Deduction-Based Software Component Retrieval*

Bernd Fischer
Abt. Softwaretechnologie, TU Braunschweig
fisch@ips.cs.tu-bs.de

Johann Schumann
Automated Reasoning, TU München
schumann@informatik.tu-muenchen.de

Gregor Snelting
Abt. Softwaretechnologie, TU Braunschweig
snelting@ips.cs.tu-bs.de

March 1998

Abstract

We present an application of automated theorem proving to software engineering: the reuse of software components based on their formal specifications. Such specifications can be viewed as *contracts*. Our work is motivated by the fact that safe reuse can only be achieved in a contract-based development process, where contracts are used as the actual medium for component retrieval. Theorem provers can be applied to match contracts against user queries, thus retrieval becomes a deduction problem. This application presents some unique challenges because a huge amount of proof tasks is generated, most of them being non-theorems. In order to achieve acceptable response time, our retrieval system NORA/HAMMR uses an incremental filter pipeline. Components are first selected or rejected by signature matching and model checking; the prover is only applied to components which survived earlier stages in the pipeline. This chapter describes design and underlying technology of NORA/HAMMR. It concludes with several empirical evaluations of retrieval performance in connection with the provers PROTEIN, SETHEO and SPASS.

1 Reuse by Contract

Reuse of approved components and concepts is a major characteristic of mature engineering disciplines. In software engineering it has even been considered from the very beginning (cf. McIlroy's [McI68] contribution to the seminal NATO conference) but despite ongoing interest and research it is by no means an established practice.

The most basic technique, reusing code pieces, still prevails. Unfortunately, it is complicated because of the amount of information which must be considered as even some very fine details of the code must be inspected and checked against the new assumptions. And while experience tells us that this is possible, it has also taught us that it is difficult. Code reuse remains inherently risky and the spectacular stories are no success stories. The investigation of the recent Ariane 5 disaster for example revealed that it was caused by the

*to appear in: *Automated Deduction — A basis for applications*, Vol. III, Chapter 11, W. Bibel and P. H. Schmidt (eds.), Kluwer 1998. Please refer always to the final version. This preprint is not for public dissemination.

reuse of an unmodified Ariane 4 software component which led to an uncaught exception crashing the software and hence the spacecraft [Lio96].

It has been argued that the ultimate reason for the crash was the component's failure to make its assumptions explicit. Jézéquel and Meyer [JM97] thus propose to use *contracts*, i.e., formal models of the components' behavior. The best-known contracts are axiomatic specifications with pre- and postconditions but other formats are also possible, e.g., type and class invariants or abstract statements. Jézéquel and Meyer conclude that

> "*reuse without a contract is a sheer folly.* From CORBA to C++ to VisualBasic to ActiveX to Java, the hype is on software components. The Ariane 5 blunder shows clearly that naïve hopes are doomed to produce results *far worse* than a traditional, reuse-less software process. To attempt to reuse software without assertions is to invite failures of potentially disastrous consequences."

Reuse by contract embodies this premise: the components are associated with contracts and organized, retrieved, and reused by these. This approach has three major benefits:

- it supports safe reuse and safe software composition because it can retrieve the components which provably fulfill the contracts,
- it promotes abstraction because the contracts become the actual medium by which components are retrieved, and
- it constitutes a formal framework in which every single reuse step can be formally justified.

In essence, reuse by contract is the application of formal methods to software reuse. Within our project NORA/HAMMR[1] we investigate not only technical problems as for example scalable and efficient architectures or contract-based library indexing and retrieval techniques but also organizational aspects such as reuse-friendly specification techniques, the integration into formal software development processes, or the interaction with conventional reuse mechanisms. Our long-term goal is to build a system which smoothly combines component design, implementation, and verification with the systematic reuse of fully specified and verified component libraries.

In this chapter we will focus on *deduction-based software component retrieval* because it is the key problem in reuse by contract: "You must find it before you can reuse it!"[2] Its purpose is to check whether the contracts of the client and a candidate component are compatible in some sense and hence the candidate is suitable for reuse. It thus builds a proof task from the contracts which formally captures the intended compatibility and uses automated deduction techniques to solve this task. If the proof obligation can be solved the candidate is retrieved, if not it is discarded—retrieval becomes a deduction problem. The crucial point is to retrieve in a short time as many relevant components as possible (high recall) while maintaining a high fraction of "right" components (high precision).

The basic idea of deduction-based retrieval is very simple and has already been proposed several times before [KRT87, RW91, MM91] but a practical implementation is very

[1]NORA is no real acronym; HAMMR is a highly-adaptive multi-method retrieval tool. This work has been supported by the DFG under grants Sn11/2-1, Sn11/2-2, Sn11/2-3 and Schu908-5/1 and within the SFB 342, subproject A5.

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

difficult. It is not sufficient to concentrate on the deduction problems because only a carefully engineered complete retrieval system generates tractable problems. But one must also take care of other aspects: integration into the software development process, contract language, user interface, response times, or degree of automatization come immediately to mind. It is the consideration of such non-deductive aspects which sets NORA/HAMMR apart from other approaches.

The remainder of this chapter is organized as follows. The next section discusses various forms of proof tasks and their effects on software reuse. Section 3 then reviews the requirements for a practical tool and develops the architecture of NORA/HAMMR. Its main feature is a pipeline of filters of increasing deductive strength which maintains a balance between fast response and high precision. Sections 4 to 6 are each devoted to a class of such filters: signature matching filters, rejection filters and confirmation filters with automated theorem provers (ATPs). We describe the respective steps which must be taken to translate the proof tasks from the application domain and to prepare them for the different provers. Section 7 details on the experimental evaluation of this process. We conclude the chapter with a survey of related work and with discussions of the lessons learnt in this application of automated theorem proving to software engineering and future work on NORA/HAMMR.

2 Contracts, Retrieval, and Reuse

2.1 Contract Languages

Reuse by contract requires that each component is associated with a contract of its own which captures the essentials of the components behavior. In practice, components are usually functions and axiomatic specifications with pre- and postconditions are the most common form of contracts. Pre- and postconditions of a component c (which we will denote by pre_c and $post_c$, respectively) could in principle be formulated in pure FOL but this is hardly acceptable by the user. Instead, a more convenient specification language with a rich set of predefined operations must be chosen as contract language or *custom logics*. In NORA/HAMMR, we apply VDM-SL [Daw91]. VDM-SL allows a concise formulation and even the specification of side effects. Figure 1 shows a typical component which has been taken from our experimental library (cf. Section 7.1).[3]

Using custom logics, however, usually requires either a specialized deductive component or a translation into FOL to retain soundness. VDM-SL is based on the three-valued *logics of partial functions* or LPF [BCJ84] for which some dedicated provers exist (e.g., *mural* [JJ+91] and Isabelle/VDM-LPF [AF97]). But these are not applicable in our case because they work interactively. We decided not to tailor a fully automatic proof procedure for LPF (e.g., based on a many-valued prover as 3TAP [BG+91]). Rather, we chose the second approach and translate the proof tasks from VDM-SL via LPF to FOL. Hence, we can re-use existing high-performance theorem provers and model checkers for FOL. We apply an optimized variant of the translation of Jones and Middelburg [JM94] which will be described in Section 3.4.

[3] In VDM-SL, ^ means concatenation of lists, [ ] the empty list, $[i]$ a singleton list with item i, and hd and tl the functions head and tail, respectively.

```
module rotate
  exports functions rotate : list $\xrightarrow{p}$ list
  types item = token; list = seq of item
  functions

    empty : list ⟶ bool
    empty(l) == l = [ ]

    rotate(l : list) l' : list
    pre    ¬ empty(l)
    post   l' = (tl l) ^ [hd l]
end rotate
```

Figure 1: Example component

2.2 Deduction-Based Retrieval and Proof Tasks

The purpose of deduction-based component retrieval is to identify components with compatible contracts. However, there are different notions of compatibility modeling different approaches to reuse by contract. Match relations formalize these notions via their associated *proof tasks* formulated in the contract language.

Generally, a component c can be considered as compatible if it bridges the gap between the pre- and postconditions the client requires. Ideally, c fits completely and can thus be plugged into the place of the query q because it has a weaker pre- and a stronger postcondition than required by q. This relation is sometimes [ZW95a, PBA95] formalized by proof tasks of the form

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

but this is not entirely adequate.[4] Reusable components are usually implemented very defensive yet general. Hence, they may work on wider domains than expected but their results on the domain extensions need not to fit the user's expectations. If we want to retrieve such components anyway we must restrict the postconditions to the domain specified by pre_q:

$$(pre_q \Rightarrow pre_c) \wedge (pre_q \wedge post_c \Rightarrow post_q) \tag{2}$$

This variant of *plug-in compatibility* increases recall compared to the standard variant but due to the domain restrictions it still supports safe reuse. The retrieved components can be considered as black boxes and may be reused "as is", without further proviso or modification.

Under a weaker notion of compatibility, c is not required to close the gap completely but must bridge only a part of it. Then, c must have a stronger postcondition than desired, at least on its own domain, but this domain is not determined by the query:

$$pre_c \wedge post_c \Rightarrow post_q \tag{3}$$

In contrast to plug-in compatibility, code reuse based on this *conditional compatibility* is potentially unsafe because the client still has to satisfy the open obligation pre_c.

[4]Actually, the proof tasks are universally closed with respect to the formal input and output parameters of the component and the query. However, to improve readability, we use these traditionally abbreviated formulations.

Deduction-based retrieval is also applicable for traditional white-box code reuse. The *partial compatibility* match relation

$$pre_c \wedge pre_q \wedge post_c \Rightarrow post_q \tag{4}$$

retrieves all components which do "the right thing" at least on the domain restricted by $pre_c \wedge pre_q$. By varying pre_q, clients can control recall and granularity of reuse. The weaker it is, the more components are retrieved but the smaller is their respective benefit. However, in the absence of a formal process, manual checks or code modifications are required in order to match the components precondition.

2.3 Deduction-Based Retrieval and Formal Software Development

Deduction-based retrieval works best in connection with formal software development processes. Here, the contracts arise naturally and do not impose extra work on the developers. Since retrieving components is an alternative to any single refinement step, reuse should be built into the process from the very beginning. This does not only increase productivity as the development steps become larger but also enables vertical prototyping and thus increases the confidence into the validity of the system being built.

3 The System NORA/HAMMR

3.1 Requirements

The goal of the project NORA/HAMMR is to build a practical deduction-based retrieval tool for the working software engineer. Since "practical" has no precise definition we will discuss some of its facets in the following. For practical purposes, *retrieval* can be measured in terms of response times, *recall* ("do we get all matching components?") and *precision* ("do we get the right components?").[5] In the case of reuse by contract, however, their relative importance depends on the applied software development process. The speed of the tool is of course also important due to the *Fourth Reuse Truism*: "You must find it faster than you can rebuild it!" [Kru92]. It is, however, dominated by the time required to discard non-matches.

In a formal process, the first proven match already saves significant time which is otherwise spent in refinement steps and their justifications. Hence, users are not very much interested in the actual recall as long as it is not zero.

In a non-formal process the picture is quite different. Here, most time is spent in analyzing and modifying the retrieved components manually, in particular when partial compatibility is used. Hence, sets of promising candidates should be available for inspection immediately and at any time. Furthermore, a high recall is more important than high precision.

For a *practical tool*, "look and feel" is extremely important. Here, the most obvious aspect is the presentation. The bare theorem provers are unsuitable for non-experts and must thus be hidden by a dedicated tool interface. It must also offer facilities to control the retrieval process and to monitor its progress.

[5]Cf. Section 7.2 for exact definitions of recall and precision.

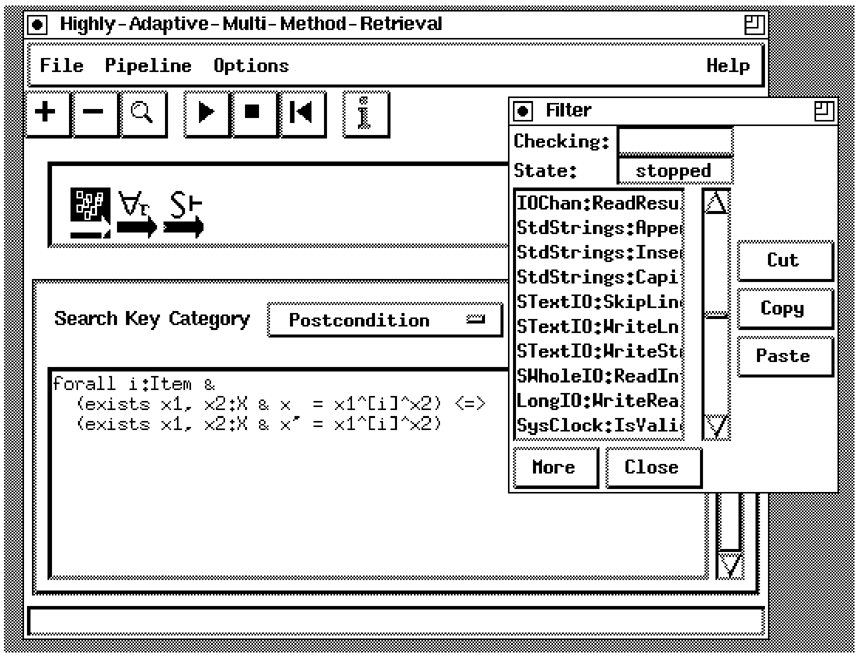
The number and structure of proof tasks pose severe problems. In contrast to other applications, e.g., in mathematics, where only a few tasks over small signatures and with few axioms have to be solved, we face an "inverse economics of scale." Each single query already induces a number of complex proof tasks which is in the order of the size of the component library. Consequently, their construction is itself a major task which must run fully automatically. It also involves steps which are usually done by "experienced experimentators", e.g., addition of key lemmas or setting of prover control parameters. Moreover, since reasonable libraries contain only few matches for any query, most of the tasks describe non-theorems. This renders the naïve generate-and-proof approach impossible. Instead, it becomes vital to rule out non-theorems using dedicated disproving tools to prevent the actual prover from drowning. For the remaining tasks, aggressive simplification is required before the proof can actually be attempted. But even then it may remain necessary to use different provers, axiomatizations, or tactics in competition to solve enough tasks.

3.2 System Architecture

NORA/HAMMR's system architecture reflects these diverse and sometimes even conflicting requirements. NORA/HAMMR is built as a pipeline of independent filters through which the candidates are fed. This design supports an incremental retrieval approach. Each component which passes a filter can immediately be inspected by the user. This early feedback is essential for an effective control of the retrieval process as queries which are too weak (i.e., in-discriminative) or too strong (i.e., over-specified) become obvious before the entire library is checked by the first filter.

Typically, a filter pipeline starts with a *signature matching* filter which checks for compatible calling conventions. Then, *rejection* filters try to efficiently discard as many non-matches as possible. Finally, *confirmation* filters are invoked to guarantee a high precision of the retrieval results. The next sections of this chapter are devoted to these filters. The pipeline itself can be freely customized although any configuration should maintain a fair balance between initial fast responses and a final high precision. However, it still allows the integration of arbitrary non-deductive methods, e.g., full text retrieval (see e.g., [MS89, MBK91, FN87]). This is especially important in a non-formal process where the established methods need to be augmented and not to be replaced in order to gain acceptance.

The prototypical graphical user interface (cf. Figure 2) reflects the system architecture. The filter pipeline may be pre-selected via a pull-down menu and may easily be customized through the central icon pad. Each icon also hides a specialized control window which allows some fine-tuning of the filter. Additional inspectors display the intermediate results and grant easy access to the components. They also allow to save retrieved components into new library files which may be used for subsequent retrieval runs. A simple, VCR-like control panel provides single-button control of the entire system. The objective of the GUI is to hide evidence of automated deduction techniques as far as possible but to grant control over these techniques as far as necessary. Hence, using NORA/HAMMR as a *retrieval tool* requires only knowledge of the contract and target languages.



```
forall i:Item &
  (exists x1, x2:X & x  = x1^[i]^x2) <=>
  (exists x1, x2:X & x' = x1^[i]^x2)
```

Figure 2: Graphical user interface

3.3 Reuse Administration

Any reuse scheme requires an initial library ("seed") and some continuous maintenance but this demands more detailed knowledge of the contract language and the applied deduction methods than the average re-user has. We thus assume that a dedicated *reuse administrator* is responsible for these tasks.

Construction of an initial library is straightforward but tedious. The components (e.g., functions, procedures, or methods) need to be extracted and indexed: the contracts must be written or checked and completed by all necessary context information (e.g., type definitions or auxiliary functions). Since we use VDM-SL as contract language, we represent a component as a VDM-SL module which contains the implicitly specified function and all referred constructs.

The components may be organized hierarchically using plug-in compatibility to reduce the number of proof tasks [MMM94, JC94] but for simplicity we currently work with a flat library. We also assume without a formal justification that the contract is a faithful representation of the component and that it contains all information of interest to the clients. Hence, the actual source code is not required for retrieval purposes.

The reuse administrator must also support the deductive infrastructure. This in particular includes the construction and maintenance of a *lemma library* which contains the axiomatization of the contract language. In NORA-HAMMR, the library is structured hierarchically and also contains meta-information as for example operator rules, data type generators, and minimal axiomatizations. This supports simplification and axiom selection using similar techniques as described in Chapter III.2.11.

3.4 Translation of Proof Tasks

Regardless of the chosen compatibility criterion, NORA/HAMMR first generates the proof tasks in form of VDM-SL-expressions which are then translated in two phases into FOL. The first phase transforms the expressions into formulas of the underlying LPF-calculus. Following [Mid93], it eliminates all extra-logical custom constructs. E.g., custom-bindings as *let*-bindings or pattern matching are replaced by appropriately scoped quantifiers, and local definitions of auxiliary predicates and functions are transformed into equivalence and equality clauses, respectively. But it also adds axioms to represent properties of locally defined data types, e.g., constructor properties for tagged unions. Finally, it simplifies the imposed sort structure and replaces all type invariants by explicit predicates using standard relativation techniques [Obe62].

The second phase takes the resulting LPF-formula φ and translates it into FOL in such a way that provability is preserved, i.e., essentially that $\vdash_{\text{LPF}} \varphi \iff \vdash_{\text{FOL}} \varphi'$ holds. Following [JM94], we use signed translation 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. Of course, the translation into pure FOL is provability-preserving for the non-inductive part of LPF only. For the proper handling of recursive data types an inductive ATP would be required but in practice our induction approximation described in Section 6.2 works sufficiently well.

The necessity or even usefulness of a three-valued logic for software specification purposes has often been questioned and we refer to Jones [CJ91, Jon95] for its defense. However, in our experience, the application of LPF proved ultimatively to be a matter of taste and not an insurmountable problem. The optimized translation avoids much of the usual overhead and after application of simplifications described in Section 5.1, the translated formulas are very similar in size and structure to respective formulas originally written in classical two-valued logics.

4 Signature Matching Filters

The main purpose of signature matching is to identify components which have compatible calling conventions in the target language. It thus has to abstract from possible implementation choices, e.g., order of parameters or currying. It can even work across language boundaries to increase interoperability. Signature matching filters do not use full contracts for retrieval but only type information. They can thus be implemented using simpler deduction techniques than full first-order theorem proving which makes them also suitable as fast pre-filters.

Signature matching is a disguised type inference problem: basically, a component can be retrieved if the desired type can be derived for it, i.e., for a library $\mathcal{L}$ the set $\{c \in \mathcal{L} \mid \vdash_{\mathcal{T}} c : \tau_q\}$ formally defines the answer for a query τ_q using the type system $\mathcal{T}$. Generally, this type-theoretic interpretation of signature matching gives us sound semantic foundations which are also applicable if the semantics of the target language itself offers no suitable interpretation of types.

In order to abstract from the implementation choices, the type system $\mathcal{T}$ used for retrieval must of course contain additional rules beyond those used for typing the language. E.g., to take care of currying, the inference rule

$$\frac{\Gamma \vdash c : (\alpha \times \beta) \to \gamma}{\Gamma \vdash c : \alpha \to (\beta \to \gamma)}$$

which allows c to have the curried type as well is required.

The type derived for a component, however, does not need to be its *principle type* [DM82]. Any query for components of type $\mathit{list}(\mathit{int}) \to \mathit{int}$ should also retrieve the polymorphic *hd*-function, whose principle type is usually $\forall \alpha \cdot \mathit{list}(\alpha) \to \alpha$. Hence, retrieving components with more general types is the equivalent to plug-in compatibility. Similarly, to support partial compatibility, the query does not need to be completely specific. It may contain free (i.e., not polymorphically bound) type variables which may be instantiated by the derivation process.

For efficient implementations, an actual type inference (which is in fact a syntax-directed proof search) is still much too expensive. Fortunately, this it not necessary because it suffices in most cases to work on the type terms only. The key idea is to replace type inference by E-unification . Here, the equational theory E captures the intended abstractions of $\mathcal{T}$, matching from the library type to the query cares for more general types and matching from the query to the library handles the free variables. To maintain the semantic integrity, it is crucial to show that E is an adequate representation of $\mathcal{T}$, i.e., that $\tau_c \doteq_E \tau_q$ for the most general type τ_c of any retrieved component c.

Signature matching has two important side effects. It identifies the respective formal parameters of the two components and it maps the respective domains (i.e., types) onto each other. Both effects are required to "join" the different scopes of two contracts together. This makes signature matching also a necessary prerequisite for the subsequent filters.

5 Rejection Filters

Detecting and rejecting non-theorems is an essential step in deduction-based retrieval because most of the tasks result from non-matching candidates. Unfortunately, most ATPs for full first-order logic are not suited for this. They exhaustively search for a proof (or refutation) of a conjecture (or its negation) but they usually fail to conclude that it is *not* valid (or contradictory). The basic idea of rejection filters is to use efficient and terminating proof procedures for logics which are decidable but weaker than FOL, e.g., quantifier-free or monadic fragments. In order to apply these procedures, the filter must either identify suitable subtasks or translate (i.e., abstract) the entire task into the weaker fragment.

Such filters are necessarily incomplete in a sense that non-theorems may still pass through. However, from a practical point of view, this is not too severe as it only reduces the effectiveness of the filters. Unfortunately, some rejection filters may also be unsound and valid tasks may be rejected improperly, e.g., because the applied abstraction is not not sufficiently precise. Candidates belonging to such tasks cannot be recovered again which reduces the recall of the entire pipeline. The engineering problem is to balance this loss against the achieved reduction.

Rejection filters can be based on different principles. In the sequel we will focus on two approaches: using logic simplification to detect non-matches and applying model generators to find counter-examples.

5.1 Simplification

Aggressive simplification of automatically generated proof tasks is always necessary because they contain much redundancy, e.g., too large quantifier scopes or redundant equations. But simplification can also be used as a rejection filter. If a task can be simplified to *false*, the candidate may obviously be rejected.

In NORA/HAMMR, we have experimented with different levels of simplification procedures. The lowest level works with the FOL connectives and quantifiers. Here we eliminate the propositional constants *true* and *false* which typically stem from the preconditions of the components, and transform the proof tasks into a (conjunctive or disjunctive) normal form and then further into anti-prenex form to minimize the quantifier scopes. The next level handles equalities. For simplicity, we do not rewrite with all equations but restrict ourselves to equations where at least one side is a single variable. Due to the structure of the proof tasks it is necessary to use inequalities for conditional rewriting, too. Both levels involve only the usual symbols of FOL and are thus hard-wired into our system. Because the preceding translation of the proof tasks already handled the LPF-interpretation of the symbols, we can apply the usual rewrite rules for simplification.

However, to achieve a significant filtering effect more semantic information must be utilized. In NORA/HAMMR, we extract this information automatically from the lemma library maintained by the reuse administrator. The lowest semantic level just identifies all axioms and lemmas suitable as rewrite rules, extracts a terminating subset and rewrites the task into the respective normal form. For operators which are identified as associative and commutative, we also use AC-rewriting. In contrast to rewrite-based *theorem proving*, rejection filters need no confluent rewrite system and hence no expensive Knuth-Bendix completion. On this level, we also consider quantifiers as "ordinary" operators and rewrite them modulo α-conversion. We can thus use the axiom $\exists x : T \cdot p(x)$ to simplify $Q \wedge \exists y : T \cdot p(y)$ to Q.

The next semantic level is concerned with properties of sorts. If a sort is marked as freely generated by some set of constructor functions, we can use this information to simplify equations between constructor terms. E.g., for lists we simplify $nil = cons(x, xs)$ to *false* and $cons(x, xs) = cons(y, nil)$ to $x = y \wedge xs = nil$. But we use the generator information even more aggressively. From the type information of the generators we can determine whether the domain of a sort contains at least two different elements or not. If this is the case, we rewrite universally quantified equations as for example $\forall x : T \cdot x = t$ to *false* provided that x does not occur free in t. Similarly, existentially quantified equations are rewritten to *true*. Both simplifications are in general unsound when partial functions are involved but the preceding LPF-translation guarantees that they do not change the provability of the original formulas.

The final semantic level again uses the generator information to unfold sorted quantifiers by a single step. This is similar to the induction approximation described in Section 6.2. The difference is that we do not generate a proper step case. For example, we rewrite $\forall l : List \cdot \mathcal{F}(l)$ only to $\forall i : Item, l' : List \cdot \mathcal{F}(nil) \wedge \mathcal{F}(cons(i, l'))$.

5.2 Counterexample Generation

A component c can always be rejected, if we find a "counter-example" for its associated proof task because it then cannot be valid. Model generators for FOL like Finder [Sla94] or MACE [McC94] try to find such counter-examples (which are just interpretations under which the task evaluates to *false*) by systematically checking all possible interpretations. Their highly efficient implementation (usually using BDD-based Davis-Putnam decision procedures) would make them ideal candidates for fast rejection filters.

However, the exhaustive search terminates only if all involved domains are finite. But most domains in our application are 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*.

Domain abstraction uses techniques from abstract interpretation [CC77] where the infinite domain is partitioned into a small finite number of sets which are called abstract domains. Abstract model checking [Jac94] then represents the abstract domains by single model elements and tries to find an abstract counter model, using an axiomatization of the abstract functions and predicates with a standard FOL model generator. The drawback of abstract model checking is that some domains in combination with certain predicates (e.g., equality) cannot be abstracted properly.

The experiments we describe in Section 7.4 are based on a second approach which uses domain approximations. From the infinite domain, we select a number of values which seem to be "crucial" for the component's behavior. E.g., for lists, we pick the empty list *nil* and some small lists with one or two elements. Then, we search for models or a counter-example. This approach mimics the manual checking for matches: if one looks for matching component, one first makes 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. For some invalid formulas, a counter-example can be found in the finitely approximated domain, e.g., $\exists l : \mathit{List} \cdot \exists i : \mathit{Item} \cdot l = [i]^\wedge l$. Vice versa, for some valid formulas the approximation may be no longer a model, e.g., $\exists x, y, z : \mathit{List} \cdot x \neq y \wedge y \neq z \wedge z \neq x$ which has a model only in domains with at least three distinct elements. While the first case is not too harmful for our application—the performance of the filter just decreases (i.e., more proof tasks can pass), the second one is dangerous: proof tasks describing valid matches might be lost.

6 Confirmation Filters

Confirmation filters are in charge of rejecting any task for which they cannot find a proof. If a proof is found, the retrieved component actually matches the query with respect to the selected compatibility criterion (see Section 2.2). Thus, the last filter in any pipeline should be such a confirmation filter.

In contrast to rejection filters, the applied deduction must be sound. Otherwise, the guaranty becomes void and the precision may become imperfect. In NORA/HAMMR, we use first-order automated theorem provers as confirmation filters. The experiments described in this chapter have been carried out with PROTEIN, (p-)SETHEO, and SPASS. These systems are sound, i.e., they always guarantee full precision of the retrieval process. However, the challenge in this application is to retain a high recall with short answer times in the range of up to a few seconds.

The proof tasks must be pre-processed by a variety of steps to generate suitable input for the different provers. While some of these steps are specially tailored for each prover (e.g., handling of sorts and equations, control of the prover), others address common problems: logic simplification of the formulas (using techniques described in Section 5.1), selection of axioms, and handling of inductive problems.

6.1 Selection of Axioms

Each proof task must contain—besides the theorem and the hypotheses—the properties of each data type (e.g., *List, Nat*) as a set of axioms. Further lemmata can be added to the formula to facilitate and abbreviate complex proofs. Automated theorem provers, however, are extremely sensitive with respect to 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 orders of magnitude, 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 strong time-constraints furthermore do not allow us to use time-consuming selection techniques. In our prototype, we therefore use a simple strategy similar to the one described in Chapter III.2.11 in this volume. The lemma library contains hierarchically structured sub-theories for all involved data types. By selecting all sub-theories which share function symbols with the theorem and the hypotheses and the sub-theories they rely on, we essentially obtain all axioms sufficient to find a proof of the given obligation.

6.2 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. The provers under consideration cannot handle induction by themselves and our strong time-constraints do not 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 a candidate with the signature $l : \mathit{List}$, and the corresponding proof task of the form $\forall l : \mathit{List} \cdot \mathcal{F}(l)$ we obtain the following cases:[6]

$$\begin{array}{l}
\forall l : \mathit{List} \cdot l = [] \Rightarrow \mathcal{F}(l) \\
\forall l : \mathit{List} \cdot \forall i : \mathit{Item} \cdot l = [i] \Rightarrow \mathcal{F}(l) \\
\forall l : \mathit{List} \cdot \forall i : \mathit{Item}, l_0 : \mathit{List} \cdot \mathcal{F}(l_0) \wedge l = [i]^\wedge l_0 \Rightarrow \mathcal{F}(l)
\end{array}$$

After rewriting the formula accordingly, we obtain three independent first order proof tasks which then must be processed by the prover. Only if all three proof tasks can be proven the entire proof task is considered to be solved. However, we cannot solve every inductive problem. Nevertheless, in combination with the simplification filter (Section 5.1, unfolding), this approximation is highly effective. For first experiments with this approximation see [SF97].

[6]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.

6.3 The Provers

All provers which are being used for the experiments (Protein, SETHEO, Spass) have been developed or enhanced within the Schwerpunkt Deduktion. They all accept formulas in first order logic and try to find a refutation for it. They receive proof obligations in sorted first order logic. Each prover has its own modules to convert the formulas into clausal normal form. The sorts of the proof tasks are imposed from the VDM-SL contracts and are structured hierarchically. VDM-SL allows only very limited overloading and polymorphism such that it can actually be resolved by the signature matching filter (cf. Section 4). It also allows, however, dynamic sorts with arbitrary invariants but these have been eliminated during the translation into FOL.

Details of the proof procedures of the provers are largely irrelevant for NORA/HAMMR. We can consider the provers as black boxes which return "proof found" or "failed to find a proof" after the given time-limit. Therefore, we will not give an extended description of each prover.

PROTEIN (*PRO*ver with a *T*heory *E*xtension *IN*terface, [BF94]) is a PTTP-based first order theorem prover over built-in theories for theory reasoning (e.g., handling of equations) using a linearizing completion technique. PROTEIN is based on the Model Elimination Calculus [Lov78] and refinements thereof.

SETHEO [LS$^+$92, GL$^+$94] is also based on the Model Elimination Calculus. However, the proof procedure is implemented as an extension of the Warren Abstract Machine. Equations are handled by the naïve approach of adding the corresponding equality axioms, or by the compilation approach used within E-SETHEO [MI$^+$97]. SETHEO provides a variety of parameters to control its search. In order to optimize SETHEO's answer times, *parallel competition* over parameters is used. The parallel prover p-SETHEO (see Chapter II.2.7 for details) allows us to obtain optimal efficiency combined with short answer times.

SPASS [WGR96, GMW97] is based on the superposition calculus developed by Bachmair and Ganzinger [BG94]. This calculus which is able to handle equality is extended by specific sort constraints (cf. Chapter I.2.9). This means that sort-information about the variables is represented as monadic predicates which are handled specifically within SPASS.

7 Experiments

7.1 The Experimental Data Base

All experiments were carried out over a database of 119 list specifications which were modified to have the type *list* → *list* because the integration of the signature matching filter is not yet completed. Approximately 75 of these specifications describe actual list processing 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 don't refer to the arguments (e.g., the result is not empty). For simplicity, we formulated the specifications such that the postconditions only use VDM-SL's built-in sequences and comparisons.

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 14161 proof tasks where 1860 or 13.1% were valid. Although the domain of our experimental library is rather limited, it serves as a good test-bed, because it allows to generate a realistically large number of proof tasks with many non-trivial matches.

7.2 Evaluation of Filters

Information retrieval methods are evaluated by the two criteria precision and recall [SM83]. Both are calculated from the set *REL* of *relevant* components which satisfy the given match relation with respect to the query and *RET*, the set of *retrieved* components which actually pass the filter. The *precision* p is defined as the relative number of hits in the response while the *recall* r measures the system's relative ability to retrieve relevant components:

$$p = \frac{|REL \cap RET|}{|RET|} \quad r = \frac{|REL \cap RET|}{|REL|}$$

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*

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

(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 rejected components. Finally, we define the *relative defect ratio* by

$$d = \frac{|\ REL \backslash RET\ |}{|\ \mathcal{L} \backslash REL\ |} \cdot \frac{|\ \mathcal{L}\ |}{|\ REL\ |}$$

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.

7.3 Rejecting Tasks with Simplification

Table 1 summarizes the results of the simplification filter. The columns labeled with FOL and Equality represent the respective syntactic levels, Sorts the full set of semantic simplifications except quantifier unfolding which is included in the last column. Runtimes (measured on a Sun SparcStation20) are well below 2 seconds. The results justify the application of simplification as first rejection filter. All four levels are sound and thus yield a total recall and a zero defect ratio. It is, however, somewhat disappointing that even aggressive syntactic simplification has absolutely no filtering effect and hence dumps the full fallout on the next filter. But both syntactic variants are already able to reduce a significant fraction (FOL: 21.0%, Equality: 26.7%) of the valid tasks to *true*. These can be added to the original rejections which gives the small total reductive effect shown in the reduction$^+$ entry.

Adding semantic information improves the situation slightly but only with unfolding we are able to reject more than 40% of the tasks because they are rewritten to *false*. The set of surviving candidates still contains more than 50% of the original non-matches (fallout) but its precision already increased by a factor of 1.7.

Level	FOL	Equality	Sorts	Unfolding
recall r	100.0%	100.0%	100.0%	100.0%
precision p	13.1%	13.1%	13.8%	22.1%
σ_p	0.19	0.19	0.2	0.25
increase	1.0	1.0	1.1	1.7
fallout	100.0%	100.0%	94.4%	53.2%
reduction	0.0%	0.0%	4.8%	40.7%
reduction$^+$	2.8%	3.5%	9.3%	45.2%
defect ratio d	0.0	0.0	0.0	0.0

Table 1: Results of simplification

Model Size: $\lvert List\rvert + \lvert Item\rvert$	2+1	3+1	3+2
recall r	74.7%	76.5%	81.3%
σ_r	0.25	0.26	0.25
precision p	18.5%	19.6%	16.5%
σ_p	0.21	0.19	0.16
increase	1.4	1.5	1.3
fallout	42.8%	41.0%	55.5%
reduction	50.1%	51.7%	39.0%
defect ratio d	0.51	0.45	0.48

Table 2: Results of model checking

7.4 Rejecting Tasks with Model Generation

For the experiments with counter-example generation, we used different approximations of the domains *List* and *Item*. Due to the large number of variables in the proof tasks, however, we are confined to small approximations with at most three elements. Table 2 shows the results obtained with the model generator MACE (with a run-time limit of 20 seconds). Regardless of the domain approximation, this filter is able to recover at least 75% of the relevant matches. This figure is at the lower end of our expectations, since almost 25% of the valuable matches are lost in this filter. Furthermore, the large standard deviation indicates that the filter's behavior is far from uniform and that it will perform poor for some queries.

On the other hand, between 41% and 55% of the non-matches (fallout) can pass this filter. Together with an increase of precision of approximately 1.5, the filtering effectiveness of the counter-example generation filter is in the range of the simplification filters. Due to the comparatively low recall, however, the defect ratios are quite high. Although this filter acts at least twice as good as blind guessing, the results are not satisfactory. The main reasons for this poor behavior are the large number of variables in the proof tasks which cause many timeouts and the deeply nested terms which would require larger domain approximations.

7.5 Using ATPs as Confirmation Filters

For the evaluation of the performance of the filters we only used the 1860 valid proof tasks of our library. Since the provers are not able to detect most non-matches, this selection is justified. Table 3 shows an overview of the results of all experiments. Proof tasks are most simple if one tries to retrieve identical components. In these cases, they are of the structure $\mathcal{A} \wedge \mathcal{F} \rightarrow \mathcal{F}$. The first row in Table 3 shows that the provers (PROTEIN, p-SETHEO with 2 processors, and SPASS) were able to detect most of these matches. However, in these cases the unnecessary axioms $\mathcal{A}$ decreased the recall substantially. E.g., p-SETHEO was able to prove 86% of these cases when the axioms were dropped. The next series of experiment considered the general case, retrieval of arbitrary matching components. For unsimplified proof tasks, the recall which was obtained with a run-time limit of 60 seconds was relatively low. No prover was able to find significantly more than 60% of the matches. Table 3 shows the detailed results for each prover in the columns labeled with *basic*. There are two major reasons for this result. First, these experiments were not carried out with the approximation of induction as described in Section 6.2. Previous experiments with SETHEO [SF97] and a smaller library showed that the overall recall could be increased by about 19% even for unsimplified tasks. Second, the pre-selection of axioms and lemmas is still too coarse. In particular, it is not independent of the other preprocessing and simplification steps. Table 4 shows results obtained by SPASS on four variants of the proof tasks. Two versions only contain a minimal set of axioms; for the others useful lemmas have been added. Furthermore, the variants differ in the way the predicates have been pre-processed: in the "expanded" case, all occurrences of the predicates *not_equal* and $<, >, \geq$ have been replaced by appropriate definitions which use $=$ and $\leq$ only. Table 4 reveals that the recall can be increased when the definitions are expanded, i.e., the number of different predicate symbols is reduced. In this case, however, the additional lemmas are counter-productive, i.e., they increase the search space without improving the overall recall.

	PROTEIN		p-SETHEO		SPASS		competition
Task type	Basic	Best	Basic	Best	Basic	Best	Best
r_{id}	50 %	65 %	71 %	66 %	89 %	82 %	94%
recall r	42.2%	56.6%	47,1%	55.9%	61.1%	71.2%	80.1%
σ_r	0.35	0.36	0.37	0.38	0.36	0.41	0.30
precision p	100 %	100 %	100 %	100 %	100 %	100 %	100 %

Table 3: Results of ATP-based confirmation filters

Both tables show that different variants of proof tasks and different provers exhibit quite a different behavior. Since our main interest is to obtain a high recall while keeping the answer-times low, parallel competition is of interest. When all provers are running different variants in a parallel competitive mode (similar to p-SETHEO), significantly better results can be obtained, as the simulated results in the last columns of both tables show. With such a technique, we were able to get an overall recall r of 80% which is high enough for practical purposes.

	minimal axioms		axioms + lemmas		competition	
expansion	yes	no	yes	no	yes	no
r_{id}	91%	89%	76%	76%	91%	89%
recall r	66.4%	61.1%	64.8%	61.7%	71.1%	66.5%
σ_r	0.35	0.36	0.41	0.40	0.35	0.36
precision p	100 %	100 %	100 %	100 %	100 %	100 %

Table 4: Detailed results of the SPASS-based filter

8 Related Work

Reuse by contract is an attempt to lift the axiomatic programming style of Hoare [Hoa69] into the large. It is thus a combination and extension of the "Systematic Software Development" [Jon90] and the "Design by contract" [Mey92] approaches. Similar work [LW94] is done in component-oriented programming which has recently emerged as a new subfield in software engineering [WBS97, LS97].

Signature matching has originally been developed as a library retrieval method for functional programming languages. Runciman and Toyn [RT91] define a partial order on polymorphic types and then use unification to retrieve components with equivalent or more general types. Rittri [Rit91, Rit90, Rit93] explored the strong connections between typed polymorphic lambda calculi and Cartesion closed categories (CCC). His system uses equational matching with respect to the theory *CCC* and is thus semantically stronger motivated than the one of Runciman and Toyn. However, since *CCC*-unification is undecidable [NPS93] he has to resort to the decidable sub-theory of *linear isomorphisms*, thus losing some of his foundations. Matthews [Mat92] extended Rittri's work by sorts to handle Haskell's type classes. DiCosmo [DiC95] introduced the notion of *definable isomorphisms* which can be used to reconstruct the necessary mapping between the identified types. Moorman and Wing [ZW95a] work with *match predicates* which are essentially Prolog-implementations of simple inference rules used in theory unification. The predicates can be combined to obtain different filters but Moorman and Wing do not discuss any decidability questions which may result from this combination (cf. [SS90] for the combination of unification algorithms). Their work also lacks any semantic foundations but is pragmatically motivated. Stringer-Calvert [SC94] has applied this framework to Ada.

Most early publications on deduction-based retrieval ignored the usability and scaling problems. We will thus discuss only more recent work; for a more general overview see [MMM98]. Moorman and Wing [ZW95b] have investigated specification matching in a slightly more general framework than ours but their main application area is also software reuse. They use the Larch/ML specification language for component description and the associated interactive Larch prover for retrieval. Mili et al. [MMM94] describe a system in which specifications are given as binary relations of legal (input, output)-pairs. They then define a subsumption relation on such relations and use this for retrieval, relying on Otter as the deductive component. However, their system is still in a prototypical stage, so no relevant statistical evaluation is presented. The work of Jeng and Cheng [JC94] also uses a subsumption test but to construct a hierarchical library. Scaling problems have been addressed by various authors. The Inscape/Inquire-system [PP93] limits the specification language to make retrieval more efficient. Similarly, AMPHION [LP$^+$94] uses a GUI to foster a more uniform specification style which in turn allows an appropriate fine-tuning of the prover. Additional speed-up is achieved by automatically "compiling" axioms into decision theories [LV95]. These techniques have successfully been applied to assemble over 100 FORTRAN programs from a scientific component library for solar system kinematics. Penix et al. [PBA95] use "semantic features" (i.e., user-defined abstract predicates which follow from the components' specifications) to classify the components and perform case-based reasoning along this classification to identify the most promising candidates. This classification process uses forward reasoning with an ATP. However, the authors give no evidence of how successful their approach is.

9 Lessons Learnt

In this chapter, we have presented the deduction-based software component retrieval tool NORA/HAMMR. From the very beginning we have considered it as an application project with strong practical emphasis. Our goal was to show that such a tool is not only theoretically possible but practical with state-of-the-art theorem provers. We can summarize our experience as follows:

1. Deduction-based software component retrieval is accepted only if any ATP evidence is hidden behind a user-friendly interface with a problem-oriented specification language; and if strong run-time constraints ("results while-u-wait") are met.

2. Only a pipeline of different signature, rejection and confirmation filters prevents the theorem prover from drowning.

3. Parallel competition between different provers and prover parameters is essential.

4. The application side (user interface, transformation of tasks into first order formulas) is at least as hard and as important as the deduction side; automated theorem proving alone is no silver bullet.

5. ATP researchers must pay more attention to the customers' point of view and provide convenient input formalisms as well as preprocessors for simplification.

6. We need fast methods to discover that a formula is not valid. Model checking is of limited value for infinite domains.

7. Experiments show that today's ATP technology makes deduction-based software component retrieval an almost realistic option in a formal development process.

In developing NORA/HAMMR, we had to deal with much low-level details. Much time was spent to transform syntactical representations of the formulas between the systems. A major problem was to generate the proof tasks from the VDM-SL specifications, due to the enormous syntactic variety. Much to our surprise, the translation from LPF to FOL was no major impediment after the optimizations. The naïve translation, however, increased the formula sizes considerably and brought retrieval to a grinding halt.

The integration of the model generators and theorem provers was easy. They can be considered as black boxes which receive a proof task in sorted first-order logic together with a run-time limit, after which they they return either *valid, not valid*, or *time-out*.

The implementation of necessary simplifications took more effort than one can usually afford in an application. Similarly, the administration of lemma libraries is an expensive task. Unfortunately, we were not able to reuse the implementation of Reif and Schellhorn (cf. Chapter III.2.11) because it is tailored towards equational logics.

A practical system must be evaluated in depth. Most other approaches in this area (cf. Section 8) are tried out with a handful of toy examples only. For the evaluation of NORA/HAMMR we spent much effort to develop libraries of reasonable size and complexity, resulting in a statistically significant body of experimental data. The evaluation of the system and its fine-tuning consumed large amounts of computation resources.

10 Future Work

Although the results of the experiments carried out so far are very encouraging, many improvements are required before NORA/HAMMR can really be used in industry. We are currently preparing experiments with a library of commercial date and time handling functions as used e.g. in stock trading software.[7]

Due to the hard time-constraints, the reduction of proof tasks, both in complexity and number, is of central importance. 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. Future work will thus include experiments with even more aggressive simplification techniques than those described in Section 5.1. We will also develop rejection filters based on decision procedures. Additionally, heuristics and knowledge-based filters similar to [PA97] can help to reduce the number of proof tasks.

As our experiments showed, current high-performance ATPs are certainly usable as confirmation filters. Much work, however, is still necessary to adapt them for such kinds of proof tasks. In particular, the requirement of full automatization and the strong time-limits must be obeyed carefully. Further important issues are the handling of inductive proofs, and the selection of appropriate axioms. Here, more powerful heuristics must be developed. A further reduction of the search space could be achieved by axiom compilation techniques similar to those of Meta-AMPHION [LV95]. However, the integration of decision procedures into the provers at hand is still an open research topic.

Acknowledgments

We are indebted to our colleagues in the DFG-Schwerpunkt for support with the experiments, particularly to Christoph Weidenbach, Peter Baumgartner, Michael Kühn, Ingo Dahn, and the other members of the ILF, PROTEIN, and SETHEO groups. Ralf Haselhorst of the DG Bank, Frankfurt, helped with the date library. Several people in Braunschweig contributed to the implementation of NORA/HAMMR, particularly Christian Lindig and Torsten Neumann, who worked on the GUI.

[7]This work is done in cooperation with the German DG Bank.

References

[AF97] S. Agerholm and J. Frost. "An Isabelle-based Theorem Prover for VDM-SL". In *Proceedings of the 10th International Conference on Theorem Proving in Higher Order Logics (TPHOLs'97)*, Lecture Notes in Computer Science. Springer, August 1997.

[BCJ84] H. Barringer, J. H. Cheng, and C. B. Jones. "A Logic Covering Undefinedness in Program Proofs". *Acta Informatica*, **21**(3):251–269, October 1984.

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

[BG+91] B. Beckert, S. Gerberding, R. Hähnle, and W. Kernig. "The Tableau-Based Theorem Prover ${}_3T^AP$ for Multiple-Valued Logics". In D. Kapur, (ed.), *Proc. 11th Intl. Conf. Automated Deduction*, *Lecture Notes in Artifical Intelligence* **607**, pp. 758–760. Springer, July 1991.

[BG94] L. Bachmair and H. Ganzinger. "Rewrite-based Equational Theorem Proving with Selection and Simplification". *Joural of Logic and Computation*, **4**(3):217–247, 1994.

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

[BvH+90] A. Bundy, F. van Harmelen, A. Smaill, and A. Ireland. "Extensions to the Rippling-Out Tactic for Guiding Inductive Proofs". In Stickel [Sti90], pp. 132–146.

[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 ACM Symp. Principles of Programming Languages*, pp. 238–252, Los Angeles, California, January 1977. ACM Press.

[CJ91] J. H. Cheng and C. B. Jones. "On the usability of logics which handle partial functions". In C. Morgan and J. C. P. Woodcock, (eds.), *Proceedings of the Third refinement Workshop*, Workshops in Computing Series, pp. 51–69, Berlin, 1991. Springer-Verlag.

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

[DiC95] R. DiCosmo. *Isomorphisms of Types: from λ-calculus to information retrieval and language design*, *Progress in Theoretical Computer Science* **14**. Birkäuser, Boston, 1995.

[DM82] L. Damas and R. Milner. "Principle Type-Schemes for Functional Programs". In *Proc. 9th ACM Symp. Principles of Programming Languages*, pp. 207–212, Albuquerque, New Mexico, January 1982. ACM Press.

[FN87] W. B. Frakes and B. A. Nejmeh. "Software Reuse Through Information Retrieval". In *Proc. 20th Annual Hawaii Intl. Conf. on Systems Sciences*, pp. 530–535, January 1987.

[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.

[GMW97] H. Ganzinger, C. Meyer, and C. Weidenbach. "Soft Typing for Ordered Resolution". In W. McCune, (ed.), *Proc. 14th Intl. Conf. Automated Deduction*, *Lecture Notes in Artifical Intelligence* **1249**, pp. 321–335. Springer, July 1997.

[Hoa69] C. A. R. Hoare. "An Axiomatic Basis for Computer Programming". *Communications of the ACM*, **12**(10):576–580, October 1969.

[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.

[JC94] J.-J. Jeng and B. H. C. Cheng. "A Formal Approach to Using More General Components". In *Proc. 9th Knowledge-Based Software Engineering Conf.*, pp. 90–97, Monterey, CA, September20-23 1994. IEEE Computer Society Press.

[JJ$^+$91] C. B. Jones, K. D. Jones, P. A. Lindsay, and R. Moore. *mural: A Formal Development Support System.* Springer, 1991.

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

[JM97] J.-M. Jézéquel and B. Meyer. "Design by Contract: The Lessons of Ariane". *IEEE Computer*, **30**(1):129–130, January 1997.

[Jon90] C. B. Jones. *Systematic Software Development Using VDM.* Prentice-Hall, Englewood Cliffs, New Jersey, second edition, 1990.

[Jon95] C. B. Jones. "Partial functions and logics: A warning". *Information Processing Letters*, **54**(2):65–67, April 1995.

[KBS95] *Proc. 10th Knowledge-Based Software Engineering Conf.*, Boston, MA, November 12-15 1995. IEEE Computer Society Press.

[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.

[Lio96] J. L. Lions et al. Ariane 5 Flight 501 Failure Report, 1996.

[Lov78] D. W. Loveland. *Automated Theorem Proving: a Logical Basis.* North–Holland, 1978.

[LP$^+$94] M. Lowry, A. Philpot, T. Pressburger, and I. Underwood. "AMPHION: automatic programming for scientific subroutine libraries". In Z. W. Raś and M. Zemankova, (eds.), *Proc. 8th Intl. Symp. on Methodologies for Intelligent Systems*, *Lecture Notes in Artifical Intelligence* **869**, pp. 326–335. Springer, October 1994.

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

[LS97] G. T. Leavens and M. Sitaraman, (eds.). *Proc. ESEC/FSE Workshop on Foundations of Component-Based Sytems*, September 1997.

[LV95] M. Lowry and J. Van Baalen. ”Meta-Amphion: Synthesis of Efficient Domain-Specific Program Synthesis Systems”. In KBSE-10 [KBS95], pp. 2–10.

[LW94] B. Liskov and J. M. Wing. ”A Behavioral Notion of Subtyping”. *ACM Transactions on Programming Languages and Systems*, **16**(6):1811–1841, November 1994.

[Mat92] B. Matthews. ”Reusing Functional Code using Type Classes for Library Search”. In *Proc. ERCIM Workshop on Methods and Tools for Software Reuse*, Heraklion, 29.-30. 10. 1992.

[MBK91] Y. S. Maarek, D. M. Berry, and G. E. Kaiser. ”An Information Retrieval Approach For Automatically Constructing Software Libraries”. *IEEE Transactions on Software Engineering*, **SE-17**(8):800–813, 1991.

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

[McI68] M. D. McIlroy. ”Mass Produced Software Components”. In P. Naur and B. Randell, (eds.), *Software Engineering: Report on a Conference*, pp. 138–150, Brussels, 1968. NATO Scientific Affairs Division.

[Mey92] B. Meyer. ”Applying “Design by Contract””. *IEEE Computer*, **25**(10):40–51, October 1992.

[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”. *Journal of Automated Reasoning*, **18**:237–246, 1997.

[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.

[MMM98] A. Mili, R. Mili, and R. Mittermeir. ”A Survey of Software Reuse Libraries”. *Annals of Software Engineering*, 1998. To appear.

[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.

[NPS93] P. Narendran, F. Pfenning, and R. Statman. ”On the Unification Problem for Cartesian Closed Categories”. In R. L. Constable, (ed.), *Proc. 8th Annual IEEE Symp. Logic in Computer Science*, pp. 57–63, Montreal, Canada, June 1993. IEEE Computer Society Press.

[Obe62] A. Oberschelp. "Untersuchungen zur mehrsortigen Quantorenlogik". *Mathematische Annalen*, **145**:297–333, 1962.

[PA97] J. Penix and P. Alexander. "Toward Automated Component Adaptation". In N. Juristo, (ed.), *Proc. 9th Intl. Conf. on Software Engineering and Knowledge Engineering*, pp. 535–542. Knowledge Systems Institute, June 1997.

[PBA95] J. Penix, P. Baraona, and P. Alexander. "Classification and Retrieval of Reusable Components Using Semantic Features". In KBSE-10 [KBS95], pp. 131–138.

[PP93] D. E. Perry and S. S. Popovitch. "Inquire: Predicate-Based Use and Reuse". In *Proc. 8th Knowledge-Based Software Engineering Conf.*, pp. 144–151, Chicago, IL, September20-23 1993. IEEE Computer Society Press.

[Rit90] M. Rittri. "Retrieving Library Identifiers via Equational Matching of Types". In Stickel [Sti90], pp. 603–617.

[Rit91] M. Rittri. "Using types as search keys in function libraries". *J. Functional Programming*, **1**(1):71–89, January 1991.

[Rit93] M. Rittri. "Retrieving Library Functions by Unifying Types Modulo Linear Isomorphism". *Theoretical Informatics and Applications*, **27**(6):523–540, 1993.

[RT91] C. Runciman and I. Toyn. "Retrieving re-usable software components by polymorphic type". *J. Functional Programming*, **1**(2):191–211, April 1991.

[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.

[SC94] D. W. J. Stringer-Calvert. Signature matching for Ada software reuse. Master's thesis, University of York, March 1994.

[SF97] J. M. P. Schumann and B. Fischer. "NORA/HAMMR: Making Deduction-Based Software Component Retrieval Practical". In *Proc. Conf. Automated Software Engineering '97*, pp. 246–254, November 1997.

[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.

[SS90] M. Schmidt-Schauß. "Unification in a Combination of Arbitrary Disjoint Equational Theories". In C. Kirchner, (ed.), *Unification*, pp. 217–266. Academic Press, London, 1990.

[Sti90] M. E. Stickel, (ed.). *Proc. 10th Intl. Conf. Automated Deduction*, *Lecture Notes in Computer Science* **449**, Kaiserslautern, July 1990. Springer.

[WBS97] W. Weck, J. Bosch, and C. Szyperski, (eds.). *Proc. Second Intl. Workshop on Component-Oriented Programming*, Jyväskylä, June 1997. Turku Centre for Computer Science. TUCS General Publication No 5.

[WGR96] C. Weidenbach, B. Gaede, and G. Rock. "Spass and Flotter version 0.42". In M. A. McRobbie and J. K. Slaney, (eds.), *Proc. 13th Intl. Conf. Automated Deduction*, *Lecture Notes in Artifical Intelligence* **1104**, pp. 141–145. Springer, July/August 1996.

[ZW95a] A. M. Zaremski and J. M. Wing. "Signature Matching: A Tool for Using Software Libraries". *ACM Transactions on Software Engineering and Methodology*, pp. 146–170, 1995.

[ZW95b] A. M. 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.