. In contrast to a competitive system, also specialized provers can be useful in a cooperation environment. Currently, we thus achieve an overall recall of approximately 80% with moderate timeouts which indicates that deduction-based retrieval has become a feasible application of automated theorem proving.

References

[Barringer, Cheng, and Jones 84] H. Barringer, J. H. Cheng, and C. B. Jones. "A Logic Covering Undefinedness in Program Proofs". *Acta Informatica*, **21**(3):251–269, 1984.

[Bündgen, Göbel, and Küchlin 94] R. Bündgen, M. Göbel, and W. Küchlin. "Parallel ReDuX $\rightarrow$ PaReDuX". In *Proc. Int. Conf. on Rewriting Techniques and Applications (RTA-95)*, *LNCS* 914, pp. 408–413, Springer, 1995.

[Bonacina and Hsiang 95] M.P. Bonacina J. Hsiang. "The Clause-Diffusion methodology for distributed deduction". *Fundamenta Informaticae*, **24**:177-207, 1995.

[Bibel and Schmitt 98] W. Bibel and P. H. Schmitt, (eds.). *Automated Deduction - A Basis for Applications*, Vol. I - III. Kluwer Academic Publishers, 1998.

[Cheng and Jeng 92] B. H. C. Cheng and J. Jeng. "Using Automated Reasoning to Determine Software Reuse". *Int. J. Software Engineering and Knowledge Engineering*, **2**(4):523–546, 1992.

[Denzinger 95] J. Denzinger. "Knowledge-Based Distributed Search Using Teamwork". In *Proc. Int. Conf. on Multi Agent Systems (ICMAS) 95*, pp. 81-88. AAAI Press, 1995.

[Denzinger and Fuchs 98] J. Denzinger and D. Fuchs. "Enhancing conventional search systems with multi-agent techniques: a case study". In *Proc. Int. Conf. on Multi Agent Systems (ICMAS) 98*, Paris, France, pp. 419-420. IEEE Computer Society, 1998.

[Dahn et al. 97] 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.

[Fuchs and Denzinger 97] D. Fuchs and J. Denzinger. "Knowledge-based cooperation between theorem provers by TECHS". *Technical Report SR-97-11*, U. Kaiserslautern, 1997.

[Fuchs 98a] D. Fuchs. "Goal lift-up: a technique for combining connection-tableau-based proof procedures". *Technical Report SR-98-06*, U. Kaiserslautern, 1998.

[Fuchs 98b] D. Fuchs. "Cooperation between Top-Down and Bottom-Up Theorem Provers by Subgoal Clause Transfer". In *Proc. 4th Int. Conf. on Artificial Intelligence and Symbolic Computation (AISC)*, Plattsburgh, NY, USA, *LNAI* 1476, pp. 157-169. Springer, 1998.

[Fuchs 98c] D. Fuchs. "Coupling Saturation-Based Provers by Exchanging Positive/Negative Information". In *Proc. 9th Int. Conf. on Rewriting Techniques and Applications (RTA-98)*, Tsukuba, Japan, *LNCS* 1379, pp. 317-331. Springer, 1998.

[Fischer, Schumann, and Snelting 98] B. Fischer, J. M. P. Schumann, and G. Snelting. "Deduction-Based Software Component Retrieval". In [Bibel and Schmitt 98], Vol. III, pp. 265-292.

[Fuchs and Wolf 98] M. Fuchs and A. Wolf. "Cooperation in Model Elimination: CPTHEO". In *Proc. CADE-15*, *LNAI* 1421, pp. 42–46. Springer, 1998.

[Harrison 95] J. Harrison. "Inductive definitions: automation and application". In *Proc. 8th Int. Workshop on Higher Order Logic Theorem Proving and Its Applications*, *LNCS* 971, pp. 200–213. Springer, 1995.

[Hähnle, Kerber, and Weidenbach 96] R. Hähnle, M. Kerber, and C. Weidenbach. Common Syntax of DFG-Schwerpunktprogramm "Deduktion". *Technical Report 10/96*, U. Karlsruhe, 1996.

[Javasoft 98] JavaSoft. JavaBeans^{TM}, Version 1.01, July 1998.

[Jones and Middelburg 94] C. B. Jones and K. Middelburg. "A Typed Logic of Partial Functions Reconstructed Classically". *Acta Informatica*, **31**(5):399–430, 1994.

[Krueger 92] C. W. Krueger. "Software Reuse". *ACM Computing Surveys*, **24**(2):131–183, 1992.

[Letz and Schumann 90] R. Letz and J. M. P. Schumann. "PARTHEO: A High-Performance Parallel Theorem Prover". In *Proc. of CADE-10*, Springer, 1990.

[Letz, Mayr, and Goller 94] R. Letz, K. Mayr, and C. Goller. "Controlled Integration of the Cut Rule into Connection Tableau Calculi". *J. Automated Reasoning*, **13**:297–337, 1994.

[Middelburg 93] K. Middelburg. Logic and Specification — Extending VDM-SL for advanced formal specification. *Computer Science: Research and Practice*. Chapman & Hall, 1993.

[Mili, Mili, and Mittermeir 98] A. Mili, R. Mili, and R. Mittermeir. "A Survey of Software Reuse Libraries". *Annals of Software Engineering*, **5**:349–414, 1998.

[Moorman Zaremski and Wing 97] A. Moorman Zaremski and J. M. Wing. "Specification Matching of Software Components". *ACM Trans. Software Engineering and Methodology*, **6**(4):333–369, 1997.

[Oberschelp 62] A. Oberschelp. "Untersuchungen zur mehrsortigen Quantorenlogik". *Mathematische Annalen*, **145**:297–333, 1962.

[Paulson 94] L. C. Paulson. *Isabelle: A Generic Theorem Prover*, *LNCS* 828. Springer, 1994.

[Reif and Schellhorn 98] W. Reif and G. Schellhorn. "Theorem Proving in Large Theories". In [Bibel and Schmitt 98], Vol. III, pp. 225-242.

[Schumann and Fischer 97] J. M. P. Schumann and B. Fischer. "NORA/HAMMR: Making Deduction-Based Software Component Retrieval Practical". In M. Lowry and Y. Ledru, (eds.), *Proc. 12th Int. Conf. Automated Software Engineering*, pp. 246–254, Lake Tahoe, IEEE Comp. Soc. Press, 1997.

[Sutcliffe 92] G. Sutcliffe. "A Heterogeneous Parallel Deduction System". *Proceedings of FGCS'92 Workshop W3*, Tokyo, Japan, 1992.

[Sutcliffe, Suttner, and Yemenis 94] G. Sutcliffe, C. B. Suttner, and T. Yemenis. "The TPTP Problem Library". In *Proc. CADE-12*, *LNCS* 814, pp. 252–266. Springer, 1994.

Acknowledgements

This work is supported by the DFG within the "Schwerpunktprogramm Deduktion", grants Wo625/3-1, De535/4-1, and Sn11/2-3. Bernd Fischer was with the TU Braunschweig, Abt. Softwaretechnologie, when writing this paper.