Bernd Fischer

Deduction-Based Software Component Retrieval

genehmigt als Dissertation zur Erlangung des Doktorgrades in den Naturwissenschaften

Fakultät für Mathematik und Informatik
Universität Passau

November 2001

Datum der mündlichen Prüfung: 1. Juni 2001

1. Gutachter: Prof. Dr.-Ing. G. Snelting, Universität Passau
2. Gutachter: Prof. C. Lengauer, Universität Passau
3. Gutachter: Prof. P. Alexander, University of Kansas

Für Sylvie. Alles wird gut.

Acknowledgments

First and foremost, I thank my advisor Prof. Dr.-Ing. Gregor Snelting who never lost faith that I would eventually finish and who for so many years provided the environment in which I could do my research. I also thank Prof. Perry Alexander and Prof. Christian Lengauer who have enthusiastically agreed to serve as additional reviewers for this thesis.

I thank my colleagues in the "Schwerpunktprogramm", Thomas Baar, Peter Baumgartner, Ingo Dahn, Dirk Fuchs, Micha Kühn, Andreas Wolf and especially Johann Schumann and Christoph Weidenbach for their help with the experiments and for the support of their theorem provers. Christian Lindig and Thorsten Robschink helped with the implementation of NORA/HAMMR's GUI. Discussions and joint papers with Thomas Baar, Dirk Fuchs, Franz-Josef Grosch, Jens Krinke, Christian Lindig, Mike Lowry, Matthias Kievernagel, John Penix, Johann Schumann, and Werner Struckmann helped to shape the ideas in this thesis. Ali Mili and Johann Schumann read draft versions of parts of the thesis and helped to improve its presentation. Thanks a lot, guys!

Special thanks are due to my office mates Jens Krinke and Jon Whittle, who suffered probably more than their fair share from my constant grumbling, and to Klaus Havelund, John Penix, and Johann Schumann for their persistent reminders to get it finally done and of course to Klaus Havelund for all the coffee. Jens Krinke also handled the remote submission of this thesis. Finally, very special thanks to Sylvie Dieckmann who saw my work morph over the last six years.

The dissertation research has been begun and to a large extent been conducted while I was employed at the Technische Universität Braunschweig, Abteilung Softwaretechnologie; it has been finished while I was employed by the Research Institute for Advanced Computer Science (RIACS) at the NASA Ames Research Center. Funding for this research has been provided by the Deutsche Forschungsgemeinschaft (DFG) within the "Schwerpunktprogramm Deduktion", grants Sn11/2-3.

Abstract

Identifying appropriate software components in a library—or *software component retrieval*—is an important task in software reuse: after all, components must be found before they can be reused. *Deduction-based* retrieval uses formal specifications as component descriptors and as search keys and an automated theorem prover to check whether a component matches a query. It is thus the only component retrieval approach which delivers proven matches only; however, its computational effort is very high. This thesis contains a detailed theoretical investigation and the first substantial experimental evaluation of deduction-based software component retrieval.

The theoretical investigation develops an abstract view of component retrieval based only on the concept of sets of relevant, matching, and found components, respectively. It is shown how properties of abstract match predicates are reflected by these sets and vice versa and how this duality can be used to build library indexes. The concepts of closure under iterated retrieval and query stability are introduced and used to characterize retrieval algorithms. The notions of precision leverage and relative defect ratio are introduced and used for the evaluation and further characterization of retrieval algorithms. Relevance conditions and reuse effects for three different retrieval modes (i.e., exact, proper, and approximate retrieval) are identified and a variety of concrete match predicates for these modes are defined. Relations between these predicates are shown and the appropriate side conditions on component specifications and queries are identified.

The experimental evaluation is facilitated by an advanced prototype retrieval system called NORA/HAMMR which has been designed and implemented for this thesis. The experimental set-up (i.e., test library, proof task generation, and applied theorem provers) is described in detail. The main technical and non-technical requirements for a realistic retrieval system are discussed and the resulting design is outlined. NORA/HAMMR's novel architecture is based on a pipeline of filters of increasing deductive strength. Dedicated rejection filters are used "upstream" to rule out non-matches as early as possible and thus to prevent the "downstream" confirmation filters from overflowing. This pipeline architecture guarantees that intermediate results of acceptable precision are available for inspection almost any time. It is amenable to parallelization which decreases the response times and increases the recall of the system.

The experiments have shown that relatively simple techniques are sufficient to identify large number of non-matches quickly and cheaply. Rejection filters based on term rewriting are used to simplify the proof tasks and to reduce them eventually to *false*, thus exposing trivial non-matches. Rewrite-based quantifier elimination techniques increase their efficiency further and reduce the fallout of the answer set to almost 25%. Non-trivial non-matches can be identified via the application of specific counterexamples. In combination with the quantifier elimination, rewrite-based implementations of counterexamples prove to be very

effective, reducing the fallout of the answer set even further to less than 15%.

This fallout level makes possible the application of fully automatic, off-the-shelf theorem provers for first-order logic. However, the experiments revealed that the applied provers (GANDALF, OTTER, SETHEO, and SPASS) were not yet mature enough to work directly on the automatically generated proof tasks. Even with a timeout of $T_{max} = 90.0$ secs. per individual proof task, the provers achieve overall recall levels of only 45%–70%. The recall can be improved significantly by proof task preprocessing. This thesis describes the effects of a pre-simplification of the conjectures and of a simple signature-based heuristic to simplify the background theory by identifying a suitable subset of axioms and lemmas. Both techniques improve the prover performances; moreover, they are to a certain extent complementary and their combination yields additional improvements, resulting in overall recall levels of 65%–85%.

NORA/HAMMR's pipeline architecture allows the users to realize different recall level/response time trade-offs via a "plug'n'play"-style combination of the filters into different pipelines. The alternatives span the range from a single-processor pipeline which achieves an overall recall of almost 74% within a guaranteed response time of less than one second per library component to a multi-processor pipeline with average response times more than 25 times as high but an overall recall level of more than 91%. This demonstrates that deduction-based retrieval has become technologically feasible.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Software Engineering, Reuse, and Retrieval

Software engineering as a discipline of computer science emerged some twenty years after the invention of the digital computer (cf. [NR68, BR69]) as a reaction to some truly spectacular failures in the development of large software systems (cf. [Bro75] for a prime example.) In analogy to the already established, "classical" engineering disciplines it had initially been defined as

> "[t]he establishment and use of sound engineering principles in order to obtain economically software that is reliable and works efficiently on real machines."
>
> F. L. Bauer, in [NR68]

One of these "sound engineering principles" is to build new artifacts—prototypes, products, or systems—on top of other, existing artifacts. And although most of the early research had been geared towards the development of software "from scratch," the importance of this principle for software engineering had been realized from the very beginnings of the discipline (cf. M. D. McIlroy's contribution "Mass Produced Software Components" [McI68] to the seminal NATO conference.)

However, in the following it became clear that the implementation of this principle in software engineering, which gradually became to be known as *software reuse* [Kru92], was a hard long-term task that is not yet solved satisfactorily: even today, software reuse can by no means be considered an established practice.

To make software reuse happen, a wide variety of aspects and issues has to be addressed, ranging from purely organizational (e.g., the so-called "not-invented-here syndrome") to purely technical. In this thesis I will address technical issues only.

The most basic technical issue addresses the nature of the artifacts: what can really be reused? Almost anything, as C. W. Krueger's survey [Kru92] shows, source code ranging from small snippets to entire subsystems but also test plans, design decisions, or documentation. By slightly stretching their meanings, this can be fitted into the following two categories:

- *domain knowledge* or
- *source code fragments*.

Reusing domain knowledge promises the higher returns but also requires the higher up-front investments, for two reasons. First, the knowledge to be reused must be collected and compiled. This process is called *domain analysis* [PA89]. Its importance has long been recognized ("It all comes back to domain analysis", I. D. Baxter, in [Tra88a]) but it is a difficult process because it requires a deep understanding of the analyzed domain, as the *3-System Rule* points out:

> "If you have not built three real systems in a particular domain, you are unlikely to be able to derive the necessary details of the domain required for successful reuse in that domain. In other words, the expertise needed for reuse arises out of 'doing' and the 'doing' must come first."
>
> T. Biggerstaff, in [Tra88a]

Second, this "raw" domain knowledge is not ready for reuse—it is not an artifact. It is not tangible, not even by software engineering standards, and it exists only as a conceptual model within the brains of the domain analysts.

The most basic and archetypical technique to make domain knowledge explicit and thus reusable is to write it down as a glossary of terms but more advanced methods have also been developed. *Domain-specific software architectures* cast the gained understanding into a typical system design [GS92, SG95]. *Application frameworks* may be considered as partially implemented architectures where only the critical parts are provided by the framework developer while all application-specific parts are implemented and filled in by an application developer. *Software generators* [Cle86] also embody domain-specific architectures but in contrast to frameworks, the application-specific parts are specified in a high-level language. The generator system then translates the specification and blends it with a pre-implemented core. This extra level of abstraction accounts for high reuse leverage factors.

All these domain-specific reuse approaches operate on the system level, i.e., complete systems or subsystems, and thus aim at large-grain reuse. *Code reuse* operates on lower abstraction levels, trading large reuse leverage factors for a much wider range of applicability. Its most basic and archetypical technique is called *code scavenging* [Kru92], i.e., copying arbitrary code fragments out of their former contexts. Although this is always possible and has virtually no

up-front costs, more systematic techniques offer more reliability and better leverages. These techniques depend on a central, well-maintained artifact repository or *software component library*. The exact nature of the components is immaterial (and still subject to discussions, cf. e.g., [WBS97, LS97]) and also depends on the applied programming language. Possible components are, e.g.,

- functions, procedures, and modules for procedural languages,
- functions, abstract datatypes, and modules for functional languages,
- methods, objects, and classes for object-oriented languages.

However, the mere existence of a component library does not automatically entail its re-use—the passive library needs active promotion. The most important aspect of this promotion is to provide prospective re-users with support to help them identifying components which suit their needs and the canonical way to provide this support is to develop tools which mechanize the identification process. This task is the topic of *software component retrieval*.

1.1.1 Information Retrieval Approaches

Software component retrieval can informally be characterized as "information retrieval meets software reuse." Historically, it evolved from attempts to apply general information retrieval methods (cf. Section 2.1 or [Fra92] for short overviews and [vR79, SM83] for general introductions) to software component libraries. Here, the basic assumption is that software components are, after all, nothing else but a certain variant of text documents and, hence, the text-based methods are applicable.

Information retrieval has developed a variety of methods which form a kind of continuum bounded by the two predominant schools "controlled vocabulary" and "free vocabulary." Figure 1.1 which is adapted from [FG89] shows the most common methods in this continuum.

Controlled vocabulary methods place limits on the legal terms for the classification of the objects and formulation of queries. Often a *thesaurus* is used to structure the vocabulary along a variety of hierarchical, equivalence, and association relations, e.g., the Booch-taxonomy [Boo87]. These relations directly lead to classification schemes. In an *enumerated classification* schema, e.g., *Dewey Decimal Classification*, the domain is broken into disjoint, hierarchically ordered classes. The components are then associated to the leafs of the class tree. But due to its hierarchical structure enumerated classification requires a full understanding and a complete and rigid coverage of the domain. Evolving domains as for example component libraries are thus better handled by a *faceted classification* schema [Pri87, Pri91, MR87, MR90]. It does not superimpose a single tree-shaped hierarchy on the domain but allows a classification along different

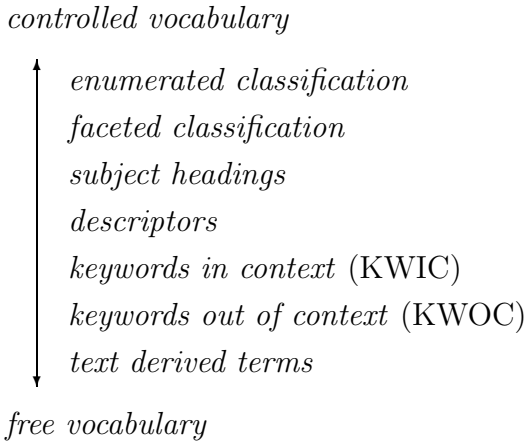


Figure 1.1: Classification of information retrieval methods

dimensions or *facets* and is thus easier to modify. Here, the components are organized in a fashion similar to a relational database where the facets correspond to the database schema. More elaborated variants refine the facets hierarchically [Bör95] or even use a formal description logic to represent the components and rely on automated classifiers to organize the library, e.g., the LaSSIE-project [DB+91, DJ94].

All classification methods help the user to understand and to navigate within the domain—they provide natural search methods which are amenable to automation. But their major drawback are the high up-front costs which are associated with the development of a classification scheme and indexing the components. *Controlled keyword* methods reduce these costs by more liberal indexing. In their barest form, each component is associated with an unstructured collection of descriptors which are drawn from a thesaurus. An intermediate form uses hierarchically ordered *subject headings*, e.g., the *Computing Reviews Subject Headings*, to structure the keywords.

Obviously, all controlled vocabulary methods require an established and well-defined collection of technical terms. But such vocabularies need not to exist for different reasons:

- There is no consensus about the terms.
- Evolving domains generate new terms which are not yet represented in the vocabulary.
- Local or technical sub-vocabularies convey meaningful information about some components but are unimportant from a global point of view.

Free vocabulary methods circumvent this problem by using an unlimited vocabulary; however, to prune the size of the vocabulary, a variety of linguistic al-

gorithms must be applied, e.g., *stemming* (i.e., reduction to lexical roots) or *phonetic matching* (i.e., taking phonetic similarities into account.) The most basic approach uses a natural language parser to extract a set of single keywords from the program text or its accompanying documentation [FN87]. Better results can be achieved if more sophisticated indexing techniques, e.g., *lexical clustering* [MS89, MBK91], or matching algorithms, e.g., *spreading activation* [Hen94, Hen96], are used. It is also possible to combine these techniques with structural information extracted from the library, e.g., inheritance relations [HM91].

1.1.2 Dedicated Component Retrieval Approaches

Dedicated component retrieval approaches deny the basic assumption of the general information retrieval approaches that software components are "ordinary" texts. More precisely, they assume that software components are highly formalized artifacts and thus have intrinsic properties which are not obvious from the textual representation but are more characteristic than the textual representation itself. Consequently, to achieve good retrieval results, these intrinsic properties must be exploited which in turn requires specialized software component retrieval algorithms.

Such algorithms were developed in different areas, depending on the property which was to be exploited. In automatic program understanding, e.g., in the Programmer's Apprentice project [RW88], *structural* representations of the components which abstract from their lexical appearance are used, e.g., program dependency graphs. A specialized match procedure called *graph parsing* [RW90a] handles irrelevant syntactic differences. The major drawback of such structural approaches is that the representation mechanisms are necessarily much too detailed and concrete. Hence, queries are difficult to formulate and internal representation aspects which are hidden in the library components may escape their scope.

These problems can be alleviated if more abstract representations are exploited. In *signature matching*, the types of the applied programming language are used: a component is retrieved if its type is "compatible" under the applied type discipline to the query. This approach originated with the work of M. Rittri [Rit89, Rit91] in the domain of functional programming. A. Moorman Zaremski and J. Wing were probably the first to investigate signature matching under a software engineering perspective [MW93, MW95a]. Its basic assumption is that types reflect the functionality of a component at the right level of abstraction, without exposing its internal representation.

The main conceptual difficulty in signature matching is an adequate definition of "compatible" types which abstracts away "irrelevant" implementation details. A component should be retrieved even though its type does not match exactly but is "similar enough". For example, in functional languages the difference between

a tupled and a curried function is considered to be minor. Obviously, the choice of the type compatibility is crucial for the behavior of a retrieval tool: if too many types are identified, its precision suffers, if too few types are identified, its recall suffers.

For functional languages, the equational theory Γ which describes the axioms of Cartesian closed categories (cf. [Rit89, Rit91, DiC95]) proved to be adequate: a component is retrieved iff its type is Γ-equal to the query. For imperative languages, no such semantically justified theory has been found and more ad-hoc solutions are pursued [SC94]. This basic approach does not exploit polymorphism to the fullest possible extent. Since a polymorphic function (e.g., of type $\forall\alpha,\beta\cdot(\alpha\to\beta)\to\mathsf{list}(\alpha)\to\mathsf{list}(\beta)$) also works on more specific types (e.g., $(\mathsf{int}\to\mathsf{int})\to\mathsf{list}(\mathsf{int})\to\mathsf{list}(\mathsf{int})$) it should also be retrieved for more specific queries. This effect can be achieved if Γ-equality is replaced by Γ-matching [Rit90]: a component is retrieved iff its type can be Γ-instantiated to the query. C. Runciman and I. Toyn [RT89, RT91] proposed an approach in which the query variables can also be instantiated, making them *don't care*-nondeterministic or "wild cards". Rittri also applied this idea to his own context [Rit93] but since Γ-unification is—in contrast to Γ-matching—undecidable [NPS93], he had to drop the distributive axioms.

While signature matching is still based on the *syntax* of the components, *specification matching* or *deduction-based software component retrieval* exploits their formal *semantics*: a component is retrieved if its *behavior* is provably "compatible" to the query. Here, formal specifications are used to characterize components and as queries such that this approach amounts to *proving* the equivalence of specifications (or some other well-defined relation between two specifications) formally.

Deduction-based software component retrieval is based on an obvious idea but it is very difficult to implement practically. It has thus been (re-) invented periodically, in different contexts and under various aspects. It has been introduced as a retrieval mechanism in the context of integrated software development environments, e.g., in the Inscape/Inquire [Per89, PP93a] or PARIS [KRT87] systems. Later, the focus shifted to the more technical aspects, especially to the proof process. G. Rollins and J. Wing [RW91] presented a prototype system which used the (higher-order) inference machine of λ-Prolog. S. Manhart and S. Meggendorfer [MM91] appear to be the first who built a system on top of a high-performance, automated theorem prover for first-order logic (i.e., SETHEO [LS$^+$92]).

Subsequently, it was realized that raw deductive power is not sufficient; consequently, the focus of the more recent work shifted back to the organizational aspects. A. Moorman Zaremski and J. Wing [Moo96, MW95b, MW97b] investigated different match relations. R. Mili et al. [MMM94, MMM97] as well as B. H. C. Cheng and J. Jeng [JC93] proposed hierarchical library organizations based on orderings of the components which speed up the retrieval pro-

cess. J. Penix et al. [PBA95, Pen98] discuss a slightly different scheme based on formally specified features. A different line of research linked deduction-based retrieval to refinement of algebraic specifications [SL91, Ste91, Gog85, GN+96].

Deduction-based software component retrieval has a unique conceptual advantage over all other component retrieval methods—it is the only method which can (in principle) guarantee that all components retrieved in response to a particular query actually do what they are required to do. In short, it is the only method which retrieves proven matches only. This makes deduction-based software component retrieval particularly suitable for the development of high-reliability or safety-critical applications, e.g., automatic stock option trading systems or space craft control systems. Moreover, this property allows a combination with other formally justified software development approaches (e.g., deductive program synthesis) without compromising the integrity of the resulting combined approach.

Finally, *behavior sampling* makes use of the most striking difference between ordinary text documents and software components, the executability of the components. Here, the basic assumption is that already a sample of a few *(input, output)*-pairs characterizes a component sufficiently. Hence, a component is run on a given set of inputs, the outputs are collected and compared to the expected results. If the correspondence is high enough, the component is retrieved. A. Podgurski and L. Pierce [PP92, PP93b] proposed this idea first. Their system uses a randomly selected inputs which are derived from a probabilistic input distribution. Park and Bais [PB97] use the inductive structure of the input domain to generate the sample. R. Hall [Hal93] describes a system tailored towards the Ada programming language; there, inputs are specified by the user. However, in practice all behavior sampling methods involve non-trivial up-front costs because the sampling process requires a controlled environment which deals with side effects, program crashes, timeouts, etc.

1.1.3 Code Reuse and Component Retrieval

In analogy to the software testing terminology, code reuse is traditionally labeled as either *black box* or *white box*, depending on whether the code needs to be inspected and modified or not. Code scavenging for example is the paradigmatic white box method. In principle, software component retrieval can support both styles. However, here I use a slightly finer classification.

- In *black box reuse*, a client may reuse the retrieved components "as is", without any further inspection, proviso, or modification.

- In *client-adaptive grey box reuse*, a client may still reuse the retrieved components "as is" and without any modification but only after having met additional conditions.

- In *component-adaptive grey box reuse*, a client may reuse the retrieved components without meeting any additional conditions but only after interface-level modifications of the components.
- In traditional *white box reuse*, arbitrary additions and modifications either on the client side or on the component side are required.

In practice, however, these different styles are not always as sharply distinguished as the definitions suggest. For example, if the retrieval algorithm can automatically provide the necessary interface-level modifications (e.g., rearranging formal parameters), component-adaptive grey box reuse may also qualify as black box style.

1.2 Scope and Assumptions

Although this thesis is set in the context of software reuse, I discuss reuse aspects almost only from the retrieval perspective, e.g., how different retrieval configurations affect the reusability of the retrieved components. Moreover, I am not concerned with any activities which happen before or after retrieval. Thus, I do not consider any of the steps necessary to set-up deduction-based retrieval, e.g., library construction (i.e., collection, selection, and homogenization of components) and component indexing (i.e., attaching specifications to the components); similarly, I do not consider any post-processing steps, e.g., adaptation and composition of retrieved components. Finally, I am not concerned with the source of the queries; the experimental setup, however, takes the possibly different characteristics of different query sources (e.g., direct query input by a user or automatic query generation by an integrated system) into account and uses a query set which covers a wide range of different specification styles.

For the purpose of this thesis, I consider the defining property of a component to be a self-contained entity which provides at an atomic "point-of-service" (e.g., function call) a single, sharply defined, and independent functionality. This point of view essentially confines components to functions, procedures, and methods and excludes what has been termed *module matching* in [Moo96, MW97b] from the scope of the thesis. But this is no real omission for two reasons:

- If a module is just considered a collection of independent points-of-service, as in [Moo96, MW97b], module matching can be reduced to component retrieval in a straightforward manner, provided that proper attention is paid to cardinality questions.
- If a module has some relevant internal structure in addition to its points-of-service, e.g., *software schemes* [VK85, KST97] or *design patterns* [GH+96], then retrieval is no longer sufficient for reuse and adaptation and composition aspects begin to dominate.

In general, a retrieval method can be considered to be deduction-based if it satisfies the following three criteria:

1. Component indexes and queries are expressed in a formal language.
2. Matching is a formally defined relation between indexes and queries.
3. Retrieval employs formal reasoning to establish the validity of the match relation.

Obviously, the essential aspect here is formality. However, to prevent that the informal information retrieval approaches are also subsumed under this description, the criteria are usually understood with the side condition that the index/query language is sufficiently expressive, e.g., to formulate the match relation. In practice, the designation deduction-based retrieval has thus been restricted to approaches which apply algebraic or, more often, axiomatic specifications. I am a proponent of the second approach because I believe that it offers *for the purpose of component retrieval* three important advantages:

- $(pre, post)$-pairs conform better to the *single point-of-service* view of components—in algebraic specifications the relevant information can be scattered over arbitrarily many equations.
- $(pre, post)$-pairs can naturally be considered *component contracts* which allows for more a intuitive interpretation of the match relations.
- $(pre, post)$-pairs make it easier to assign *multiple indexes* to a single component—again, in algebraic specifications the relevant information can be scattered over arbitrarily many equations.

In the following, I use a functional subset of VDM-SL [A$^+$93, Daw91, PL92] as *contract language*. The restriction to a functional subset is justified by the assumption that indexes are *essentially functional*: side effects are not in agreement with the single point-of-service view and must be made explicit, e.g., through additional parameters. Similarly, parameters must be either of type *in* or *out*, but not both, and *inout*-parameters must be split appropriately. Some additional assumptions on the component specifications are made explicit in Section 2.2.4.

1.3 Goals

It should be clear from the brief survey given in Section 1.1.2 that deduction-based component retrieval is not a new idea. However, it should also be clear that it is not yet a practical and established technique. This raises the question of whether deduction-based component retrieval can be made practical at all or whether it is just an interesting failure. Prior research has ignored this important

question almost completely; it is even unknown whether (much less how) current theorem provers can handle the deductive load of component retrieval at all. In this thesis, I will argue that (and show how) this is the case, and, hence, that deduction-based component retrieval can in fact be made practical. This gives rise to a first concrete research goal:

Goal #1: Evaluate under which circumstances and to which extent current fully automatic, off-the-shelf theorem provers for first-order logic are capable of solving the emerging proof tasks.

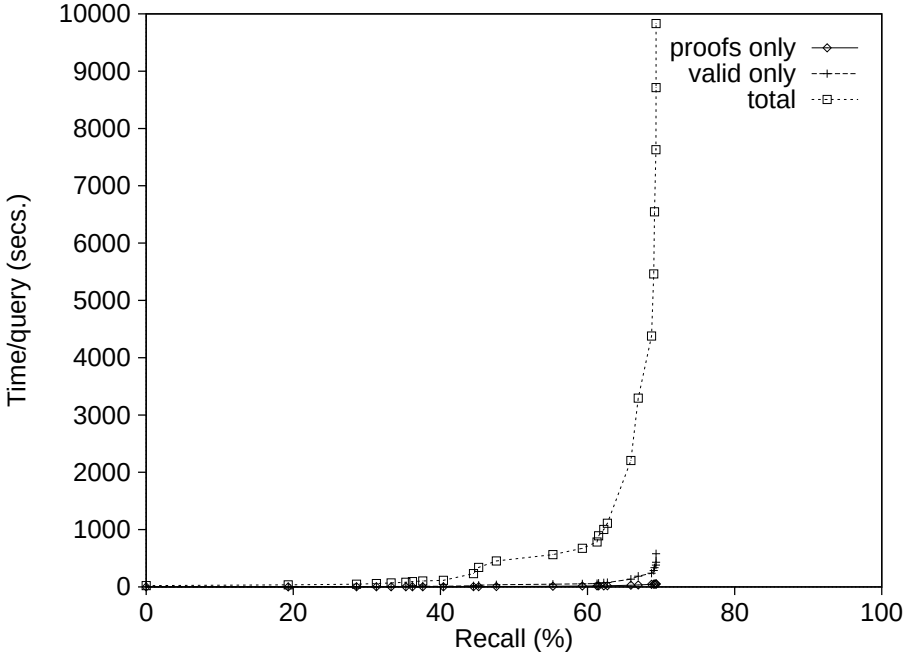


Figure 1.2: Response times over recall

Figure 1.2 helps to explain why practicability becomes a concern and how it can be addressed. It shows for three different scenarios the average response times per query over the average recall level (i.e., the percentage of retrieved matches), achieved using SPASS as the best available off-the-shelf theorem prover. The first scenario (labeled *total*) corresponds to the simple generate-and-test approach often applied in previous research, i.e., for each component in the library the proof task is generated from query and component and its validity is tested using the prover. It clearly demonstrates that practicability really is a concern. Even for moderate recall levels, e.g., 50%, response times are already up to 10 minutes per query; better recall levels induce exponentially growing response times. Worse yet, the maximal recall level achieved on average is only approximately 70% which means that in many cases no matching components are found at all. The second scenario (labeled *proofs only*) shows how little time the prover actually spends on

the tasks where it is successful (i.e., can find a proof). Hence, if we could restrict the application of the prover a priori to the cases where it will be successful, we would have succeeded in making deduction-based retrieval practical. However, due to the undecidability of first-order logic, this scenario is unrealistic. The third scenario (labeled *valid only*) is an approximation of the optimal second scenario where all invalid proof tasks (which are associated with non-matches) have been removed. While it is still unrealistic—again due to the undecidability of first-order logic—we have a much better chance of approximating it by filtering out as many invalid tasks as fast as possible. This scenario (or any good approximation thereof) still exhibits the exponential growth in response times but to a much lesser—and now tolerable—extent. However, the maximally achieved recall level is obviously the same as in the simplistic scenario because the tasks ultimately fed into the prover are the same. This observation gives rise to the next research goal:

Goal #2: Develop and evaluate methods to increase the performance of off-the-shelf theorem provers for first-order logic when applied to the emerging proof tasks.

These individual methods to increase prover performance, however, are only some of the building blocks for a complete retrieval *system*. The overarching practical goal of this thesis is to build such a system, more precisely:

Goal #3: Build a prototype retrieval system which demonstrates the feasibility of the approach.

In addition to pursuing these more practical goals, I also try to address in this thesis some of the theoretical issues of deduction-based software component retrieval which remain open despite the mostly theoretical nature of previous research, in particular:

Goal #4: Understand the nature of the software component retrieval process in general and the additional problems of the deduction-based approach in particular.

Goal #5: Understand the interdependencies between formal specifications, retrieval and reusability.

To the best of my knowledge, none of these goals has been achieved with previous research. I will discuss related work throughout the thesis and again in Section 10.2; any serious experimental evaluation is (with the exception of J. Penix's work [Pen98, PA99] which is based on the experiments presented here) notoriously absent from the literature.

1.4 Outline

The remainder of this thesis is conceptually organized into three main parts: *Reuse by Contract* (chapters 2 to 4), *Rejection Techniques* (chapters 5 and 6), and *Specification Matching* (chapters 7 to 9). The first part is essentially of theoretical nature and addresses mostly goals #4 and #5; Chapter 4, however, discusses the decisions which led to the design NORA/HAMMR and thus addresses goal #3. The second and third parts contain the experimental results of this thesis and address goals #1 and #2. Chapter 10 contains a summary of the main contributions and results of this thesis, a discussion of and comparison with related work, and a short sketch of some future work.

Reuse by Contract

The first part discusses the concept of reuse by contract and its realization within the NORA/HAMMR[1] retrieval system. Reuse by contract is an attempt to apply the design by contract software development principle [Mey92] to software reuse.

> "For reuse to be effective, Design by Contract is a requirement. Without a precise specification attached to each reusable component—precondition, postcondition, invariant—no one can trust a supposedly reusable component."
>
> J.-M. Jézéquel and B. Meyer, in [JM97]

Chapter 2 re-investigates the software component retrieval process in general and the additional problems introduced by the deduction-based approach in particular. Conceptually, the main problem is that the retrieval process is now based on an undecidable operation—specification matching—which in turn makes it more difficult to apply the "traditional" information retrieval notions of relevant and retrieved components to understand and assess systems. The approach followed here is to refine the usual notion of "retrieved components" into two separate concepts, matching components and found components, which allows to deal with the problems introduced by the undecidability of specification matching without compromising the notion of relevance.

Chapter 3 contains a detailed discussion of the structure and interpretation of *match predicates*, i.e., formally defined relations between component indexes and queries which determine which components should be retrieved in response to a given query. Match predicates play a central role in the concept of reuse by contract because they also determine the circumstances under and limits to which retrieved components can be reused—in effect, the match predicates are the most important part of the *reuse contract*. To quote again from Jézéquel and Meyer:

[1] NORA is no real acronym but an inference-based software engineering environment, HAMMR is the highly adaptive multi-method retrieval system within NORA.

> "*reuse without a contract is a sheer folly.* ... 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."
>
> J.-M. Jézéquel and B. Meyer, in [JM97]

Chapter 4 describes the experimental set-up in some more detail, in particular the NORA/HAMMR-system and the test library. It explores the requirements for a practical and usable deduction-based retrieval system and how they are reflected within NORA/HAMMR's architecture. It also contains a short description of the applied theorem provers.

Rejection Techniques

The second part discusses some techniques to reduce the load deduction-based software component retrieval puts on the ATPs. This is *the* cornerstone to make the concept practical. While load reduction can be achieved with different approaches, NORA/HAMMR only applies *number reduction* techniques, i.e., it tries to minimize the number of emerging proof tasks; due to the specific problem profile, the greatest effects are achieved if tasks associated with non-matches are ruled out. However, since subsequent confirmative steps cannot recover accidentally dropped matches, their number must be minimized.

Chapter 5 describes *recall-preserving* rejection filters which are based on simplification. The approach taken here is to use different term rewriting systems to simplify the proof tasks and to reject all such components where the associated proof task can be rewritten to *false*. This chapter contains a description of the different rewrite systems applied for simplification, and, in particular, of the specialized rules which have been developed to handle (freely) generated datatypes more efficiently.

If a proof task is not contradictory, however, the simplification-based techniques described in Chapter 5 are not strong enough and fail to produce a definitive result. Chapter 6 discusses rejection filters based on counterexamples. A counterexample is simply a specific structure, interpretation, and variable assignment under which the proof task evaluates to *false*. Counterexamples can in general be used in two different ways. If the number of all structures is finite, the proof task can be evaluated exhaustively over all potential counterexamples; if no actual counterexample is found, the task is valid and the component can safely be accepted (i.e., *model checking*). Unfortunately, this is not the case here—in the test library, the *list*-datatype induces infiniteness. In NORA/HAMMR, the proof task is thus checked against a small, fixed number of potential counterexamples; if no actual counterexample is found, nothing can be concluded.

The counterexamples are specified by additional axioms which do not hold in general, e.g., constraining the size of the *item*-domain. However, since these

additional axioms are not necessarily model conservative theory enrichments, the filters are no longer guaranteed to be recall-preserving. Moreover, the counterexample structures are still infinite such that the proof tasks cannot be simply evaluated; instead, the simplification-based techniques described in Chapter 5 and off-the-shelf ATPs are used to identify the actual counterexamples.

Specification Matching

The final third part contains a first substantial[2] experimental evaluation of the specification matching task. This evaluation is conceptually split into two parts.

Chapter 7 investigates the naive implementation of specification matching as proposed in the existing literature [MW95b, MW97b, MMM94, MMM97, CJ92, JC94, MM91], i.e., a simple generate-and-test approach. This chapter serves as base case for the more elaborate variants provided by NORA/HAMMR. It can also be used as a benchmark for the performance of off-the-shelf ATPs to a class of proof tasks which is typical for software engineering applications.

Chapters 8 and 9 investigate the effects of some proof task preprocessing steps. Such preprocessing steps become necessary because the proof tasks are generated completely automatically and thus exhibit completely different characteristics than the benchmarks the ATP community usually uses for testing and tuning of the provers, e.g., the TPTP-library [SSY94].

Usually, the proof tasks in ATP benchmarks cannot be simplified in any obvious way, e.g., by eliminating the propositional constants *true* and *false*. In deduction-based retrieval, however, the proof tasks can usually be simplified substantially, depending on the match predicate and the query. In principle, such redundancies should not make any difference because they can be eliminated easily, e.g., during clausification. In practice, however, many simplifications—even simple ones—do have effects on the performance of the ATPs. Chapter 8 investigates and quantifies some of these effects.

Similarly, the proof tasks in ATP benchmarks contain often a very finely tuned set of axioms and lemmas; sometimes, this set is even obtained by a manual post-mortem analysis of a successful proof. Obviously, this effort cannot be spent in deduction-based retrieval. However, the supply of axioms and lemmas is crucial—a single missed key lemma can make a proof much harder or even impossible. Unfortunately, the naive solution to include *all* available lemmas does not work because it induces too large search spaces. Chapter 9 describes a simple signature-based heuristic which selects only such axioms and lemmas which are necessary

[2]This evaluation is by no means claimed to be exhaustive, despite the by far more than 100.000 proof tasks which have been generated and checked. A really exhaustive evaluation would require more experimentation with different theorem provers, control strategies, and parameter settings as well as with libraries over other domains. However, I am confident that the numbers would not change dramatically and that the same conclusions could be drawn. Besides, there is only so much work a single person can do in the course of one dissertation.

to find a proof at all or are likely to shorten it and omits all those which only increase the search space.

Chapter 2

Software Component Retrieval

Software component retrieval is the application of information retrieval (IR) to software reuse. IR has developed a large body of concepts and notions which are useful not only for text-based approaches to component retrieval but also for the development, description, and analysis of specialized algorithms.

In the following section, I informally introduce the main IR notions and discuss how software component retrieval can be considered as a special instance of IR. I follow [vR79, SM83, Fuh95] for the general IR nomenclature and build on [MMM98] for the aspects which are more specific to component retrieval. In Section 2.2, I discuss the distinction between matching and retrieved components which is particular to deduction-based retrieval and some of its ramifications. Finally, in Section 2.3, I introduce and define a variety of measures for the evaluation of (component) retrieval systems.

2.1 Information Retrieval Terminology

Information retrieval is essentially the process of the content-based, goal-directed extraction of relevant text documents, or more general, *assets* from large collections. Its overall goal is to deliver an *exhaustive* sub-collection of all *single* relevant assets (i.e., independently and individually contributing to the user's goal). IR differs from the clearer defined *data* retrieval by mechanism, purpose, and result.

- Assets are not extracted by precise asset identifiers (e.g., unique keys) or attribute values but by (approximate) descriptions of their contents.
- Assets are in the first place not extracted for further (automatic) processing but to solve a user's goal.
- Retrieved assets do not necessarily solve the *intended* goal, even if their content descriptions perfectly fit the goal description.

In general, information retrieval is "fuzzier" than data retrieval. The main problem ("relevance problem") is that assets are retrieved by descriptions of their content and the goal but are judged as useful or *relevant* by different, more vague and subjective criteria. Hence, an asset which completely solves the goal for one user may be completely useless for another.

IR comprises three different tasks: navigation, matching, and grading. *Navigation* determines which assets are visited at all and in what order they are visited. *Brute-force navigation* (i.e., visiting all assets in the collection) is always possible. More elaborate navigation schemes require internally structured asset collections or additional indexes; sometimes the term *libraries* is reserved for such structured collections. *Matching* determines which assets possibly have relevant contents, *grading* determines in what order these assets are presented. Methods without the relevance check (i.e., navigation and presentation only) are also called *browsing*.

Methods without grading are usually called *binary retrieval*; there, the selected documents are presented in an arbitrary order, e.g., the order in which they have been visited. Methods with grading calculate a *status value* or *retrieval weight* for each document. The documents are ordered and presented by decreasing weight. Documents with the same weight form a *rank*; the ordering within a rank is arbitrary. Retrieval can then be defined via the ranks, either by selecting all documents up to a certain rank, independent of their total number (*rank-oriented*), or by selecting a certain number of the highest-ranking documents, independent of the lowest actual rank (*totaling*).

Information retrieval methods do not work with the actual assets but only with unique abstract representations called *surrogates*. The abstraction process from the assets to the surrogates is also known as *indexing* or *classification*. Properly, a surrogate is confined to "organizational" information only, (e.g., location, bytecode representation), but often the notion is stretched such that it may also contain an abstract representation of the asset's contents. This is called *key* even though it does in general not satisfy the usual key property in the database sense of the word, i.e., it does not necessarily identify the asset uniquely. Matching only works with the keys. A *library storage structure* is an ordering on these *(key, surrogate)*-pairs which may be used for navigation. It may also contain additional index entries for existing component surrogates or *redundant keys*. Usually, the actual surrogates are then only associated to the most specific keys in the navigation ordering; keys without directly associated surrogates are called *blind keys*.

Queries are similar to (blind) keys; they are abstract representations of the assets to be retrieved. The query language need not to be exactly the same as the key language but may be either a restriction or an extension. Binary retrieval using a query language which supports the usual set operations on intermediate results is also called *boolean retrieval*.

Since IR involves a great deal of subjectivity, system evaluation is very important. The most basic but still most common measures are recall and precision. The *precision* p of a query result denotes the fraction of relevant assets in the query result while the *recall* r measures the ratio between the number of retrieved relevant assets and the total number of relevant assets contained in the library. Sometimes, the *fallout* f is defined as a third measure. It is similar to the recall but uses the irrelevant assets instead of the relevant assets and thus indicates the ability of a system to reject irrelevant assets.

Of these three measures, only the precision is directly observable outside the system (i.e., by its users) because the other two are defined relative to the number of relevant assets in the *entire* library. However, this number is difficult to evaluate:

- Relevance need not be a binary property and some assets in a library may be more relevant than others.
- Relevance need not be a binary function of the query and component only but may depend on the particular library. Assets which must be considered relevant for a poor library can be ignored in better collections.
- Assets need not be relevant in the first place but may become relevant only after the inspection of other retrieval results ("reference chasing").
- For large libraries a relevance judgment for every single asset is not feasible.

In practice, the first three aspects are ignored and to cope with large libraries, statistical methods (e.g., relevance judgment only on test sets, query generalization) are applied.

In order to get reliable numbers for the quality of a system, the single observations must be averaged. The arithmetic means of the above measures can be considered *query-oriented* averages because all queries are weighted the same, regardless of their answer sets. Since query-oriented averages are corrupted by queries with empty answer sets, the *document-oriented* averages are sometimes used. They circumvent this problem as they sum up the respective numerators and denominators before the quotient is calculated (cf. Def. 2.3.4).

Software component retrieval is a special instance of information retrieval in that the assets are reusable software artifacts, or more specifically, in this context, functions and procedures (cf. Section 1.2.) The specialization to *reusable components* allows a specialization of the notion of relevance which is crucial for a understanding of component retrieval.

The *retrieval policy* describes on a general and abstract level how components must be related to the original user's goal (i.e., task to be solved) to be considered relevant. In *exact retrieval*, components are considered relevant only if the satisfy the user's goal exactly while *proper retrieval* also allows for more general components. Both policies are usually implemented in a binary retrieval framework

because they retrieve "perfect" matches which cannot be ordered by proximity to the goal but only by arbitrary external criteria. In *approximate retrieval*, a component is already relevant if it satisfies the user's goal partially. A partial solution can be defined semantically or syntactically:

- Semantically, a partial solution either provides the required result on a subset of the required domain, or it provides a useful intermediate result which requires further processing.

- Syntactically, a partial solution requires (a few) modifications.

Approximate retrieval corresponds more closely to the usual ranking case because it comes with a built-in notion of proximity. In the semantic case, proximity is the coincidence between required and provided domains, in the syntactic case it is the modification distance. In general, approximate retrieval does not subsume exact or proper retrieval, respectively, because perfect matches are not necessarily ranked highest. However, if this is the case, the approximation is also called *normalized*.

The retrieval policy is related to the reuse policy or reuse style. Clearly, exact and proper retrieval aim at black box reuse: "perfect" matches can be reused "as is". Under the approximate retrieval policy, however, the possible reuse policy is not determined so clearly. For a semantic view of partial solutions it is usually grey box reuse; this can be "lifted" to black box reuse by a software development process which tracks all open obligations stemming from partiality. However, for a syntactic view only white box reuse remains possible.

The *relevance condition* is a formalization of the retrieval policy. It is a logical expression parameterized over a component surrogate and the query whose formal validity entails the relevance of a component. However, it is not uniquely determined—for the same retrieval policy different relevance conditions are possible. The *retrieval goal* is a partial instantiation of the relevance condition with the query.[1]

2.2 Relevant vs. Matching vs. Found

Classical, text-based information retrieval only distinguishes between relevant (i.e., desirable) and retrieved assets. In software component retrieval, a more detailed view is more appropriate, particularly in the deduction-based case where retrieval is based on an undecidable operation. Here, a query and a component (more precisely, their keys q and c) can be related in three different ways:

[1] In [MMM98], Mili et al. do not distinguish between retrieval *policy* and *goal* but denote both as retrieval goal. Unfortunately, they also overload this notion with some aspects of the retrieval *mechanism* (i.e., the match condition).

- c is *relevant* for q, i.e., the *relevance judgment* or *relevance condition* $\rho(q, c)$ is satisfied.
- c *matches* q, i.e., c *should be* retrieved for q because a *match predicate* or *match condition* $\mu(q, c)$ is satisfied.
- c is actually found for q, i.e., *retrieved* by an algorithm φ. The match predicate μ can thus be considered an (abstract) specification for φ, or vice versa, φ as implementation of μ.

For a library $\mathcal{L}$, each of the predicates π induces a set $[\![q]\!]_\pi = \{c \in \mathcal{L} \mid \pi(q, c)\}$ of components which is used to characterize and evaluate retrieval systems. If the respective predicates ρ and φ are determined by the context, I also use the "traditional" IR notations $\mathrm{REL}(q) = [\![q]\!]_\rho$ and $\mathrm{RET}(q) = [\![q]\!]_\varphi$ to denote the sets of relevant and retrieved components, respectively.

Relevance judgments are "at will" (i.e., depend on arbitrary external criteria) and are thus beyond any further investigation. Match and retrieve predicates, however, possess some internal structure which warrants a closer look, especially at the exact relationship between both.

2.2.1 Match Predicates

Different match predicates have already been investigated by A. Moorman Zaremski [Moo96] who distinguished between *partial order* and *equivalence* matches, depending on the kind of order the predicate induces on the keys. Although reflexivity, transitivity, and symmetry are in fact the most interesting properties of match predicates, the restriction to the ordering on the *keys* does not adequately capture the essentials of a match predicate—the induced ordering on the match sets.

In the following, $\mathcal{K}$ denotes the universe of possible keys, i.e., all possible component specifications. Obviously, this universe must include the library, i.e., $\mathcal{L} \subset \mathcal{K}$ holds.

Reflexivity

Obviously, a match predicate cannot be completely arbitrary—it must at least be reflexive: if the query and the key of the component are *the same*, the match is obvious.

Transitivity

Both partial order and equivalence matches require transitivity of the match predicate. Transitivity entails the pleasant property that more general queries also have larger match sets.

Lemma 2.2.1 *If μ is transitive, then $\forall k, k' \in \mathcal{K} \cdot \mu(k, k') \Rightarrow [\![k']\!]_\mu \subseteq [\![k]\!]_\mu$.*
Proof: *Assume $c \in [\![k']\!]_\mu$; hence, $\mu(k', c)$. With transitivity, $\mu(k, k')$ then implies $\mu(k, c)$, thus also $c \in [\![k]\!]_\mu$.* $\triangle$

However, in practice, transitivity is a rather strict property which is not valid for some quite common match predicates, e.g., keyword-based retrieval with a disjunctive query interpretation—even if both q_1 and q_2 as well as q_2 and q_3 have a common keyword, q_1 and q_3 need not necessarily have one.[2] But since the monotonicity property of Lemma 2.2.1 *is* important in practice, a relaxed definition would be helpful. Such a definition should reflect the fact that the actual ordering on the keys is less important than the induced relation between the match sets.

Definition 2.2.2 (quasi-transitive) *A match predicate μ is* quasi-transitive *iff* $\forall k, k' \in \mathcal{K} \cdot \mu(k, k') \Rightarrow [\![k']\!]_\mu \subseteq [\![k]\!]_\mu$.

It is easy to see that any (reflexive and) transitive match predicate is also quasi-transitive. However, any quasi-transitive match predicate induces a (transitive) subset relation and, moreover, match predicate and subset relation coincide.

Lemma 2.2.3 *If μ is quasi-transitive then $\forall k, k' \in \mathcal{K} \cdot \mu(k, k') \Leftrightarrow [\![k']\!]_\mu \subseteq [\![k]\!]_\mu$.*
Proof: *"$\Rightarrow$" is the definition of quasi-transitivity. For "$\Leftarrow$" , $k' \in [\![k']\!]_\mu$ by reflexivity of μ which implies $k' \in [\![k]\!]_\mu$ by $[\![k']\!]_\mu \subseteq [\![k]\!]_\mu$ and thus $\mu(k, k')$ by definition of the match set $[\![\cdot]\!]_\mu$.* $\triangle$

Hence, the only way to achieve the desirable monotonicity property over the match sets is to have a transitive match predicate in the first place.

Symmetry

Partial order and equivalence matches differ only in symmetry properties: if the match relation is a partial order, it must be *anti-symmetric* or identitive, otherwise it must be *symmetric*. However, as with transitivity, anti-symmetry is a rather strict property and it holds even less often. A match relation cannot be a partial order match if it provides a certain degree of freedom and allows equivalent but different formulations of a query because it then is no longer anti-symmetric. Hence, a more relaxed definition based on the match sets is more appropriate.

Definition 2.2.4 (quasi-identitive) *A match predicate μ is* quasi-identitive *iff* $\forall k, k' \in \mathcal{K} \cdot \mu(k, k') \wedge \mu(k', k) \Rightarrow [\![k]\!]_\mu = [\![k']\!]_\mu$.

[2] More generally, no match predicate which is based on any kind of distance measure d between query and component (i.e., $\mu(q, c) \leftrightarrow d(q, c) \leq t, t \geq 1$) can be transitive. This is an immediate consequence of the triangle inequality for distances.

Since quasi-identitivity obviously follows from transitivity, reflexivity and transitivity are *the* important properties of a match relation. Hence, in algebraic terms, match relations need be only pre-orders instead of partial orders as claimed by A. Moorman Zaremski. However, for quasi-identitive match predicates, a partial order can be recovered if the keys are factored through the match sets. Similar observations also hold for the case of symmetry and equivalence matches.

2.2.2 Library Indexes

Match predicates can also be used to build *internal* indexes over component libraries, i.e., indexes which do not require an additional level of information but use the components in the library. The canonical way [Moo96, MMM97] to build such an internal library index comprises two steps:

1. Merge equivalent components into a single node.
2. Order the nodes by generality.

Both steps can be reduced to operations on the match sets: equivalent components have equal match sets, more general components have larger match sets. Hence, for an index the match set of each component in the library is calculated, or under an algebraic point of view, the library is factored through the match predicate.

Definition 2.2.5 (library index) *For a library $\mathcal{L}$, the* library index *induced by a predicate π is*

$$\mathcal{L}/\pi = \{[\![\, c \,]\!]_\pi \mid c \in \mathcal{L}\}$$

A library index is called normal *if $\langle \mathcal{L}/\pi, \supseteq \rangle$ is a homomorphic image of $\langle \mathcal{L}, \pi \rangle$ and* complete *if $\langle \mathcal{L}/\pi, \supseteq \rangle$ is a lattice.*

Any library index is a partial order with respect to the superset relation; however, even a match predicate which is not a partial order match may induce a normal index. The following corollary is a direct consequence of the above definition and Lemma 2.2.3.

Corollary 2.2.6 *If a match predicate μ is transitive then its induced library index is normal for any library $\mathcal{L}$.*

In fact, for transitive match predicates the algebraic relation between a library and its induced index is quite strong. It can be shown that there always exists a *retraction-section* pair between the library and its induced index. Recall that a retraction-section pair between two partially ordered sets $\mathcal{A} = \langle A, \leq \rangle$ and $\mathcal{B} = \langle B, \sqsubseteq \rangle$ is a pair $(r : \mathcal{A} \to \mathcal{B}, s : \mathcal{B} \to \mathcal{A})$ of homomorphisms such that $r(s(b)) = b$ (or, equivalently, $r \circ s = \mathrm{id}_{\mathcal{B}}$) and $s(r(a)) \leq a$ holds $\forall a \in A, b \in B$.

Theorem 2.2.7 *If μ is reflexive and transitive then there exists a retraction-section pair between $\langle \mathcal{L}, \mu \rangle$ and $\langle \mathcal{L}/\mu, \supseteq \rangle$.*
Proof: *Select an arbitrary representation function* $\mathrm{rep} : \mathcal{L}/\mu \to \mathcal{L}$ *such that* $[\![\mathrm{rep}(m)]\!]_\mu = m$*, i.e., for each index class select an arbitrary "maximal" representative. Since the index classes are always non-empty due to the reflexivity of μ, such a representative always exists.* rep *is a homomorphism because* $m_1 \supseteq m_2 \Rightarrow \mu(\mathrm{rep}(m_1), \mathrm{rep}(m_2))$ *follows with transitivity from Lemma 2.2.3,* $[\![\cdot]\!]_\mu$ *is also a homomorphism due to Lemma 2.2.3. Now,* $([\![\cdot]\!]_\mu, \mathrm{rep})$ *is the retraction-section pair:*

- $\forall m \in \mathcal{L}/\mu : [\![\mathrm{rep}(m)]\!]_\mu = m$ *holds by construction of* rep*,*
- $\forall c \in \mathcal{L} : \mu(\mathrm{rep}([\![c]\!]_\mu), c)$ *holds because it is by definition of quasi-transitivity equivalent to* $[\![\mathrm{rep}([\![c]\!]_\mu)]\!]_\mu \supseteq [\![c]\!]_\mu$ *which reduces by construction of* rep *to* $[\![c]\!]_\mu \supseteq [\![c]\!]_\mu$*.*

$\triangle$

Complete normal indexes are very useful for *browsing* because they already contain any "interesting" subset of the library. However, they cannot be obtained by mere restrictions on the match relation but require more elaborate preprocessing [Fis98, Fis00]; hence, they will not be investigated here.

2.2.3 Match Predicates and Retrieval Algorithms

Since match predicate and retrieval algorithm can be considered specification and implementation, the usual notions of soundness and completeness apply.

Definition 2.2.8 (soundness, completeness) *A retrieval algorithm φ is called* sound (complete) *with respect to a match predicate μ iff* $\forall k \in \mathcal{K} \cdot [\![k]\!]_\varphi \subseteq [\![k]\!]_\mu$ $(\forall k \in \mathcal{K} \cdot [\![k]\!]_\mu \subseteq [\![k]\!]_\varphi)$.

Sound and complete algorithms are the common case in standard information retrieval, but for (deduction-based) software component retrieval they are the very exception; they are only possible for decidable match predicates. Fortunately, soundness and completeness are mainly of theoretical interest. From a practical point of view, a different property is more important.

Definition 2.2.9 (closure under iterated retrieval) *φ is* closed under iterated retrieval *iff*

$$\forall q, c \in \mathcal{K} \cdot c \in [\![q]\!]_\varphi \Rightarrow [\![c]\!]_\varphi \subseteq [\![q]\!]_\varphi$$

Closure under iterated retrieval is a "law of no surprise": nothing new is retrieved if the keys of retrieved components are re-used as follow-up queries. As expected, it follows from soundness and completeness, at least for transitive match relations.

Lemma 2.2.10 *If μ is transitive and φ is sound and complete w.r.t. μ, then φ is also closed under iterated retrieval.*
Proof: *Soundness and completeness of φ give $[\![q]\!]_\varphi = [\![q]\!]_\mu$ for all q. Hence, quasi-transitivity becomes $\forall q, q' \in \mathcal{K} \cdot \mu(q, q') \Rightarrow [\![q']\!]_\varphi \subseteq [\![q]\!]_\varphi$ which is by the definition of the match set and by the above equivalent to Definition 2.2.9.* $\triangle$

Hence, signature matching (at least the original version defined by [Rit91]) is closed under iterated retrieval but soundness and completeness alone do not suffice: keyword-based retrieval with disjunctive query interpretation is not quasi-transitive and thus also not closed.

An important question is whether closed retrieval algorithms are at all possible for deduction-based retrieval. Fortunately, this is in fact the case. The important property of such algorithms is *query stability*. A retrieval algorithm is called query-stable if its results reflect the ordering on the queries.

Definition 2.2.11 (query stability) *φ is* query-stable *with respect to μ iff*

$$\forall k, k' \in \mathcal{K} \cdot \mu(k, k') \Rightarrow [\![k']\!]_\varphi \subseteq [\![k]\!]_\varphi$$

φ is strictly query-stable *with respect to μ iff*

$$\forall k, k' \in \mathcal{K} \cdot \mu(k, k') \wedge \mu(k', k) \Rightarrow [\![k']\!]_\varphi = [\![k]\!]_\varphi$$

Query stability is another "law of no surprise". It assures that a more general query actually *retrieves*—and not only *matches*—a larger set of components (cf. Lemma 2.2.1). Similarly, strict query stability assures that equivalent keys actually retrieve identical sets of components. Query stability and strict query stability are thus the equivalent of quasi-transitivity and quasi-identity, respectively, for retrieval algorithms.

Corollary 2.2.12 *If μ is transitive and φ is query-stable w.r.t. μ, then φ is also closed under iterated retrieval.*

This leads to the question of how query-stable algorithms look like. Obviously, query stability again follows from soundness and completeness, but this does not help very much. For an antisymmetric match predicate, soundness is already sufficient because then $[\![q]\!]_\varphi \subseteq [\![q]\!]_\mu$ but $| [\![q]\!]_\mu | \leq 1$, i.e., there are no other retrieved components whose keys can be reused for a follow-up query.

Corollary 2.2.13

1. *If μ is transitive and φ is sound and complete w.r.t. μ, then φ is query-stable and also strictly query-stable w.r.t. μ.*

2. *If μ is antisymmetric and φ is sound w.r.t. μ, then φ is also strictly query-stable w.r.t. μ.*

Fortunately, completeness is in general not necessary because it can (in some sense) be compiled into an approximation of a library index.

Algorithm 2.2.14 (closed retrieval algorithm) *For any retrieval algorithm φ a closed algorithm can be obtained by the following steps.*

1. *Calculate* $\mathcal{L}/\varphi^*$
2. RET $:= \emptyset$
   R $:= \mathcal{L}$
3. **while** $R \neq \emptyset$ **do**
   Select $c \in R$
   if $\varphi(q, c)$
   then RET $:=$ RET $\cup [\![c]\!]_{\varphi^*}$
   R $:= R \setminus [\![c]\!]_{\varphi^*}$
   else R $:= R \setminus \{c\}$
   fi
   od

The first step is an indexing step. As usual, φ^* denotes the reflexive-transitive closure of φ; hence $\mathcal{L}/\varphi^*$ (i.e., the index of $\mathcal{L}$ with respect to φ^*, cf. Definition 2.2.5) is a partial order by construction. Moreover, $\mathcal{L}/\varphi^*$ is a sound approximation of $\mathcal{L}/\mu$.

Lemma 2.2.15 *If μ is transitive and φ is sound w.r.t. μ then $[\![c]\!]_\varphi \subseteq [\![c]\!]_\mu$ holds for all $c \in \mathcal{K}$.*
Proof: *By soundness, $[\![c]\!]_\varphi \subseteq [\![c]\!]_\mu$; this implies $[\![c]\!]_{\varphi^*} \subseteq [\![c]\!]_{\mu^*}$ by definition of reflexive-transitive closure. Due to transitivity $[\![c]\!]_{\mu^*} = [\![c]\!]_\mu$ which then establishes the claim.* $\triangle$

Step 2 and 3 comprise the actual retrieval algorithm. Initially, nothing is retrieved and the entire library is unchecked (i.e., in R). The main loop of the algorithm then just selects an arbitrary, still unchecked component c and tries to establish the match using the base algorithm φ. If it succeeds, c's entire index class is retrieved from the indexed library and removed from the test set, otherwise only c is discarded.

Theorem 2.2.16 *If μ is transitive and φ is sound w.r.t. μ then the closed retrieval algorithm is sound w.r.t. μ and closed under iterated retrieval.*
Proof: *Soundness follows from Lemma 2.2.15, closure from using φ^* for the index and the fact that only c and not $[\![c]\!]_\mu$ is discarded if $\varphi(q, c)$ fails.* $\triangle$

2.2.4 Library Assumptions

In order to show more properties of match predicates and retrieval algorithms in general or in order to show some properties of some specific match predicates,

more assumptions about the components in the library (more precisely, their specifications) have to be made.

Definition 2.2.17 (implementability, non-triviality, determinism) *A specification s comprising the precondition* pre_s *and postcondition* post_s *is called*

- implementable *iff* $\forall \vec{x} \cdot pre_s(\vec{x}) \Rightarrow \exists \vec{y} \cdot post_s(\vec{x}, \vec{y})$
- non-trivial *iff* $\forall \vec{x} \cdot pre_s(\vec{x}) \Rightarrow \exists \vec{y} \cdot \neg post_s(\vec{x}, \vec{y})$
- deterministic *iff* $\forall \vec{x} \cdot pre_s(\vec{x}) \Rightarrow \exists_1 \vec{y} \cdot post_s(\vec{x}, \vec{y})$ [3]
- strictly deterministic *iff* $\forall \vec{x} \cdot \exists_1 \vec{y} \cdot post_s(\vec{x}, \vec{y})$

Sometimes it is necessary to make additional assumptions on the behavior of a component *outside* its proper domain. Minimality excludes accidentally valid return values; it corresponds to a conservative view of components where nothing is returned if the precondition is violated. Maximality reflects the interpretation that any result is valid if the precondition is violated; a specification becomes maximal if its original postcondition $post_s$ is replaced by $pre_s \Rightarrow post_s$.

Definition 2.2.18 (minimality, maximality) *A specification s is called*

- minimal *iff* $\forall \vec{x} \cdot \neg\, pre_s(\vec{x}) \Rightarrow \forall \vec{y} \cdot \neg post_s(\vec{x}, \vec{y})$
- maximal *iff* $\forall \vec{x} \cdot \neg\, pre_s(\vec{x}) \Rightarrow \forall \vec{y} \cdot post_s(\vec{x}, \vec{y})$

However, these specific assumptions about specifications are not justified in general. I call a library *reasonable* if all component indexes are implementable and minimal.

Similarly, it is necessary to make assumptions on the set of sensible queries which can be posed against a library; however, these assumptions also depend on the chosen relevance predicate ρ.

Definition 2.2.19 (admissible query) *A query $q \in Q$ is called* admissible *w.r.t. a library $\mathcal{L}$ and a relevance predicate ρ iff* $\mathrm{REL}(q) \subset \mathcal{L}$. *$Q$ is called admissible iff all $q \in Q$ are admissible.*

Unless stated explicitly otherwise, I assume libraries to be reasonable and queries to be admissible for the remainder of this thesis.

[3] The *unique existential quantifier* $\exists_1$ is defined as usual as an abbreviation: $\exists_1 x \cdot p(x) \equiv \exists x \cdot (p(x) \wedge \forall y \cdot (p(y) \Rightarrow x = y))$.

2.3 System Evaluation

In the following, I define the usual measures for the evaluation of binary retrieval systems; some of the measures are tailored towards filtering. All measures assume an objective, user-independent, and binary relevance predicate. For specification matching, the relevance predicate is often identified with the match predicate. I only define measures w.r.t. a single query; I denote the respective arithmetic means (w.r.t. a set Q of queries) by overbars. If Q is "sufficiently large" and immaterial, I usually drop it.

The first step towards a system evaluation is to identify the possible extreme system behaviors, e.g., retrieving nothing or the entire library. This gives rise to the following three—hypothetical—retrieval algorithms.

Definition 2.3.1 (ideal, incompetent, and indifferent algorithm) *A retrieval algorithm is called*

- ideal *iff* $\mathrm{RET}(q) = \mathrm{REL}(q)$,
- incompetent *iff* $\mathrm{RET}(q) = \emptyset$, *and*
- indifferent *iff* $\mathrm{RET}(q) = \mathcal{L}$

for every query q.

In practice, a retrieval system can only be evaluated over a test library but the quality of the test library obviously influences the perceived quality of the retrieval system. To account for this, and to judge the quality of a library, the number of relevant components in relation to the size of the library or the *expected relevance* of a component can be used.

Definition 2.3.2 (expected relevance) *The* expected relevance *of an arbitrary component for a query q is*

$$\gamma(q) = \frac{|\,\mathrm{REL}(q)\,|}{|\,\mathcal{L}\,|}$$

The expected relevance is also the probability of picking a relevant component at random, or, alternatively, the precision of the entire library. $\overline{\gamma}$ is also called the *general retrieval factor* for a given library because $\overline{\gamma} \cdot |\,\overline{\mathrm{RET}}_{\varphi}\,|$ denotes the average number of relevant components which can be expected for a query.

Recall and precision of a query are defined independent of the expected relevance as the number of retrieved relevant components in relation to the number of total retrieved and relevant components, respectively.

Definition 2.3.3 (recall, precision) *For a query $q \in \mathcal{K}$, the* recall $r(q)$ *and* precision $p(q)$ *of the answer set* $\mathrm{RET}(q)$ *are defined as follows:*

$$r(q) = \frac{|\,\mathrm{RET}(q) \cap \mathrm{REL}(q)\,|}{|\,\mathrm{REL}(q)\,|}$$

$$p(q) = \frac{|\,\mathrm{RET}(q) \cap \mathrm{REL}(q)\,|}{|\,\mathrm{RET}(q)\,|}$$

Recall and precision are not defined if no components are relevant and retrieved, respectively, but both definitions can be extended consistently:

- If $\mathrm{REL}(q) = \emptyset$ then all relevant components can also be considered as retrieved—hence, $r(q) = 1$. This is consistent with the assumption that $\mathrm{RET}(q) \supseteq \mathrm{REL}(q)$ should always imply $r(q) = 1$.
- If $\mathrm{RET}(q) = \emptyset$ then all retrieved components can also be considered as relevant—hence, $p(q) = 1$. This is consistent with the assumption that $\mathrm{RET}(q) \subseteq \mathrm{REL}(q)$ should always imply $p(q) = 1$.

Although the extended definition of precision is especially justified in the deduction-based case, both definition extensions can affect the query-oriented averages $\overline{r}$ and $\overline{p}$. Alternatively, the *document-oriented* averages can be defined.

Definition 2.3.4 (document-oriented averages) *For a set of queries $Q \subset \mathcal{K}$, the* document-oriented averages *of recall and precision are defined as follows:*

$$\underline{r}(Q) = \frac{\sum_{q \in Q} |\,\mathrm{RET}(q) \cap \mathrm{REL}(q)\,|}{\sum_{q \in Q} |\,\mathrm{REL}(q)\,|}$$

$$\underline{p}(Q) = \frac{\sum_{q \in Q} |\,\mathrm{RET}(q) \cap \mathrm{REL}(q)\,|}{\sum_{q \in Q} |\,\mathrm{RET}(q)\,|}$$

However, query-oriented and document-oriented averages can differ quite substantially, even if the relevant and retrieved sets, respectively, are not completely empty but only unevenly large. Consider for example $|\,q\,| = 10$, where 9 queries have a single relevant component which is missed and the last query has 9 relevant components which are all retrieved. Then $\overline{r}(Q) = \frac{9\times 0.00 + 1\times 1.00}{10} = 10\%$ but $\underline{r}(Q) = \frac{9\times 0 + 1\times 9}{18} = 50\%$. Hence, while the query-oriented average reflects the user's view of a retrieval system, the document-oriented average reflects the system provider's view: in total, 50% of the relevant components have been delivered.

In an indifferent system (i.e., $\mathrm{RET}(q) = \mathcal{L}$), the definitions of precision and expected relevance coincide. If such a retrieval system is used as a filter, this means that the precision of its output is the same as that of its input, or that it did not increase the precision. This observation motivates the following definition.

Definition 2.3.5 (precision leverage) *The* precision leverage *of a retrieval algorithm φ for a query q is*

$$\delta_p(q) = \frac{p(q)}{\gamma(q)}$$

The precision leverage is a quality measure for retrieval algorithms and filters which takes the quality of the library properly into account; a filter can be considered to be *useful for* q iff $\delta_p(q) > 1$ because it performs better than just picking the same number of components randomly. Obviously, $\delta_p(q)$ is close to 1 for large values of $\gamma(q)$, even in the ideal case. This confirms the experience that very focussed (or small) libraries do not really require automatic retrieval support. The maximal precision leverage is $|\mathcal{L}|$; it occurs only for a single relevant component and is achieved only by an ideal algorithm. In contrast to recall and precision, the definition of the precision leverage cannot be extended consistently for $\mathrm{REL}(q) = \emptyset$.

Loss and *junk* (also called *noise*) are the complementary measures to recall and precision: the loss denotes the relative number of missed matches (i.e., non-retrieved relevant components) while the junk is the relative number of mismatches (i.e., retrieved irrelevant components.)

Definition 2.3.6 (loss, junk) *For a query $q \in \mathcal{K}$, the* loss $l(q)$ *and* junk $j(q)$ *and of the answer set* $\mathrm{RET}(q)$ *are defined as follows:*

$$l(q) = \frac{|\mathrm{REL}(q) \setminus \mathrm{RET}(q)|}{|\mathrm{REL}(q)|} = 1 - r(q)$$

$$j(q) = \frac{|\mathrm{RET}(q) \setminus \mathrm{REL}(q)|}{|\mathrm{RET}(q)|} = 1 - p(q)$$

Recall and precision (and thus also loss and junk) are *user-oriented* measures. They make statements only about the quality of an answer set w.r.t. the query but without respect to the entire library. Hence, they are not suitable for evaluating the *filtering effects* of an algorithm. Here, the *system-oriented* measures fallout and error quota can be used. The *fallout* is defined as the relative number of non-rejected irrelevant components w.r.t. all irrelevant components, while the *error quota* denotes the relative number of rejected relevant components w.r.t. all rejected components.

Definition 2.3.7 (fallout, error quota) *For a query q, the* fallout $f(q)$ *and* error quota $e(q)$ *of the answer set* $\mathrm{RET}(q)$ *are defined as follows:*

$$f(q) = \frac{|\mathrm{RET}(q) \setminus \mathrm{REL}(q)|}{|\mathcal{L} \setminus \mathrm{REL}(q)|}$$

$$e(q) = \frac{|\mathrm{REL}(q) \setminus \mathrm{RET}(q)|}{|\mathcal{L} \setminus \mathrm{RET}(q)|}$$

Hence, the fallout measures the rejective power of an algorithm (i.e., its ability to filter out irrelevant components) while the error quota denotes the precision of this filtering process. Obviously, fallout and error quota are related to junk and loss, respectively, as $j(q) = 0 \Leftrightarrow f(q) = 0$ and $l(q) = 0 \Leftrightarrow e(q) = 0$, but they are not correlated and a small error quota does not necessarily entail a small loss. In general, since the library usually contains mostly irrelevant documents for any specific query and since usually only a small fraction of the documents is retrieved (i.e., $|\mathcal{L}| \gg |\operatorname{RET}(q)|, |\operatorname{REL}(q)|$), fallout and error quota are smaller and much more evenly distributed than junk and loss.

The error quota can again be considered in relation to the precision of the input. This yields the *relative defect ratio*.

Definition 2.3.8 (relative defect ratio) *The* relative defect ratio *of a retrieval algorithm φ for a query q is*

$$\delta_e(q) = \frac{e(q)}{\gamma(q)}$$

φ is called a *zero-defect algorithm* iff $\delta_e(q) = 0$ and *defective* iff $\delta_e(q) > 1$ which implies that its ability to reject only irrelevant components is even worse than a purely random choice. Precision leverage and relative defect ratio are flip sides of the same coin. However, they are not correlated and a larger precision leverage is consistent with a larger defect ratio.

The notions of precision leverage and relative defect ratio (and the derived notions as useful or zero-defect algorithms) can be used nicely to characterize retrieval algorithms. For example, for any ideal algorithm φ_{ideal} (cf. Definition 2.3.1) the following theorem holds.

Theorem 2.3.9 *φ_{ideal} is a useful, zero-defect retrieval algorithm.*
Proof: *For φ_{ideal}, $\operatorname{RET}(q) = \operatorname{REL}(q)$ holds for each $q \in Q$ which implies $p(q) = 1$ and hence*

$$\begin{aligned}\delta_p(q) &= \frac{p(q)}{\gamma(q)} \\ &= \frac{1}{\frac{|\operatorname{REL}(q)|}{|\mathcal{L}|}} \\ &= \frac{|\mathcal{L}|}{|\operatorname{REL}(q)|}\end{aligned}$$

which is strictly greater than 1 since $\operatorname{REL}(q) \subset \mathcal{L}$ holds for all admissible queries q. Hence, φ_{ideal} is useful. Moreover, $\operatorname{RET}(q) = \operatorname{REL}(q)$ also immediately implies $e(q) = 0$ and thus $\delta_e(q) = 0$. $\triangle$

Similar results can be achieved for the other hypothetical algorithms. Moreover, it is possible to show that more general results hold, for example that a useful algorithm is never defective (cf. Theorem 2.3.11).

Theorem 2.3.10 *Every non-indifferent zero-defect retrieval algorithm φ is useful.*
Proof: *Let q be an arbitrary query. Since φ is a zero-defect algorithm, $\delta_e(q) = 0$ holds which implies $e(q) = 0$ and thus $\mathrm{REL}(q) \subseteq \mathrm{RET}(q)$. Hence,*

$$p(q) = \frac{|\,\mathrm{REL}(q)|}{|\,\mathrm{RET}(q)|}$$

and thus

$$\delta_e(q) = \frac{|\,\mathrm{REL}(q)|}{|\,\mathrm{RET}(q)|} \cdot \frac{|\mathcal{L}|}{|\,\mathrm{REL}(q)|}$$

which is strictly greater than 1 iff $\mathrm{RET}(q) \neq \mathcal{L}$, i.e., φ is not indifferent. $\triangle$

Theorem 2.3.11 *Every useful retrieval algorithm φ is non-indifferent and non-defective.*
Proof: *Assume φ that is indifferent, i.e., $\mathrm{RET}(q) = \mathcal{L}$. Since φ is useful,*

$$\begin{aligned} & \delta_p(q) > 1 \\ \Leftrightarrow\ & \frac{|\,\mathrm{RET}(q) \cap \mathrm{REL}(q)|}{|\,\mathrm{RET}(q)|} > \frac{|\,\mathrm{REL}(q)|}{|\mathcal{L}|} \\ \Leftrightarrow\ & \frac{|\,\mathrm{REL}(q)|}{|\mathcal{L}|} > 1 \end{aligned}$$

which is an immediate contradiction. Now assume φ that is defective, i.e., $\delta_e(q) > 1$ for a q. Then

$$\frac{|\,\mathrm{REL}(q) \setminus \mathrm{RET}(q)|}{|\mathcal{L} \setminus \mathrm{RET}(q)|} > \frac{|\,\mathrm{REL}(q)|}{|\mathcal{L}|} \qquad (*)$$

must hold. Since $\mathrm{REL}(q) \setminus \mathrm{RET}(q) = \mathrm{REL}(q) \setminus (\mathrm{RET}(q) \cap \mathrm{REL}(q))$ and $\mathrm{RET}(q) \subset \mathcal{L}$, (*) becomes

$$\begin{aligned} & \frac{|\,\mathrm{REL}(q) \setminus (\mathrm{RET}(q) \cap \mathrm{REL}(q))|}{|\mathcal{L} \setminus \mathrm{RET}(q)|} > \frac{|\,\mathrm{REL}(q)|}{|\mathcal{L}|} \\ \Leftrightarrow\ & \frac{|\,\mathrm{REL}(q)| - |\,\mathrm{RET}(q) \cap \mathrm{REL}(q)|}{|\mathcal{L}| - |\,\mathrm{RET}(q)|} > \frac{|\,\mathrm{REL}(q)|}{|\mathcal{L}|} \\ \Leftrightarrow\ & |\mathcal{L}| \cdot |\,\mathrm{REL}(q)| - |\mathcal{L}| \cdot |\,\mathrm{RET}(q) \cap \mathrm{REL}(q)\,| \\ & \quad > |\mathcal{L}| \cdot |\,\mathrm{REL}(q)| - |\,\mathrm{REL}(q)| \cdot |\,\mathrm{RET}(q)\,| \\ \Leftrightarrow\ & |\mathcal{L}| \cdot |\,\mathrm{RET}(q) \cap \mathrm{REL}(q)\,| < |\,\mathrm{REL}(q)| \cdot |\,\mathrm{RET}(q)\,| \qquad (**) \end{aligned}$$

On the other hand, since φ is useful,

$$\begin{aligned} & \delta_p(q) > 1 \\ \Leftrightarrow\ & \frac{|\,\mathrm{RET}(q) \cap \mathrm{REL}(q)|}{|\,\mathrm{RET}(q)|} > \frac{|\,\mathrm{REL}(q)|}{|\mathcal{L}|} \\ \Leftrightarrow\ & |\,\mathcal{L}\,| \cdot |\,\mathrm{RET}(q) \cap \mathrm{REL}(q)\,| > |\,\mathrm{REL}(q)\,| \cdot |\,\mathrm{RET}(q)\,| \end{aligned}$$

which is a contradiction to (**). Hence, φ cannot be defective. $\triangle$

Limits of Recall/Precision-Based Evaluation

Although the evaluation of retrieval systems based on recall and precision is the most common approach, it has been criticized in the literature quite often (cf. [Fuh95]). The major drawbacks cited are:

- Recall and precision work on ordinal scales only, that is, two measures cannot be compared quantitatively but only qualitatively.
- Recall and precision are no utility measures, that is, they are not suitable for indicating the usefulness of a retrieval system directly.
- Recall and precision are unsuitable for ranking retrieval systems. Unfortunately, this is to some extent also true for the derived measures precision leverage and relative defect ratio.

The first two drawbacks can be mitigated partially by comparison to the three hypothetical retrieval algorithms (cf. Def. 2.3.1). In an ideal retrieval system, retrieved and relevant components coincide, and, hence, $p_{\mathrm{ideal}} = r_{\mathrm{ideal}} = 1$, and $f_{\mathrm{ideal}} = 0$. In practice, however, precision and recall are antagonistic. In an indifferent system which always returns the *entire* library $\mathcal{L}$, $p_{\mathrm{indiff}} \to 0$ (for $|\,\mathcal{L}\,| \to \infty$), $r_{\mathrm{indiff}} = 1$ but also $f_{\mathrm{indiff}} = 1$—that is, the system is not only unable to narrow it's response but in addition also dumps a great share of the irrelevant assets contained in the library on the users. In an incompetent system, nothing is ever retrieved and thus $p_{\mathrm{incom}} = 1, f_{\mathrm{incom}} = 0$ as the ideal system but $r_{\mathrm{incom}} = 0$.

The last drawback, however, is system-immanent because recall and precision measure independent, or more precisely, antagonistic dimensions. As long as one system has both better recall *and* precision values as another system, everything is fine but if the values are skewed (e.g., better recall at the expense of precision), nothing decisive can be concluded. Unfortunately, even the usual workaround to consider both systems at a common recall (or precision) level does not work completely in the deduction-based case, because it does not assign ranks and, hence, does not allow to "trim" the responses to a common recall level.

Chapter 3

Contracts, Retrieval, and Reuse

One of the characteristics of deduction-based retrieval is a formally defined relation between indexes and queries (cf. page 9). In fact, this formal definition or *match predicate* is central to the entire concept of reuse by contract because it determines which components can be retrieved and, ultimately, how they may be reused. But as there are different possible ways to reuse components based on their contracts, there are also different possible ways to define matching formally.

Early research [RW91, KRT87, MM91] did not realize this choice and worked with a single match predicate, although the applied definitions are slightly different. A. Moorman Zaremski and J. Wing [MW95b, Moo96, MW97b] appear to be the first who have systematically investigated different match predicates. They defined two generic forms called *generic pre/post match* [Moo96, p. 41] and *generic specification match* [Moo96, p. 42] and then derived several variants by syntactic modifications of these generic forms. However, their investigation is problematic in two respects. From the software engineering perspective, the reuse effects of the different variants are not always presented clearly. Some of the definitions, e.g., *specialized match* [Moo96, p. 49] look contrived while others, e.g., *plug-in match* [Moo96, p. 44] or *generalized match* [Moo96, p. 48] are slight modifications of a more intuitive but missing definition. From the logical perspective, the handling of the formal parameters and their types is insufficient. The parameters are without distinction all universally quantified and the types are just identified but not properly matched onto each other.

Both problems are a direct consequence of the syntactic approach taken by Moorman Zaremski and Wing. The generic forms offer only a limited degree of freedom but this is exhausted systematically. Semantic approaches [PBA95, Pen98, FSS98, MMM97] are not subject to such problems and generally derive more intuitive match predicates.

Mistaking matching for relevance is another common fallacy in deduction-based retrieval. Work of A. Mili, R. Mittermeir et al. [BMM92, MMM97, JD+97, MMM98] shed some light on that issue but as the formal discussions in the preceding chapter show, it is more complicated than it appears superficially. I

follow the approach begun in our own earlier work [FKS95a, FKS95b, FKS95c] and *deliberately* use match predicates also to capture relevance. However, this does not imply that always the *same* predicate is used for both purposes.

In this chapter, I thus analyze how the different match predicates are built and how their choice affects the reuse styles and vice versa, taking into account—and avoiding—the aforementioned problems of the approach of Moorman Zaremski and Wing.

3.1 General Structure of Match Predicates

For their investigations, Moorman Zaremski and Wing have reduced the match predicates to their propositional structure, using, e.g., Q_{post} as abbreviation for the postcondition predicate $Q_{post}(\vec{x}, \vec{y})$ over parameters $\vec{x}$ and return variables $\vec{y}$. Although this convention allows a concise notation which emphasizes the structural similarities and differences between the different definitions, it is also deceptive as the scope and kind of the quantifiers are left implicit.[1]

A detailed analysis requires a more explicit notation. In the following, I use

- q and c to denote the entire query and component or their contracts, respectively, or to index parts of the contracts,
- *pre* and *post* to denote the respective pre- and postconditions,
- $\vec{x}$ to denote the formal arguments of a contract (i.e., parameters),
- $\vec{y}$ to denote the formal results of a contract (i.e., return variables),
- T to denote a *type compatibility predicate*,
- r and u as (secondary) indexes to denote the restricted and unrestricted or universal formal arguments, respectively.

Most of these notational conventions are straightforward. The last two are required for a more general handling of the formal parameters and their types; their precise nature will be made clear subsequently.

In general, all match predicates exhibit some structural similarities even if they are not derived syntactically because they are built up from only a few constituents. On a very abstract level,

$$Q\,\vec{x} \circ \vec{y} \cdot T[\vec{x}, \vec{y}] \dagger \mathcal{F}[pre_q, pre_c, post_q, post_c] \tag{3.1}$$

can be considered as *the* general structure of match predicates. Here,

[1] Cf. also a similar remark by J. Penix: "it is helpful to keep in mind that the missing variable arguments are all universally quantified" [Pen98, p. 22].

- $Q\,\vec{x} \circ \vec{y}$ is the *prefix* of the predicate where Q is an arbitrary sequence of universal and existential quantifiers and the formal parameters and results $\vec{x} \circ \vec{y}$ of q and c may be rearranged,
- $T[\ldots]$ is the type compatibility predicate of the match predicate,
- $\dagger$ is the *relativation connective* which may be either conjunction or implication, depending on the prefix, and
- $\mathcal{F}[\ldots]$ is the *body* of the predicate.

Moorman Zaremski and Wing's investigations fixed a certain prefix, type compatibility predicate, and relativation connective and focussed on the different bodies only.

3.1.1 Type Compatibility Predicates

The purpose of type compatibility predicates is to provide "glue" between the possibly different domains of the query and the candidate component. In the simplest case this is just a renaming of objects (i.e., isomorphism) but the general case is more complicated and resembles the data reification process in model-oriented specification (cf. [Jon90, Ch. 8]). In principle, all match definitions are thus schemas which are parameterized by the type compatibility predicates.

However, gluing can be considered from two different points of view:

- Under the *conceptual abstraction view* (cf. Figure 3.1), the domain of the query is not important in itself but only as an abstraction of relevant concepts of the component's domain because the client will adapt itself to the component's domain. Hence, the retrieved components will only be supplied with appropriate arguments.
- Under the *integration view* (cf. Figure 3.2), the domain of the query is the "real" domain of the client and not only a conceptual abstraction. Hence, the glue is required to convert actual values between the two domains.

The canonical examples to show the difference between the views are enumeration types, e.g., the colors of a traffic light. If the colors are only considered as abstractions, only their number and ordering are relevant and not their actual names. Hence, any other type with at least three different values will do. However, if the client already works on the actual type and only looks for additional functionality, then the type compatibility predicate must translate between the colors and their representations in the component (e.g., integers) to allow a smooth integration without the need for any manual modification.

These two different views can be visualized by two different commuting diagrams. For a conceptual abstraction, type compatibility can in principle be

considered as a function T which "lifts" the component's contract c into the query q for the purpose of finding a proof. Hence, as Figure 3.1 shows the type compatibility predicate as two arrows going from the component's domain and codomain to the query's domain and codomain, respectively. After a proof has

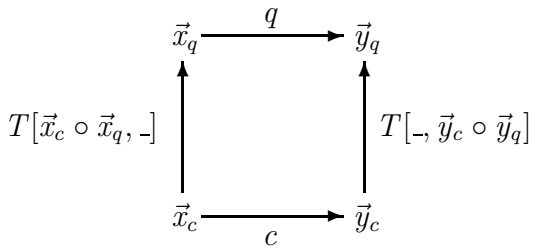


Figure 3.1: Type compatibility as conceptual abstraction

been found, T is not actually required for reusing the retrieved components. Hence, it is internal to the proof, or to put it into different words, need not be constructed explicitly.

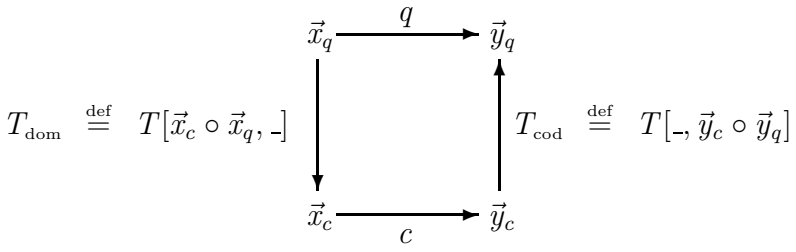


Figure 3.2: Type compatibility as integration

In contrast to this, for a proper integration the arrows must run in opposite directions. The type compatibility predicate can thus conceptually be split into two different parts T_{dom} and T_{cod} (cf. Figure 3.2.) Both parts are actually required to integrate any retrieved components without manual modification into a client program: T_{dom} translates the query domain into the component domain, T_{cod} translates the results back.

However, this leaves the question of how the type compatibility predicates can be obtained systematically. In data reification, the retrieve functions must be specified explicitly for each reification step but this is obviously not feasible in deduction-based retrieval. Hence, if the type compatibility is not trivial, then it must be

- constructed by the prover at proof time, or
- approximated by signature matching.

However, the first variant makes the match predicate a higher-order proof problem which in practice leaves approximation by signature matching the only feasible approach.

This approximation need neither be an isomorphism nor even a function between the respective domains and codomains, but must at least satisfy a weaker *adequacy condition*.

Definition 3.1.1 (adequacy condition) *A type compatibility predicate* $T \subseteq A \times B$ *is* adequate *if it can be represented by two relations* T_{dom} *and* T_{cod} *such that*

1. $T = T_{\text{dom}} \cup T_{\text{cod}}$
2. T_{dom} *it is left-total, i.e.,* $\forall x : A \cdot \exists y : B \cdot T_{\text{dom}}(x, y)$
3. T_{cod} *it is right-total, i.e.,* $\forall y : B \cdot \exists x : A \cdot T_{\text{cod}}(x, y)$
4. $T_{\text{cod}} \circ T_{\text{dom}} = id_A$

T *is* strictly adequate *if* T_{cod} *is also left-total.* T *is* functional *if both* T_{dom} *and* T_{cod} *are functions.*

In many cases, type compatibility predicates establish an isomorphism between the respective datatypes and are thus both strictly adequate and functional, e.g., $\forall(x_1, x_2) : A, (x_2, x_1) : B \cdot T((x_1, x_2), (x_2, x_1))$ which switches the order within tuples. However, even if T is only adequate, it can be used to translate between two datatypes: by left totality each element can be embedded, by right totality each embedded element can be translated back and by property 3.1.1.4 this process does not drop any information. If T is functional, T_{dom} is called the *canonical injection function* and T_{cod} the *retrieve function*.

The probably best-known example for a non-functional (i.e., truly relational) but still strictly adequate type compatibility predicate is the encoding of Boolean values by integers as in the C programming language. There, zero uniquely represents the *false*-constant but any non-zero integer can represent *true*, although it is customary to use a functional encoding $true \mapsto 1$ for this purpose. Hence, $T_{\text{dom}}(true, x) \Leftrightarrow x = 1$ but $\forall x \geq 1 \cdot T_{\text{cod}}(true, x)$.

If a type compatibility predicate T is used to integrate retrieved components, it can be split into the two relations $(T_{\text{dom}}, T_{\text{cod}})$ by projection to the domain and codomain, respectively, as shown in Figure 3.2. Since T_{dom} is only used to translate the arguments into the form expected by the component, it need not be right-total. Similarly, since T_{cod} is only used to translate each result back into the form expected by the query, it need not be left-total. However, if it is not even right-total on the codomain, integration may still be possible, but it must then additionally be shown that the actual return values are mapped correctly. The codomain of T_{dom} need not cover all formal parameters of the component. Those

which are covered are called *restricted* by T, the others are called *universal* or *unrestricted*. Consider for example a query with a single parameter $l : list$ and a component with two parameters $(i : item, l : list)$. Obviously, T should identify the two lists and leave the item i unrestricted. For a complete covering of the entire domain of the component, T could be extended by a relational join with the type of the unrestricted variables but here I work only with the minimal covering because this allows more control over the reuse effects.

More specific properties which relate the "static" domains with the actual preconditions, e.g.,

$$\forall \vec{x}_q \forall \vec{x}_{c,r} \forall \vec{x}_{c,u} \cdot T_{\mathrm{dom}}(\vec{x}_q, \vec{x}_{c,r}) \Rightarrow (pre_q(\vec{x}_q) \Leftrightarrow pre_c(\vec{x}_{c,u} \circ \vec{x}_{c,r}))$$

cannot reasonably be expected to hold by a general approximation, because they depend on the particular component and query.

The integrative view can also be used to define a pointwise "equality" of (deterministic) specifications modulo a type compatibility predicate.

Definition 3.1.2 (implementation modulo type compatibility) *A deterministic component c implements a deterministic query q pointwise modulo the type compatibility predicate T if*

$$\begin{array}{l} \forall \vec{x}_q \forall \vec{x}_{c,r} \forall \vec{x}_{c,u} \cdot \\ \quad T_{\mathrm{dom}}(\vec{x}_q, \vec{x}_{c,r}) \wedge pre_q(\vec{x}_q) \wedge pre_c(\vec{x}_{c,u} \circ \vec{x}_{c,r}) \Rightarrow T_{\mathrm{cod}}(q(\vec{x}_q), c(\vec{x}_{c,u} \circ \vec{x}_{c,r})) \end{array}$$

The relation to the usual pointwise equality of functions becomes clearer for total specifications without unrestricted arguments and functional type compatibilities. In that case, the condition simplifies to $\forall \vec{x}_q \cdot q(\vec{x}_q) = T_{\mathrm{cod}}(c(T_{\mathrm{dom}}(\vec{x}_q)))$.

Type compatibility predicates can be considered as a generalization of monadic sort predicates which are used to relativize restricted quantifiers. Following the translation

$$\begin{array}{rcl} \varphi(\forall x : R \cdot P[x]) & = & \forall x \cdot R(x) \Rightarrow \varphi(P[x]) \\ \varphi(\exists x : R \cdot P[x]) & = & \exists x \cdot R(x) \wedge \varphi(P[x]) \end{array}$$

from a sorted logic into an unsorted logic by [Obe62], (3.1) could also be interpreted as

$$Q\, \vec{x} \circ \vec{y} : T \cdot \mathcal{F}[pre_q, pre_c, post_q, post_c]$$

if φ is extended in the obvious way such that the relativation connective becomes an implication only if Q is a purely universal string and a conjunction otherwise. This reflects the intended interpretation that the "legal" instances of $\vec{x} \circ \vec{y}$ are restricted to the set T which in turn has the characteristic predicate $T[\vec{x}, \vec{y}]$.

3.1.2 Multiple Type Compatibility Predicates

In order to increase the recall, it is sometimes advisable to use more than one type compatibility predicate. For example, since the order of component parameters is somewhat arbitrary, parameters of the same type can be identified using any possible permutation; each of these permutations gives then rise to a type compatibility predicate. The intuitive interpretation of multiple type compatibility predicates is obviously that the component should already be retrieved if a proof is found for one of the predicates.

Unfortunately multiple type compatibility predicates cannot be represented as a disjunction over the respective variants and, hence, cannot be nested inside the proof task:

- For a universally quantified component parameter, a disjuncted predicate would occur in the premise of an implication; however, then the task's body must hold for *any* instead of only an *arbitrary* variant which obviously defeats the original purpose to increase the recall.

- For an existentially quantified component parameter, a disjuncted predicate would occur in the conclusion of an implication; however, then the proof may rely on actually using different variants of the type compatibility predicate which defeats their original purpose as glue.

Consequently, multiple type compatibility predicates lead in general to multiple proof tasks and may thus congest the retrieval system; hence, they should be handled with caution.

3.1.3 Universal Prefixes

In the existing literature, all top-level variables (i.e., formal parameters and return variables of the query and component, respectively) are universally quantified. This need not necessarily be the case. Other prefix variants are also useful to support different reuse styles, although at least the formal parameters of the query must be universally quantified: systematic reuse is possible only if the retrieved components satisfy the chosen match relation for *any* possible value in the query domain.[2] If existential quantifiers were used, accidental matches might spoil the precision and thus the reuse effect.

Universal prefixes follow the standard practice and use universal quantifiers for the query variables and for the restricted formal arguments and the results of the candidate. Consequently, the relativation connective must always be an implication. The variants differ, however, in the prefix position and quantifier of

[2]Remember that preconditions can be used to constrain that domain systematically.

the unrestricted variables.[3] In the following, I refer to the standard variant as the *fully universal form* or *fully universal prefix*, respectively.

Definition 3.1.3 (fully universal form) *The* fully universal form *of a match predicate has the general structure*

$$\forall \vec{x}_q \forall \vec{y}_q \forall \vec{x}_{c,u} \forall \vec{x}_{c,r} \forall \vec{y}_c \cdot T_{\mathrm{dom}}(\vec{x}_q, \vec{x}_{c,r}) \wedge T_{\mathrm{cod}}(\vec{y}_q, \vec{y}_c) \Rightarrow \mathcal{F}[pre_q, pre_c, post_q, post_c]$$

for any body $\mathcal{F}$*.*

Generally, if a universal prefix is used, the body must hold for every possible mapping between the domains and codomains of the query and of the candidate, respectively, as specified by the type compatibility predicates. If the fully universal form is used, the body must additionally hold for arbitrary instances of the unrestricted arguments. Hence, they are essentially interpreted in a *don't know*-nondeterministic way, or as *surplus parameters* as the required functionality does not depend on their value. For example, if the contract

$$\begin{array}{l} \mathit{insert\text{-}some}\ (l : \mathit{list})\ r : \mathit{list} \\ \mathsf{pre}\ \ \mathsf{true} \\ \mathsf{post}\ \exists\, i : \mathit{item}, l_1, l_2 : \mathit{list} \cdot l = l_1 \curvearrowright l_2 \wedge r = l_1 \curvearrowright [i] \curvearrowright l_2 \end{array}$$

is used to retrieve components which insert an arbitrary single element at an arbitrary position into a list, then the usual *cons*-function

$$\begin{array}{l} \mathit{cons}\ (i : \mathit{item}, l : \mathit{list})\ r : \mathit{list} \\ \mathsf{pre}\ \ \mathsf{true} \\ \mathsf{post}\ r = [i] \curvearrowright l \end{array}$$

intuitively matches,[4] regardless which actual element is *cons*-ed in front of the list. In contrast to *cons*, the *append*-function

$$\begin{array}{l} \mathit{append}\ (l_1, l_2 : \mathit{list})\ r : \mathit{list} \\ \mathsf{pre}\ \ \mathsf{true} \\ \mathsf{post}\ r = l_1 \curvearrowright l_2 \end{array}$$

does not match *insert-some* under a fully universal prefix because none of the two argument lists can be guaranteed to be a singleton list. However, this can be fixed if the additional parameter (which can be either of l_1 and l_2) is fixed appropriately. The following definition reflects this idea.

[3] Throughout this section, I use $\mathcal{F}[\mathrm{pre}_q, \mathrm{pre}_c, \mathrm{post}_q, \mathrm{post}_c]$ to denote the body of an unspecified match predicate. This body usually refers to all bound variables $\vec{x}_q, \vec{y}_q, \vec{x_c} = \vec{x}_{c,u} \circ \vec{x}_{c,r}$ and $\vec{y}_c$.

[4] See the following sections for a detailed discussion of the exact nature of "matches." For the examples in this subsection, matching can be interpreted as equivalence of specifications (cf. Def. 3.2.2).

Definition 3.1.4 (curried universal form) *The* curried universal form *of a match predicate has the general structure*

$$\exists \vec{x}_{c,u} \forall \vec{x}_q \forall \vec{y}_q \forall \vec{x}_{c,r} \forall \vec{y}_c \cdot T_{\mathrm{dom}}(\vec{x}_q, \vec{x}_{c,r}) \wedge T_{\mathrm{cod}}(\vec{y}_q, \vec{y}_c) \Rightarrow \mathcal{F}[pre_q, pre_c, post_q, post_c]$$

for any body $\mathcal{F}$*.*

The curried universal form is named for its analogy to the currying technique which is used in functional programming languages to instantiate the first arguments of a function to certain values. And as with currying, the additional parameters are instantiated with fixed *values*: since the unrestricted formal arguments $\vec{x}_{c,u}$ are existentially quantified and precede all other bound variables in the prefix, the witnesses which satisfy this formula are *constants*. Now, using the curried universal form, *append* matches *insert-some*, as desired: if T identifies l with l_2 and both return values with each other, respectively, then the proof task effectively reduces (again modulo the exact definition of the match predicate) to

$$\exists\, l_1 : \mathit{list} \cdot \forall\, l_2 : \mathit{list} \cdot \exists\, i : \mathit{item} \cdot l_1 \curvearrowright l_2 = [i] \curvearrowright l_2$$

which becomes trivially true if some singleton list $[c]$ with an arbitrary element c is chosen for l_1.[5]

In connection with type compatibility predicates, currying already yields a powerful retrieval mechanism which is not confined to anonymous constants but can even "calculate" the required instantiations of the additional parameters. Consider for example *append* as query and the component

$$\begin{array}{l}
\mathit{insert\text{-}list}\ (l_1, l_2 : \mathit{list}, n : \mathbb{N})\ r : \mathit{list} \\
\mathsf{pre}\ \ n \leq \mathsf{len}\ l_1 \\
\mathsf{post}\ \mathsf{len}\ r = \mathsf{len}\ l_1 + \mathsf{len}\ l_2 \wedge \\
\qquad \forall\, m : \mathbb{N}_1 \cdot (m \leq n \ \Rightarrow\ r(m) = l_1(m)) \wedge \\
\qquad\qquad\quad (m \leq \mathsf{len}\ l_2 \ \Rightarrow\ r(m+n) = l_2(m)) \wedge \\
\qquad\qquad\quad (m > n \wedge m \leq \mathsf{len}\ l_1 \ \Rightarrow\ r(m + \mathsf{len}\ l_2) = l_1(m))
\end{array}$$

which inserts a list l_2 into another list l_1, after a given position n. *insert-list* matches but the match is not obvious. T must switch the order of the parameters and n must be fixed to zero: appending l_2 to l_1 is the same as inserting l_1 at position zero into l_2. But then, *append* can be expressed as a call to *insert-list*.

```
fun append l1 l2 = insert_list l2 l1 0
```

However, currying is not always sufficient. Sometimes, there exists no single value for which the match predicate holds, but a functional dependency from the query arguments to the additional parameters. This effect can be captured by nesting the existential binder one step deeper into the prefix.

[5]Note that the inner existential quantifier originates from the specification of *insert-some* and is not part of the curried universal prefix.

Definition 3.1.5 (functionally dependent universal form) *The* functionally dependent universal form *of a match predicate has the general structure*

$$\forall \vec{x}_q \exists \vec{x}_{c,u} \forall \vec{y}_q \forall \vec{x}_{c,r} \forall \vec{y}_c \cdot T_{\mathrm{dom}}(\vec{x}_q, \vec{x}_{c,r}) \wedge T_{\mathrm{cod}}(\vec{y}_q, \vec{y}_c) \Rightarrow \mathcal{F}[pre_q, pre_c, post_q, post_c]$$

for any body $\mathcal{F}$*.*

Now, the witnesses can be *functions* which depend on the outer query arguments $\vec{x}_q$. This dependency allows the query

$$\begin{array}{l} \mathit{front}\,(l : \mathit{list})\ r : \mathit{list} \\ \mathsf{pre}\ \ l \neq [\,] \\ \mathsf{post}\ \exists\, i : \mathit{item} \cdot l = r \curvearrowright [i] \end{array}$$

to retrieve the more general function

$$\begin{array}{l} \mathit{delete\text{-}segment}\,(l : \mathit{list}, n_1, n_2 : \mathbb{N})\ r : \mathit{list} \\ \mathsf{pre}\ \ n_1 + n_2 \leq \mathsf{len}\ l + 1 \\ \mathsf{post}\ \mathsf{len}\ r = \mathsf{len}\ l - n_2 \wedge \\ \qquad \forall\, m : \mathbb{N}_1 \cdot (m < n_1\ \Rightarrow\ r(m) = l(m)) \wedge \\ \qquad\qquad\qquad (m \geq n_1 + n_2 \wedge m \leq \mathsf{len}\ l\ \Rightarrow\ r(m - n_2) = l(m)) \end{array}$$

which deletes an arbitrary segment from the argument list by instantiating the parameters n_1 and n_2 appropriately:

```
fun front l = delete_segment l (len l) 1
```

However, the functional dependency supported by Def. 3.1.5 is *abstract* in the sense that it is a dependency in the query domain: if the query domain is considered as an abstract datatype, the witness can be composed entirely in terms of that datatype. Hence, retrieved components need not to be modified but can be adapted "from the outside"—reuse via the functionally dependent universal form follows the client-adaptive grey box style. This abstract view is not always sufficient and the functional dependency need to be expressed in the domain of the component, i.e., in the *implementation* of the datatype. This can again be achieved by nesting the existential binder one step deeper into the prefix.

Definition 3.1.6 (weak universal form) *The* weak universal form *of a match predicate has the general structure*

$$\forall \vec{x}_q \forall \vec{x}_{c,r} \exists \vec{x}_{c,u} \forall \vec{y}_q \forall \vec{y}_c \cdot T_{\mathrm{dom}}(\vec{x}_q, \vec{x}_{c,r}) \wedge T_{\mathrm{cod}}(\vec{y}_q, \vec{y}_c) \Rightarrow \mathcal{F}[pre_q, pre_c, post_q, post_c]$$

for any body $\mathcal{F}$*.*

Of course, this variant implies that the witness can no longer be constructed by the client and, hence, that the component must be modified or at least wrapped internally—it supports component-adaptive grey box reuse only.

3.1.4 Existential Prefixes

Universal prefixes embody the assumption that the body of the match predicate must be satisfied for *every* possible mapping between the domains and codomains of the query and of the candidate, respectively. This assumption ensures independence of the particular data representation but that can be too restrictive to yield good recall values. Existential prefixes thus relax the restriction by replacing some of the universal quantifiers by existential ones—hence the name. However, care must be taken with the bodies to avoid unintended matches and thus a loss of precision. Also, by using different bodies some of the effects could even be achieved under universal prefixes.

The cod-*existential form* is the least permissive relaxation. Here, the body of the match predicate must still hold for any possible mapping between the domains but not between the codomains. Instead, for each value in the codomain of the query only one corresponding value in the codomain of the component is required.

Definition 3.1.7 (cod-existential form) *The* cod-existential form *of a match predicate has the general structure*

$$\forall \vec{x}_q \forall \vec{x}_{c,r} \forall \vec{x}_{c,u} \forall \vec{y}_q \exists \vec{y}_c \cdot T_{\mathrm{dom}}(\vec{x}_q, \vec{x}_{c,r}) \Rightarrow (T_{\mathrm{cod}}(\vec{y}_q, \vec{y}_c) \wedge \mathcal{F}[pre_q, pre_c, post_q, post_c])$$

for any body $\mathcal{F}$*.*

Due to the universal quantification of the domains, the match is established independently of the representation of the arguments, i.e., the retrieved components may still be reused without further concern for the arguments. Consider for example a query for a set union

$$\begin{array}{l} \mathit{union}\,(s_1, s_2 : \mathit{set})\ r : \mathit{set} \\ \mathsf{pre}\ \ \mathsf{true} \\ \mathsf{post}\ r = s_1 \cup s_2 \end{array}$$

and assume that a *set* is implemented by an arbitrary *list* such that the members of both coincide, i.e., $T(s, l) \Leftrightarrow s = \mathsf{elems}\ l$. Obviously, under a universal prefix, *append* (cf. p. 42) does not match: each set has arbitrarily many valid representations as list, but $post_{_{append}}$ is valid only for one. However, independent of the chosen argument representation, this one is a valid representation of the result required by *union* and, hence, *append* matches under the cod-existential prefix.

The apparent drawback of the existential quantification is that no further assumptions about the results which the retrieved components produce are possible. Assume that Booleans are encoded by integers "as usual" (cf. p. 39) and consider the query

$$\begin{array}{l} \mathit{member}\,(l : \mathit{list}, i : \mathit{item})\ r : \mathbb{B} \\ \mathsf{pre}\ \ \mathsf{true} \\ \mathsf{post}\ r\ \ \Leftrightarrow\ \ \exists\, n \in \mathsf{inds}\ l \cdot l(n) = i \end{array}$$

which under a cod-existential prefix retrieves both components

$$\begin{array}{l} \mathit{numOccurrences}\,(l : \mathit{list}, i : \mathit{item})\ r : \mathbb{N} \\ \mathsf{pre}\ \ \mathsf{true} \\ \mathsf{post}\ r = \mathsf{card}\ \{n \mid n \in \mathsf{inds}\ l \cdot l(n) = i\} \end{array}$$

and

$$\begin{array}{l} \mathit{findLastOccurrence}\,(l : \mathit{list}, i : \mathit{item})\ r : \mathbb{N} \\ \mathsf{pre}\ \ \mathsf{true} \\ \mathsf{post}\ r = \mathit{max}(\{0\} \cup \{n \mid n \in \mathsf{inds}\ l \cdot l(n) = i\}) \end{array}$$

Yet, both components use different encodings for *true* and none uses the standard encoding $\mathit{true} \mapsto 1$. However, this is no valid objection: *any* assumptions on the retrieved components should be specified explicitly. Hence, if it is really important that the standard practice be followed, then a different type compatibility predicate must be used.

A further increase of recall can be achieved if the representation independence of the arguments is given up. Consider for example the query *member* and the component

$$\begin{array}{l} \mathit{isSegment}\,(l_1, l_2 : \mathit{list})\ r : \mathbb{B} \\ \mathsf{pre}\ \ \mathsf{true} \\ \mathsf{post}\ r\ \Leftrightarrow\ \exists\, l_3, l_4 : \mathit{list} \cdot l_1 = l_3 \frown l_2 \frown l_4 \end{array}$$

which checks whether the second argument occurs as sublist in the first and assume that the *item* i is identified with the *list* l_2 if it has the "right" head element, i.e., $T_{\mathrm{dom}}((i, l), (l_1, l_2)) \Leftrightarrow l = l_1 \wedge i = \mathsf{hd}\ l_2$. Still, *isSegment* cannot be retrieved because—in contrast to *union/append*—the representation of the argument is relevant: a proof is possible only for the canonical representation $i \mapsto [i]$.

Apart from "tweaking" the type compatibility predicate, there are generally two ways to make a proof possible. In the ideal case, T_{dom} is functional and the canonical injection function yields just the required representation. But even if it is relational, a restriction to the intended direction of interpretation (i.e., T_{dom}) increases the likelihood of a proof. At the same time and for the same reason, T_{cod} can also be restricted to its intended direction (i.e., T_{cod}).

Definition 3.1.8 (integrative form) *The* integrative form *of a match predicate has the general structure*

$$\forall \vec{x}_q \forall \vec{x}_{c,r} \forall \vec{x}_{c,u} \forall \vec{y}_q \exists \vec{y}_c \cdot T_{\mathrm{dom}}(\vec{x}_q, \vec{x}_{c,r}) \Rightarrow (T_{\mathrm{cod}}(\vec{y}_q, \vec{y}_c) \wedge \mathcal{F}[\mathit{pre}_q, \mathit{pre}_c, \mathit{post}_q, \mathit{post}_c])$$

for any body $\mathcal{F}$*.*

To some extent, the cod-existential and integrative forms still support black box reuse. If T is functional, the injection and retrieve functions take care of the

necessary argument and return value adaptations. If T_{dom} is relational, the body holds for any possible or canonic translation of the arguments, respectively. The critical point is the existential quantification of the return values in connection with non-deterministic component specifications: if a component can have several possible return values for a set of arguments, it is no longer guaranteed that each of them actually satisfies the body of the match condition.

In principle, different variants of the two forms can be derived by varying type and position of the quantifier of the unrestricted formal arguments in the same way as in the universal case. However, the reuse effects are also the same as in the universal case; I will occasionally use these variants but without explicit definitions.

In a far less ideal case as above, the choice of the argument representation is non-deterministic: for each argument there exists a suitable representation but not each (canonical) representation is necessarily suitable. For an example, consider again the implementation of sets by lists, as before, and the query

$$
\begin{array}{l}
\mathit{numElements}\,(s : \mathit{set})\ r : \mathbb{N} \\
\mathsf{pre}\ \ \mathsf{true} \\
\mathsf{post}\ r = \mathsf{card}\ s
\end{array}
$$

and component

$$
\begin{array}{l}
\mathit{length}\,(l : \mathit{list})\ r : \mathbb{N} \\
\mathsf{pre}\ \ \mathsf{true} \\
\mathsf{post}\ r = \mathsf{len}\ l
\end{array}
$$

Obviously, *length* does not match, not even under the integrative prefix because in this case only some of the possible list representations of a set are suitable—duplicate-free lists. The *existential form* reflects this situation and allows *length* to be retrieved.

Definition 3.1.9 (existential form) *The* existential form *of a match predicate has the general structure*

$$\forall \vec{x}_q \exists \vec{x}_{c,r} \exists \vec{x}_{c,u} \forall \vec{y}_q \exists \vec{y}_c \cdot T_{\text{dom}}(\vec{x}_q, \vec{x}_{c,r}) \wedge T_{\text{cod}}(\vec{y}_q, \vec{y}_c) \wedge \mathcal{F}[\mathit{pre}_q, \mathit{pre}_c, \mathit{post}_q, \mathit{post}_c]$$

for any body $\mathcal{F}$*.*

However, the increased recall is paid for with a loss of confidence. Since the actual representation of the arguments (which in some sense corresponds to the witness in the universal forms) is no longer known *a priori*, black box reuse is no longer supported. Similar to the functionally dependent universal forms (cf. Definitions 3.1.5 and 3.1.6), the arguments must be wrapped, either from outside or inside the component itself, such that reuse becomes grey-box style. Hence, the quantification of the unrestricted variables follows that model.

3.2 Exact Retrieval

3.2.1 Relevance Condition and Reuse Effects

For exact retrieval, the relevance condition can informally be stated as *behavioral equivalence*: a component is deemed relevant if and only if it behaves *under all conditions exactly* as required. Hence, it must also fail (even in the same way) whenever the query specifies a failure and must otherwise return exactly the same results the query demands.

However, behavioral equivalence is a rather rigid notion as a component must not "do more" (i.e., be more specific) than required. Consequently, if a query is nondeterministic, no actual component in a library can be behaviorally equivalent (assuming a deterministic implementation language) to the query, and, hence, nothing is relevant. For example, if a user inadvertently poses the nondeterministic query

$$
\begin{array}{l}
\mathit{query\text{-}sort}\ (l : \mathit{list})\ \ r : \mathit{list} \\
\mathsf{pre}\ \ \mathsf{true} \\
\mathsf{post}\ \mathit{sorted}(r) \wedge \forall\, i : \mathit{item} \cdot \mathit{member}(l, i)\ \ \Leftrightarrow\ \ \mathit{member}(r, i)
\end{array}
$$

to retrieve sort routines, no component can be relevant because *query-sort* does not specify whether duplicates are to be removed or not but each component has to make its choice about this.[6]

From the software engineering perspective, the rigidity of exact retrieval has also advantages—it supports *black box reuse* par excellence. Due to the behavioral equivalence, retrieved relevant components may obviously be plugged into the intended place "as is", without further proviso or modification. But it guarantees some even stronger properties than proper retrieval (cf. Section 3.3) which also supports black box reuse. In exact retrieval, components cannot have unintended and, hence, unnoticed side effects. They may thus be considered and exchanged in isolation, without regard to the environment. Such isolated changes are typical for re-engineering in general and software maintenance and system tuning in particular, e.g.,

- replacing a prototype by a more efficient but functionally equivalent implementation,
- calling an external component through a foreign language interface, or
- changing to a different version of a particular library.

[6] Incidentally, this example also highlights the roots of the relevance problem. If the retrieval goal is to retrieve sort functions without duplicate elimination, then the library could contain relevant components, even under the exact retrieval policy. But then the specification *query-sort* is just inadequate to retrieve them. To account for such effects, an accurate empirical evaluation should thus also average over different formalizations of the retrieval goals.

3.2.2 Rigid Match

The intuitive formalization of behavioral equivalence and thus also the usual match predicate body for exact retrieval is equivalence of the pre- and postconditions, respectively:

$$(pre_q \Leftrightarrow pre_c) \wedge (post_q \Leftrightarrow post_c) \tag{3.2}$$

Since the goal of exact retrieval is to retrieve components which can be considered in isolation, the body must also hold for any instantiation of the unrestricted parameters. Hence, the fully universal form (cf. Def. 3.1.3) must be used.

Definition 3.2.1 (rigid match) *For a query q and a component c,* rigid match μ_{rigid} *is defined as*

$$\begin{aligned}
&\forall \vec{x}_q \forall \vec{y}_q \forall \vec{x}_{c,u} \forall \vec{x}_{c,r} \forall \vec{y}_c \cdot \\
&\quad T_{\mathrm{dom}}(\vec{x}_q, \vec{x}_{c,r}) \wedge T_{\mathrm{cod}}(\vec{y}_q, \vec{y}_c) \Rightarrow \\
&\qquad \big(pre_q(\vec{x}_q) \Leftrightarrow pre_c(\vec{x}_c)\big) \wedge \big(post_q(\vec{x}_q, \vec{y}_q) \Leftrightarrow post_c(\vec{x}_c, \vec{y}_c))\big)
\end{aligned}$$

The same match predicate (except for the explicit notation of the unrestricted parameters) is called *exact pre/post match* in [Moo96, p. 43]. Its purpose is essentially to capture irrelevant syntactic differences in the specifications which can become rather large, as the following equivalent variants of *member* (cf. p. 45) show.

$$\begin{aligned}
&\mathit{member}\text{-}2\,(l : \mathit{list}, i : \mathit{item})\ r : \mathbb{B} \\
&\mathsf{pre}\ \ \mathsf{true} \\
&\mathsf{post}\ r \ \Leftrightarrow\ \exists\, l_1, l_2 : \mathit{list} \cdot l = l_1 \frown [i] \frown l_2
\end{aligned}$$

$$\begin{aligned}
&\mathit{member}\text{-}3\,(l : \mathit{list}, i : \mathit{item})\ r : \mathbb{B} \\
&\mathsf{pre}\ \ \mathsf{true} \\
&\mathsf{post}\ r \ \Leftrightarrow\ (l \neq [\,] \wedge (i = \mathsf{hd}\ l \vee \mathit{member}\text{-}3(\mathsf{tl}\ l, i))
\end{aligned}$$

Rigid match is in fact a very rigid relation between two specifications. They must not only have the same domains (modulo the type compatibility predicate) but must not even diverge outside their domains. Hence, the query

$$\begin{aligned}
&\mathit{split\text{-}after\text{-}first\text{-}occ}\text{-}1\,(l : \mathit{list}, i : \mathit{item})\ (r_1, r_2) : \mathit{list} \times \mathit{list} \\
&\mathsf{pre}\ \ \mathit{member}(l, i) \\
&\mathsf{post}\ l = r_1 \frown r_2 \wedge \exists\, l_1 : \mathit{list} \cdot r_1 = l_1 \frown [i] \wedge \neg\, \mathit{member}(l_1, i)
\end{aligned}$$

which splits a list l after the first occurrence of an item i (which must occur in l) does not match a slightly differently specified component

$$\begin{aligned}
&\mathit{split\text{-}after\text{-}first\text{-}occ}\text{-}2\,(l : \mathit{list}, i : \mathit{item})\ (r_1, r_2) : \mathit{list} \times \mathit{list} \\
&\mathsf{pre}\ \ \mathit{member}(l, i)
\end{aligned}$$

$$\begin{array}{l}\mathsf{post}\ l = r_1 \frown r_2 \wedge \\ \quad \forall\, l_1, l_2 : \mathit{list} \cdot l = l_1 \frown [i] \frown l_2 \wedge \neg\, \mathit{member}(l_1, i) \ \Rightarrow\ r_1 = l_1 \frown [i]\end{array}$$

which does *exactly* the same for all *legal* inputs but has a different error behavior: while $post_{\textit{split-after-first-occ-1}}$ is *false* for *any* r_1 and r_2 if the precondition is not satisfied, $post_{\textit{split-after-first-occ-2}}$ still is *true* for an arbitrary partition of l into r_1 and r_2 (i.e., *split-after-first-occ*-1 is minimal while *split-after-first-occ*-2 is neither minimal nor maximal).

3.2.3 Exact Match

Rigid match is particularly well-suited for reverse engineering where it is often necessary, e.g., for compatibility reasons, to duplicate even the exact error behavior. In most re-engineering applications, however, this is not the case. Instead, for an exact match it is sufficient that the retrieved components have the same domain and are equivalent on that domain.

Definition 3.2.2 (exact match) *For a query q and a component c,* exact match *μ_{exact} is defined as*

$$\begin{array}{l}\forall \vec{x}_q \forall \vec{y}_q \forall \vec{x}_{c,u} \forall \vec{x}_{c,r} \forall \vec{y}_c \cdot \\ \quad T_{\mathrm{dom}}(\vec{x}_q, \vec{x}_{c,r}) \wedge T_{\mathrm{cod}}(\vec{y}_q, \vec{y}_c) \Rightarrow \\ \qquad \big(\mathit{pre}_q(\vec{x}_q) \Leftrightarrow \mathit{pre}_c(\vec{x}_c)\big) \wedge \big(\mathit{pre}_q(\vec{x}_q) \Rightarrow (\mathit{post}_q(\vec{x}_q, \vec{y}_q) \Leftrightarrow \mathit{post}_c(\vec{x}_c, \vec{y}_c))\big)\big)\end{array}$$

Now, using exact match, *split-after-first-occ*-1 and *split-after-first-occ*-2 match each other, as intuitively expected. This behavior is not accidental, as the following easy corollaries show.

Corollary 3.2.3 *Rigid match implies exact match for any query and component.*

Lemma 3.2.4 *Exact match implies rigid match for any total (minimal, maximal, strictly deterministic) query and component.*
Proof: *For total specifications, the definition of μ_{exact} obviously collapses into μ_{rigid}; similarly, if $pre_q(\vec{x}_q)$ holds. Now assume $\neg pre_q(\vec{x}_q)$ which implies (from the first conjunct in the body of μ_{exact}) $\neg pre_c(\vec{x}_c)$ for all $\vec{x}_c$ such that $T_{\mathrm{cod}}(\vec{y}_q, \vec{y}_c)$ holds, because q and c match under μ_{exact}. Hence, if both q and c are minimal, $\neg post_q \vec{x}_q, \vec{y}_q)$ and $\neg post_c(\vec{x}_c, \vec{y}_c)$ must hold for all $\vec{y}_q$ and $\vec{y}_c$, respectively, which implies $\mu_{rigid}(q, c)$. In the cases of maximimality and strict determinacy similar arguments hold.* $\triangle$

Obviously, Lemma 3.2.4 cannot be "strengthened" to the converse direction, i.e., if exact match implies rigid match for q and c, then this does not entail that q and c are minimal: exact match is reflexive but not any specification is minimal. Similar observations also constrain the relations between most other match predicates.

However, the choice of pre_q to constrain the equivalence on the postconditions is arbitrary and for practical reasons, e.g., to obtain easier or at least alternative proof conditions, different equivalent variants of the body may be used. In particular, it is useful to break the equivalence between the two postconditions into two implications such that for each direction the respective precondition can be assumed:

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

3.2.4 Predicate Equivalence Matches

An alternative view of behavioral equivalence is not equivalence of the respective pre- and postconditions but equivalence of the entire specification predicates:

$$\mathcal{F}[pre_q, post_q] \Leftrightarrow \mathcal{F}[pre_c, post_c]$$

For the construction of the specification predicate $\mathcal{F}[pre, post]$ two interpretations prevail which both give rise to an equivalence match.

Under the *conjunctive interpretation*, the specification predicate is true if and only if both pre- and postcondition are true. This corresponds to a relational view of specifications: a predicate is identified with the set of valid *(input, output)*-pairs. Hence, the *conjunctive predicate equivalence match* can be interpreted as isomorphism (modulo the type compatibility predicate) between the respective sets of *(input, output)*-pairs.

Definition 3.2.5 (conjunctive predicate equivalence match) *For a query q and a component c,* conjunctive predicate equivalence match $\mu_{\wedge\text{-}pred\text{-}\equiv}$ *is defined as*

$$\begin{aligned}
&\forall \vec{x}_q \forall \vec{y}_q \forall \vec{x}_{c,u} \forall \vec{x}_{c,r} \forall \vec{y}_c \cdot \\
&\quad T_{\mathrm{dom}}(\vec{x}_q, \vec{x}_{c,r}) \wedge T_{\mathrm{cod}}(\vec{y}_q, \vec{y}_c) \Rightarrow \\
&\qquad \big(pre_q(\vec{x}_q) \wedge post_q(\vec{x}_q, \vec{y}_q) \Leftrightarrow pre_c(\vec{x}_c) \wedge post_c(\vec{x}_c, \vec{y}_c)\big)
\end{aligned}$$

Similarly to exact match, the equivalence check on the postconditions is also "blocked out" if *both* preconditions are *false*. Hence, *split-after-first-occ*-1 and *split-after-first-occ*-2 also match each other using conjunctive predicate equivalence match. However, since both preconditions are not directly related to each other, accidental matches may spoil the precision, i.e., the match predicate becomes true although the specifications of query and component are not equivalent. For example, consider the well-formed but non-implementable query

query-total-front $(l : list)\ r : list$
pre true
post $\exists\, i : item \cdot l = r \curvearrowright [i]$

and the candidate component *front*. Although *front* has a smaller domain than required (and is thus not retrieved under the rigid and exact matches), it matches under the conjunctive predicate equivalence match, because *query-total-front* is non-implementable and thus both specification predicates become *false* on the domain difference (i.e., for $l = [\,]$).

Fortunately, this kind of behavior can occur only for "pathological" (i.e., non-implementable) specifications, as the subsequent Lemma 3.2.7 shows.

Corollary 3.2.6 *Exact match implies conjunctive predicate equivalence match for any query and component.*

Lemma 3.2.7

1. *Conjunctive predicate equivalence match implies exact match for any implementable query and component.*
2. *Conjunctive predicate equivalence match implies rigid match for any implementable, minimal query and component.*

Proof: *Assume that* $\mu_{\wedge\text{-}pred\text{-}\equiv}(q, c)$ *holds but not* $\mu_{exact}(q, c)$ *or* $\mu_{rigid}(q, c)$*, respectively. Then* $\vec{x}_q$ *and* $\vec{x}_c$ *must exist such that* $T_{\mathrm{dom}}(\vec{x}_q, \vec{x}_{c,r})$ *and either:*

- *The left conjunct of exact or rigid match is wrong, i.e.,* $pre_q(\vec{x}_q) \Leftrightarrow \neg pre_c(\vec{x}_c)$ *and* $\forall \vec{y}_q \forall \vec{y}_c \cdot T_{\mathrm{cod}}(\vec{y}_q, \vec{y}_c) \Rightarrow \big(pre_q(\vec{x}_q) \wedge post_q(\vec{x}_q, \vec{y}_q) \Leftrightarrow pre_c(\vec{x}_c) \wedge post_c(\vec{x}_c, \vec{y}_c)\big)$*. Assume that* $pre_q(\vec{x}_q)$ *holds. Thus* $\neg pre_c(\vec{x}_c)$ *and the specification predicate of c is consistently false. Hence, because c matches onto q by assumption,* $\forall \vec{y}_q \cdot \neg post_q(\vec{x}_q, \vec{y}_q)$ *must hold but this is a contradiction to the assumption that the query is implementable. In the case of* $\neg pre_q(\vec{x}_q)$*, the contradiction follows by symmetry from the implementability of the component.*
- $pre_q(\vec{x}_q) \Leftrightarrow pre_c(\vec{x}_c)$ *and the right conjunct of exact and rigid match is wrong. For exact match this implies that* $pre_q(\vec{x}_q)$ *holds, hence* $pre_c(\vec{x}_c)$*, and, because q and c match under conjunctive predicate equivalence, then also* $\forall \vec{y}_q \forall \vec{y}_c \cdot T_{\mathrm{cod}}(\vec{y}_q, \vec{y}_c) \Rightarrow \big(post_q(\vec{x}_q, \vec{y}_q) \Leftrightarrow post_c(\vec{x}_c, \vec{y}_c)\big)$ *which is a contradiction to* $\neg\mu_{exact}(q, c)$*. For rigid match, the case that* $pre_q(\vec{x}_q)$ *holds leads to the same contradiction. Hence,* $\neg pre_q(\vec{x}_q)$ *must hold, thus also* $\neg pre_c(\vec{x}_c)$ *but then the postconditions coincide since query and component are supposed to be minimal, i.e., a contradiction to* $\neg\mu_{rigid}(q, c)$ *arises.*

△

For retrieval purposes it would be more appropriate to shift the restrictions on the components, leaving the queries and thus the users unrestricted. However, as the above example shows, exact match and conjunctive predicate equivalence match do not even coincide for reasonable libraries. The proof of Lemma 3.2.7 gives only the following corollary (and its converse) but this is the best we can achieve.

Corollary 3.2.8 *Conjunctive predicate equivalence match implies exact match for any query and any implementable, total component.*

Under the *implicative interpretation*, the specification predicate is true if and only if the truth of the precondition implies the truth of the postcondition. This corresponds to the usual contract-based view of specifications: if the precondition is broken, anything may happen; if the component is called nevertheless, it may react arbitrarily.

Definition 3.2.9 (implicative predicate equivalence match) *For a query q and a component c,* implicative predicate equivalence match *is defined as*

$$\begin{aligned}&\forall \vec{x}_q \forall \vec{y}_q \forall \vec{x}_{c,u} \forall \vec{x}_{c,r} \forall \vec{y}_c \cdot \\ &\quad T_{\mathrm{dom}}(\vec{x}_q, \vec{x}_{c,r}) \wedge T_{\mathrm{cod}}(\vec{y}_q, \vec{y}_c) \Rightarrow \\ &\qquad \big(pre_q(\vec{x}_q) \Rightarrow post_q(\vec{x}_q, \vec{y}_q)\big) \Leftrightarrow \big(pre_c(\vec{x}_c) \Rightarrow post_c(\vec{x}_c, \vec{y}_c)\big)\big)\end{aligned}$$

Moorman Zaremski prefers the implicative interpretation of specifications; in [Moo96, p. 48], the above implicative variant is called *exact predicate match*. Again, the equivalence check on the postconditions is "blocked out" if *both* preconditions are *false* and again *split-after-first-occ*-1 and *split-after-first-occ*-2 match onto each other. And as under the conjunctive interpretation, it is a again relaxation of exact match.

Corollary 3.2.10 *Exact match implies implicative predicate equivalence match for any query and component.*

However, the reverse direction of the preceding corollary as well as the relation between both predicate equivalence matches are more complicated. Although the implicative interpretation follows from the conjunctive, this does not directly scale up to the match predicates, due to the accidental matches which non-implementable specifications may cause under the conjunctive interpretation.

Corollary 3.2.11 *Conjunctive predicate equivalence match implies implicative predicate equivalence match for any implementable query and component.*

The attempt to show the reverse direction exposes a fallacy of the implicative predicate equivalence match which is similar to but more serious than the one described for the conjunctive predicate equivalence match (cf. page 51). Again, components with a "wrong" (i.e., too small) domain may be retrieved under certain circumstances. This fallacy is caused by the position of the preconditions, i.e., by the implicative interpretation. It occurs for example if the query

$$\begin{aligned}&\mathit{query\text{-}completed\text{-}front}\ (l : \mathit{list})\ r : \mathit{list} \\ &\mathsf{pre}\ \ \mathsf{true} \\ &\mathsf{post}\ l \neq [\,] \ \Rightarrow\ (\exists\, i : \mathit{item} \cdot l = r \frown [i])\end{aligned}$$

which can be considered as a partial specification for a *total* variant of the *front* function is checked against the original version (cf. p. 44). Obviously, both specifications are equivalent for non-empty lists but not even the case $l = [\,]$ prevents a match. Although both the pre- ($pre_{\textit{query-completed-front}}([\,])$ vs. $\neg pre_{\textit{front}}([\,])$) and postconditions ($\forall r : \textit{list} \cdot post_{\textit{query-completed-front}}([\,], r)$ vs. $\forall r : \textit{list} \cdot \neg post_{\textit{front}}([\,], r)$) differ, the implicative specification predicates are equivalent. A closer inspection reveals that such an accidental match can only happen if the component can be completed arbitrarily because the query is trivial (i.e., indiscriminate) on the extended domain. Hence:

Lemma 3.2.12 *Implicative predicate equivalence match implies conjunctive predicate equivalence match for any non-trivial query and component.*
Proof: *Assume that $\mu_{\Rightarrow\text{-}pred\text{-}\equiv}(q, c)$ holds but not $\mu_{\wedge\text{-}pred\text{-}\equiv}(q, c)$. Then $\vec{x}_q$ and $\vec{x}_c$ must exist such that $T_{\mathrm{dom}}(\vec{x}_q, \vec{x}_{c,r})$ and $pre_q(\vec{x}_q) \Leftrightarrow \neg pre_c(\vec{x}_c)$, due to the propositional structure of both match predicates. If $pre_q(\vec{x}_q)$ holds, $\forall \vec{y}_q \cdot post_q(\vec{x}_q, \vec{y}_q)$ must also hold because the implicative specification predicate of c is equivalent to true and q and c match under $\mu_{\Rightarrow\text{-}pred\text{-}\equiv}$ but this is a contradiction to the non-triviality of q. If $\neg pre_q(\vec{x}_q)$ holds, $pre_c(\vec{x}_c)$ must hold, and a similar contradiction arises due to the non-triviality of c.* $\triangle$

Generally, the differences between the match predicates for exact retrieval manifest themselves only if some border conditions are not met. For two reasonable classes of specifications, all the above predicates coincide. Obviously, all preconditions "reduce away" if they are true for all inputs, leaving a simple equivalence of the respective postconditions. Hence:

Corollary 3.2.13 *For total components and queries, the rigid, exact, and conjunctive and implicative predicate equivalence matches coincide.*

A second reasonable class originates from coalescing the side conditions of the preceding corollaries.

Corollary 3.2.14 *For implementable non-trivial minimal components and queries, the rigid, exact, and conjunctive and implicative predicate equivalence matches coincide.*

It is easy to verify that the respective proofs still hold if non-triviality (and minimality) are substituted by (strict) determinism, provided that the codomains are non-trivial (i.e., contain at least two different elements) and the type compatibility predicates are strictly adequate. Total specifications, however, are not necessarily subsumed by the latter classes as, e.g., the simplest total specification for lists,

$\textit{total}\,(l : \textit{list})\ r : \textit{list}$
pre true

post true

is neither non-trivial nor deterministic nor minimal. Hence, Corollaries 3.2.13 and 3.2.14 describe two different classes of components.

3.2.5 Prefix Variations

Obviously, the number of retrieved components can be increased if the fully universal prefixes are replaced by weaker universal variants (except for the weak universal form which does not support black-box reuse). However, these variations make sense only if the relevance condition is slightly modified: the additionally retrieved components can no longer behave exactly as required *under all conditions* (cf. p. 48) precisely because the unrestricted parameters may have been instantiated for the proof.

Behavioral equivalence requires independence of a particular data representation in the domain *and* codomain and the universal prefixes assure that. But if the type compatibility is not both strictly adequate and functional, the complete "cross coverage" of the two codomains is too strict and may result in a loss of recall (cf. the discussion following Def. 3.1.7). Using the cod-existential form prevents this loss.

Definition 3.2.15 (representation-independent exact match) *For a query q and a component c,* representation-independent exact match *is defined as*

$$\begin{aligned}
&\forall \vec{x}_q \forall \vec{x}_{c,u} \forall \vec{x}_{c,r} \forall \vec{y}_q \exists \vec{y}_c \cdot \\
&\quad T_{\mathrm{dom}}(\vec{x}_q, \vec{x}_{c,r}) \Rightarrow \\
&\qquad T_{\mathrm{cod}}(\vec{y}_q, \vec{y}_c) \wedge \big(pre_q(\vec{x}_q) \Leftrightarrow pre_c(\vec{x}_c)\big) \\
&\qquad\quad \wedge \big(pre_q(\vec{x}_q) \Rightarrow (post_q(\vec{x}_q, \vec{y}_q) \Leftrightarrow post_c(\vec{x}_c, \vec{y}_c))\big)
\end{aligned}$$

The *union/append*-example (cf. p. 45) demonstrates the increased recall compared to the standard version of exact match. The precise relation between the two variants is captured by the following corollaries.

Corollary 3.2.16 *Exact match implies representation-independent exact match for any query and component.*

Lemma 3.2.17 *For an adequate and functional type compatibility predicate T_{cod} (i.e., a total function), representation-independent exact match implies exact match for any query and component.*

Proof: *By adequacy, $\forall \vec{y}_q \exists \vec{y}_c \cdot T_{\mathrm{cod}}(\vec{y}_q, \vec{y}_c)$ always holds, and by functionality this $\vec{y}_c$ is unique. Hence, $\forall \vec{y}_q \exists \vec{y}_c \cdot T_{\mathrm{cod}}(\vec{y}_q, \vec{y}_c) \wedge \mathcal{F}[\vec{x}_q, \vec{x}_c, \vec{y}_q, \vec{y}_c]$ is equivalent to $\forall \vec{y}_q \forall \vec{y}_c \cdot T_{\mathrm{cod}}(\vec{y}_q, \vec{y}_c) \Rightarrow \mathcal{F}[\vec{x}_q, \vec{x}_c, \vec{y}_q, \vec{y}_c]$.* $\triangle$

Moreover, *deterministic* components which are retrieved *only* under the representation-independent exact match can (via the injection and retrieve functions) still be used to implement the query *without any observable difference* but they are not behaviorally equivalent in the sense of (3.2); again, the *union/append*-example demonstrates this.

3.2.6 Equivalence Properties

The relevance condition of exact retrieval, behavioral *equivalence*, as well as some of the chosen match names suggest that the match predicates defined in this section are equivalence relations over the set of components. A. Moorman Zaremski [Moo96] states (without proof) that rigid and implicative predicate equivalence match are in fact equivalence matches, claiming that "proving that the matches are equivalences or partial orders is straightforward and based on the properties of [propositional equivalence and implication]." [Moo96, p. 50f].

However, this is not the whole truth: while it is easy to see that the propositional structures of the respective bodies satisfy the requirements for an equivalence relation, it is impossible to show this for the full match predicates *without imposing additional assumptions on the type compatibility predicates.*

Reflexivity

If the specifications for q and c are identical expressions, the type compatibility predicates must relate $\vec{x}_q$ to $\vec{x}_{c,r}$ and $\vec{y}_q$ to $\vec{y}_c$; hence, $\vec{x}_{c,u} = \emptyset$. The definition of rigid match then simplifies to

$$\begin{aligned}&\forall \vec{x}_q \forall \vec{y}_q \forall \vec{x}_{c,r} \forall \vec{y}_c \cdot \\ &\quad T_{\text{dom}}(\vec{x}_q, \vec{x}_{c,r}) \wedge T_{\text{cod}}(\vec{y}_q, \vec{y}_c) \Rightarrow \\ &\qquad \big(\text{pre}_c(\vec{x}_q) \Leftrightarrow \text{pre}_c(\vec{x}_c)\big) \wedge \big(\text{post}_c(\vec{x}_q, \vec{y}_q) \Leftrightarrow \text{post}_c(\vec{x}_c, \vec{y}_c)\big)\big)\end{aligned}$$

which is valid only if both T_{dom} and T_{cod} are simply the equality predicate. This observation also holds for the other match predicates.

Consequently, reflexivity for all the above matches holds only under the assumption that (one variant of) the type compatibility predicate is the equality predicate.

Symmetry

The main problem in showing symmetry (and also transitivity) is that in general for each of domain and codomain two different type compatibility predicates T and T' may be involved. These two can then of course not be completely arbitrary but must be compatible in some sense.

For symmetry, this means that

$$T_{\mathrm{dom}}(x, y) \Leftrightarrow T'_{\mathrm{dom}}(y, x)$$

must hold (and similarly for the codomain predicate T_{cod}); under these assumptions, symmetry is indeed straightforward. Again, if equality is used as type compatibility predicate, these assumptions hold trivially.

3.3 Proper Retrieval

3.3.1 Relevance Condition and Reuse Effects

For proper retrieval, the relevance condition can informally be stated as *substitutivity*: a component is deemed relevant if and only if it returns correct results on the required domain. This is in fact a generalization of exact retrieval because it relaxes behavioral equivalence in two ways:

- The domains need not to be equal and the components behavior outside the domain explicitly fixed by the query is no longer relevant. It may fail, in any way, but it may also return an arbitrary result.

- On the required domain, results need not to be exactly the same but only "correct": components are free to return more specific results than specified. Hence, non-determinism in queries is no longer mandatory (i.e., *don't know*) to the components and thus an obstacle to retrieval but a choice (i.e., *don't care*) and thus benefits retrieval.

From the software engineering perspective, both aspects are very important:

- Library components are often implemented in a very general way and thus have a wider domain than expected.

- Library components are often implemented in a very defensive way and have a defined behavior outside the usual domain but this behavior need not satisfy the usual specification. For example, the *front*-function may return a *nil*-pointer when it is called with the empty list.

- Library components are often implemented using sophisticated algorithms and may thus return more specific results than anticipated, e.g., a key-stable sort function.

- Non-deterministic queries allow the users to concentrate on the aspects they consider to be essential.

Consequently, whenever the focus is not on rigidity and isolated changes as in software modification but on a more liberal notion of software construction by parts composition, proper retrieval is more appropriate. The composition process is still safe, because proper retrieval also supports black box reuse. Due to the substitutivity, all retrieved components are guaranteed to satisfy the explicitly required behavior; any additional behavior is either more specific and not in contradiction to the query or visible only outside the required domain.

3.3.2 Proper Matches

The intuitive formalization of substitutivity can be derived by relaxing the formalization of behavioral equivalence (cf. Equation 3.2) appropriately. Since the domains need no longer be equal, the equivalence on the preconditions is replaced by an implication that reflects the possibly wider domain of the component: $pre_q \Rightarrow pre_c$. Similarly, the equivalence on the postconditions is also replaced by an implication, but this time the order is reversed, reflecting the fact that now the codomain of the component need only to be contained in that of the query. Additionally, this implication can be restricted to the explicitly required domain (cf. also exact match, Def. 3.2.2). Hence:

$$(pre_q \Rightarrow pre_c) \wedge (pre_q \wedge post_c \Rightarrow post_q) \tag{3.3}$$

This is a quite common formula. Under the relational view of specifications, it corresponds exactly to the notion of relational refinement which R. Mili et al [MMM97] use. Implicitly, (3.3) also occurs in the reification process in model-oriented specification methods. Here, both conjuncts are considered separately; they are known as *domain* and *result rule* [Jon90, p. 190], respectively. This separation emphasizes their different roles: the domain rule reduces undefinedness, the result rule reduces non-determinism [BF+93, p. 184f].

Proper match takes Formula (3.3) as body and wraps it into a fully universal quantifier prefix.

Definition 3.3.1 (proper match) *For a query q and a component c,* proper match μ_{proper} *is defined as*

$$\begin{aligned}
&\forall \vec{x}_q \forall \vec{y}_q \forall \vec{x}_{c,u} \forall \vec{x}_{c,r} \forall \vec{y}_c \cdot \\
&\quad T_{\mathrm{dom}}(\vec{x}_q, \vec{x}_{c,r}) \wedge T_{\mathrm{cod}}(\vec{y}_q, \vec{y}_c) \Rightarrow \\
&\qquad \big(pre_q(\vec{x}_q) \Rightarrow pre_c(\vec{x}_c)\big) \wedge \big(pre_q(\vec{x}_q) \wedge post_c(\vec{x}_c, \vec{y}_c) \Rightarrow post_q(\vec{x}_q, \vec{y}_q)\big)
\end{aligned}$$

Proper match is generally considered to be the canonical formalization of proper retrieval. It thus appears throughout the body of the deduction-based software component retrieval literature, although under a variety of different names (e.g., (relaxed) plug-in match [SF97, CC99], satisfies match [PBA95, Pen98, PA99], refinement [MMM97] or behavioral subtyping [Ame90, LW94]).

As exact match, proper match supports black box reuse but it also provides all the additional advantages substitutivity offers for component retrieval purposes over behavioral equivalence, as discussed in Section 3.3.1.

For an example of the more appropriate handling of domains which are wider than anticipated, consider again the query *front* (p. 44) and a possible total implementation

$$\begin{array}{l} \mathit{front\text{-}total}\,(l : \mathit{list})\ r : \mathit{list} \\ \mathsf{pre}\ \ \mathsf{true} \\ \mathsf{post}\ (l = [] \wedge r = []) \vee (\exists\, i : \mathit{item} \cdot l = r \frown [i]) \end{array}$$

which returns the empty list for an input consisting of the empty list. Clearly, *front-total* is not retrieved under any of the matches for exact retrieval[7] but, as expected, it is retrieved under proper match because pre_{front} restricts the codomain test appropriately and thus prevents the completion $(l = [] \wedge r = [])$ from rendering the proof impossible.

front-total also illustrates the non-determinism handling of proper match. Consider the query

$$\begin{array}{l} \mathit{segment\text{-}total}\,(l : \mathit{list})\ r : \mathit{list} \\ \mathsf{pre}\ \ \mathsf{true} \\ \mathsf{post}\ \exists\, l_1, l_2 : \mathit{list} \cdot l = l_1 \frown r \frown l_2 \end{array}$$

which queries for functions returning an arbitrary sublist. Under proper match, *front-total* matches because it returns a *specific* sublist—the empty list for the empty input, the front segment otherwise. This is no longer sufficient for the equivalence matches used for exact retrieval. The query explicitly allows that for example $([i, i], [])$ is a *legal* and thus under any of the equivalence matches also *required* input/output-pair, but obviously *front-total* does not provide that. This situation is similar to the one which distinguishes rigid and exact match (cf. p. 49); however, there the reasoning was confined to illegal inputs while it here also applies to the legal inputs.

By construction, proper match is a relaxation of the exact match used in exact retrieval. To show any relation in the reverse direction (i.e., to conclude from proper match to exact match), it is necessary to recover the dropped directions of the equivalences. This can be accomplished by reversing the role of query and component. It is thus easy to see that the following two corollaries hold.

Corollary 3.3.2 *Exact match implies proper match for any query and component.*

[7]For rigid and exact match it is easy to see that the equivalences fail due to the different domains; for the implicative predicate equivalence match consider the input/output-pair $([], [i])$ for an arbitrary item i, which makes the specification predicate *true* for the query but *false* for the actual component. For conjunctive predicate equivalence match consider the input/output-pair $([], [])$ which makes the specification predicate *false* for the query but *true* for the actual component.

Corollary 3.3.3 *Any two specifications for which proper match holds in both directions also match under exact match, i.e.,*

$$\forall k_1, k_2 \in \mathcal{K} \cdot \mu_{proper}(k_1, k_2) \wedge \mu_{proper}(k_2, k_1) \Rightarrow \mu_{exact}(k_1, k_2)$$

The reverse direction of the preceding corollary does not hold in general; however, it follows from Corollary 3.3.2 if the type compatibility predicates are sufficiently restricted such that μ_{exact} becomes symmetric.

It is conceivable to define an intermediate variant between exact and proper match, which only applies the result rule, i.e., guarantees identical domains but allows more specific results. However, the usefulness of such a match definition is doubtful.

Some other variations of proper match, however, are used in practice. The two most common variations result from (syntactic) modifications of the *result rule guard* (i.e., the occurrence of pre_q in the second conjunct of Formula 3.3); they thus vary the domain on which the result rule must hold.

The first variant, *strict plug-in match*, dates back to one of the original[8] papers on specification matching by E. Rollins and J. Wing [RW91] and is thus widely used, especially in early work [MW95b, FKS95c].[9] It drops the guard completely, i.e., replaces it by *true*.

Definition 3.3.4 (strict plug-in match) *For a query q and a component c,* strict plug-in match μ_{proper} *is defined as*

$$\begin{aligned}
&\forall \vec{x}_q \forall \vec{y}_q \forall \vec{x}_{c,u} \forall \vec{x}_{c,r} \forall \vec{y}_c \cdot \\
&\quad T_{\text{dom}}(\vec{x}_q, \vec{x}_{c,r}) \wedge T_{\text{cod}}(\vec{y}_q, \vec{y}_c) \Rightarrow \\
&\qquad \big(pre_q(\vec{x}_q) \Rightarrow pre_c(\vec{x}_c)\big) \wedge \big(post_c(\vec{x}_c, \vec{y}_c) \Rightarrow post_q(\vec{x}_q, \vec{y}_q)\big)
\end{aligned}$$

The intuition behind strict plug-in match is basically the same as behind proper match: since the component requires less but grants more than the query demands, it can be "plugged into the place" of the query. Figure 3.3 which is adapted from [MW95b] illustrates this process; it also shows that the guard is lost if the illustrative picture is directly (re-) translated into the match definition.

However, for partial functions, strict plug-in match is strictly stronger (i.e., retrieves fewer components) than proper match. In the above example, it fails to retrieve *front-total* for the query *front* precisely because *front-total* does not implement a proper *front* function on its entire domain.

Corollary 3.3.5 *Strict plug-in match implies proper match for any query and component.*

[8] The technical report [RW90b] of the same title as [RW91] is dated October 3, 1990 and is to the best of my knowledge the first publication which uses the phrase "specification matching".

[9] In the cited papers it is generally called *plug-in match*.

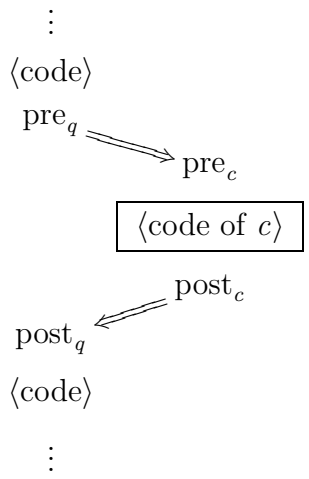


Figure 3.3: Plug-in compatibility and strict plug-in match

Corollary 3.3.6 *Proper match implies strict plug-in match for any total query and any component.*

This loss of recall is partially compensated by a greater robustness of the retrieved components. Since they provide the required functionality unconditionally (i.e., over the entire domain given by the argument types), they are insensitive to changes of or errors in the query domain. This in turn allows to re-check in such cases only the left conjunct of the match (i.e., the implication between the preconditions) which leads to simpler proof tasks.

The second variant, *robust plug-in match*, uses the component's precondition as guard. This yields a weaker match definition than strict plug-in match and thus a higher recall, but does not compromise robustness (in the sense described above) because the component still delivers the requested results for its own entire domain and not only for the query domain. Robust plug-in match is also known as *weak plug-in* match [Pen98].

Definition 3.3.7 (robust plug-in match) *For a query q and a component c,* robust plug-in match μ_{robust} *is defined as*

$$\begin{aligned}&\forall \vec{x}_q \forall \vec{y}_q \forall \vec{x}_{c,u} \forall \vec{x}_{c,r} \forall \vec{y}_c .\\ &\quad T_{\mathrm{dom}}(\vec{x}_q, \vec{x}_{c,r}) \wedge T_{\mathrm{cod}}(\vec{y}_q, \vec{y}_c) \Rightarrow \\ &\qquad \big(pre_q(\vec{x}_q) \Rightarrow pre_c(\vec{x}_c)\big) \wedge \big(pre_c(\vec{x}_c) \wedge post_c(\vec{x}_c, \vec{y}_c) \Rightarrow post_q(\vec{x}_q, \vec{y}_q)\big)\end{aligned}$$

However, robust plug-in match is also strictly stronger than proper match and still fails to retrieve *front-total* for the query *front*, for the same reasons as above.

3.3.3 Predicate Subsumption Matches

Similarly to the situation in exact retrieval, substitutivity can also be interpreted as a relation between the entire specification predicates:

$$\mathcal{F}[pre_q, post_q] \dagger \mathcal{F}[pre_c, post_c]$$

However, the exact nature of the relation † is not obvious; moreover, the suitability of any given relation heavily depends on the interpretation of the specification predicate and the nature of the queries and components.

In [Moo96, MW97b], A. Moorman Zaremski and J. Wing use implication in both directions as the relation † but give it different interpretations ("generalization" and "specialization"). Substitutivity, however, can also be interpreted in both directions:

- From the query to the component: whenever the specification predicate of the query holds, that of the component must also hold. Under the implicative interpretation, this direction is the "specialized match" of [Moo96, MW97b].
- From the component to the query: the specification predicate of the query is logically weaker than that of the component. Under the implicative interpretation, this direction is the "generalized match" of [Moo96, MW97b].

The confusion arises from the fact that the polarity of the query precondition (i.e., whether in a disjunctive normal form pre_q is negated or not) depends on the chosen interpretation of the specification predicate. Obviously, pre_q should be negated (and pre_c not) to capture the domain rule adequately (cf. Equation 3.3). However, under the conjunctive interpretation the respective pre- and postconditions have the same polarity. This in turn implies that each choice of a direction tackles only one aspect of substitutivity correctly.

Definition 3.3.8 (predicate subsumption matches) *For a query q and a component c,* left *and* right conjunctive predicate subsumption match *are defined as*

$$\begin{aligned}
&\forall \vec{x}_q \forall \vec{y}_q \forall \vec{x}_{c,u} \forall \vec{x}_{c,r} \forall \vec{y}_c \cdot \\
&\quad T_{\mathrm{dom}}(\vec{x}_q, \vec{x}_{c,r}) \wedge T_{\mathrm{cod}}(\vec{y}_q, \vec{y}_c) \Rightarrow \\
&\qquad \big(pre_q(\vec{x}_q) \wedge post_q(\vec{x}_q, \vec{y}_q) \Leftarrow pre_c(\vec{x}_c) \wedge post_c(\vec{x}_c, \vec{y}_c)\big)
\end{aligned}$$

and

$$\begin{aligned}
&\forall \vec{x}_q \forall \vec{y}_q \forall \vec{x}_{c,u} \forall \vec{x}_{c,r} \forall \vec{y}_c \cdot \\
&\quad T_{\mathrm{dom}}(\vec{x}_q, \vec{x}_{c,r}) \wedge T_{\mathrm{cod}}(\vec{y}_q, \vec{y}_c) \Rightarrow \\
&\qquad \big(pre_q(\vec{x}_q) \wedge post_q(\vec{x}_q, \vec{y}_q) \Rightarrow pre_c(\vec{x}_c) \wedge post_c(\vec{x}_c, \vec{y}_c)\big)
\end{aligned}$$

respectively. The left *and* right implicative predicate subsumption match *are defined as*

$$\forall \vec{x}_q \forall \vec{y}_q \forall \vec{x}_{c,u} \forall \vec{x}_{c,r} \forall \vec{y}_c \cdot \\ T_{\mathrm{dom}}(\vec{x}_q, \vec{x}_{c,r}) \wedge T_{\mathrm{cod}}(\vec{y}_q, \vec{y}_c) \Rightarrow \\ \big((pre_q(\vec{x}_q) \Rightarrow post_q(\vec{x}_q, \vec{y}_q)) \Leftarrow (pre_c(\vec{x}_c) \Rightarrow post_c(\vec{x}_c, \vec{y}_c))\big)$$

and

$$\forall \vec{x}_q \forall \vec{y}_q \forall \vec{x}_{c,u} \forall \vec{x}_{c,r} \forall \vec{y}_c \cdot \\ T_{\mathrm{dom}}(\vec{x}_q, \vec{x}_{c,r}) \Rightarrow T_{\mathrm{cod}}(\vec{y}_q, \vec{y}_c) \Rightarrow \\ \big((pre_q(\vec{x}_q) \Rightarrow post_q(\vec{x}_q, \vec{y}_q)) \Rightarrow (pre_c(\vec{x}_c) \wedge post_c(\vec{x}_c, \vec{y}_c))\big)$$

respectively.

By construction, each predicate subsumption match is a relaxation of the respective predicate equivalence match, i.e., the following corollary holds trivially.

Corollary 3.3.9 *For any query and component,*

1. *conjunctive predicate equivalence match implies left and right conjunctive predicate subsumption match,*
2. *implicative predicate equivalence match implies left and right implicative predicate subsumption match.*

3.4 Approximate Retrieval

3.4.1 Relevance Conditions and Reuse Effects

Under exact and proper retrieval, relevant components constitute *complete* solutions for the stated problem (i.e., query). Under approximate retrieval a different policy is used. Here relevant components need to solve the problem only partially or even need to be modified to yield a solution. This policy gives rise to several different suitable relevance conditions.

In the specification-based case, partial solutions are usually defined by *undefinedness*: a component is already considered as relevant if it returns correct results *on its own domain* which may be smaller than required—that is, even if it is less defined than required.

Similarly to proper retrieval, approximate retrieval is under this relevance condition aimed at software construction by parts composition. The composition process can still be safe but the client has to satisfy open obligations which result from the preconditions of the retrieved components. This can be done by repeated queries using *follow-up contracts* which are determined by the respective composition step:

- *Sequential composition* or functional layering of components can be considered as "trading" the original obligation $post_q$ against the usually simpler pre_c, yielding (pre_q, pre_c) as follow-up contract.
- *Alternative composition* or adjoining of components can be considered as "discounting" pre_c from the original contract. It leaves $(pre_q \wedge \neg pre_c, post_q)$ as follow-up contract which may be narrowed further by adjoining more components already retrieved by the original query or may be closed (completely) by a new query.

Figures 3.4 and 3.5 illustrate how the follow-up contracts can be used to bridge the original gap between the pre- and postcondition of the query.

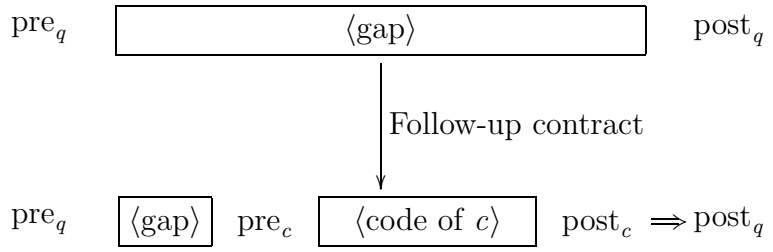


Figure 3.4: Follow-up contracts for sequential composition

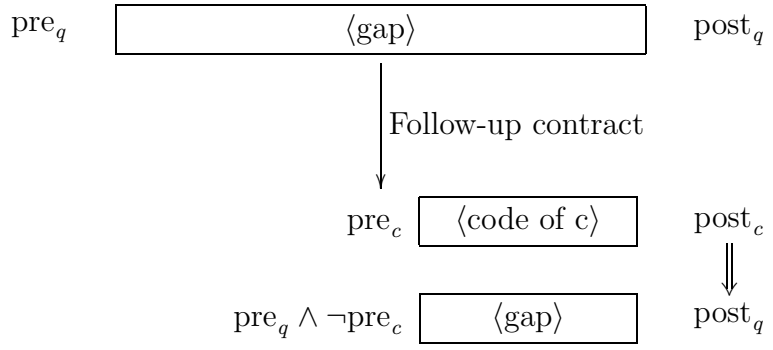


Figure 3.5: Follow-up contracts for alternative composition

However, partial solutions can also be defined by *non-determinism*. Here, two different approaches are possible. In the first, non-determinism is accepted on the domain such that a component is already considered as relevant if it works for a particular data representation. Although this can be considered as a partial domain match, the reuse effects are entirely different. Since the actual domain

is only a non-homogeneous subset of the required domain, follow-up contracts are no longer defined precisely and the systematic, black-box style composition of the retrieved components becomes impossible. Instead, the acceptable data representation(s) must be inferred, either from the proof, or, more likely, from the source code of the component which makes it a white-box approach.

In the second approach, non-determinism is accepted on the codomain: a component is already considered as relevant if at least one of its possible results is correct. Again, this compromises black-box reuse because an inspection of the source code is required which possible result is *actually* returned.

In the partial domain interpretation, the functional composition process is constrained to one direction: contracts are always closed *backwards* or *top-down*, trading the (original) postcondition against a simpler precondition. An entirely different interpretation of a partial solution can be derived if this constraint is revoked: a component is already considered as relevant if it provides an aspect of the required result. Consider for example a query for a sort function which eliminates duplicates and assume that the library contains a sort function *without* duplicate elimination and a separate duplicate elimination function which works only on sorted lists. Intuitively, both components are relevant because they can be functionally composed to satisfy the query:

$$q = \mathit{eliminate\text{-}duplicates} \circ \mathit{sort}$$

Yet, under the partial domain interpretation only the outer component *eliminate-duplicates* would be considered as relevant.

However, to arrive at an "intuitive" definition of relevance, care must be taken in the definition of what constitutes "an aspect of the required result"—just a possible completion via component composition does not suffice. For example, this would make the identity function *always* relevant:

$$q = \mathit{eliminate\text{-}duplicates} \circ \mathit{id} \circ \mathit{sort}$$

In [JD+97], L. L. Jilani et al. informally define approximate retrieval as a method which "identifies all the components that come closest to the query at hand" and define different semantic notions of proximity to be used in retrieval. However, this has the severe drawback that the relevance of a component can no longer be defined in terms of its index and the query only but also depends on the particular library (cf. also [MMM98]). Here, I concentrate on the aforementioned binary relevance notions.

3.4.2 Partial Domain Matches

An appropriate formalization of a partial solution in terms of undefinedness (i.e., partial domains) can, oddly enough, not be derived from the domain rule (cf. Section 3.3). Any match condition which only involves the preconditions cannot

ensure the suitability of the retrieved components at all and is thus bound to yield a very low precision.

However, the reasoning which leads to the definition of robust plugin-match (cf. Def. 3.3.7) can also be applied to approximate retrieval. In robust plugin-match, the component's own precondition is used as guard for the implication in the second conjunct, ensuring that the the component delivers the requested results for its own domain, if not for the query domain. Only its first conjunct, the domain rule, is used to enforce that the component's domain subsumes the query domain. If this conjunct is dropped, the relevance condition is captured exactly. Hence:

$$pre_c \wedge post_c \Rightarrow post_q \tag{3.4}$$

For the *conditional plug-in match*, this body is wrapped into a fully universal prefix. This match definition is in the literature also known as *weak post match* [Moo96, MW97b], or *semantic feature* [PBA95, Pen98].[10] The name conditional plug-in match (and similarly *conditional compatibility* [FSS98]) emphasizes the fact the components which are retrieved under this match can still be "plugged into the place of the query", albeit under the condition that the open obligations which result from their preconditions are eventually satisfied.

Definition 3.4.1 (conditional plug-in match) *For a query q and a component c,* conditional plug-in match μ_{cond} *is defined as*

$$\begin{aligned}
&\forall \vec{x}_q \forall \vec{y}_q \forall \vec{x}_{c,u} \forall \vec{x}_{c,r} \forall \vec{y}_c \cdot \\
&\quad T_{\mathrm{dom}}(\vec{x}_q, \vec{x}_{c,r}) \wedge T_{\mathrm{cod}}(\vec{y}_q, \vec{y}_c) \Rightarrow \\
&\qquad \big(pre_c(\vec{x}_c) \wedge post_c(\vec{x}_c, \vec{y}_c) \Rightarrow post_q(\vec{x}_q, \vec{y}_q)\big)
\end{aligned}$$

By construction, this match variant enables components which "almost fit" to be retrieved without respect to their actual domain. It thus emphasizes the similarities in the components' functionality and neglects their domain differences. This property makes conditional plug-in match suitable for library indexing purposes. Consider for example the query[11]

$$\begin{aligned}
&\mathit{segment\text{-}feature}\ (l : \mathit{list})\ r : \mathit{list} \\
&\mathsf{post}\ \exists\, l_1, l_2 : \mathit{list} \cdot l = l_1 \curvearrowright r \curvearrowright l_2
\end{aligned}$$

Using conditional plug-in match, both *front*-variations (i.e., *front* (cf. p. 44) and *front-total* (cf. p. 59)) match while only *front-total* matches under proper match and its derived forms.

[10] This name is based on the interpretation that all retrieved components share the property described by the query. J. Penix et al. use this interpretation for library indexing.

[11] Note that the two queries *segment-total* and *segment-feature* are semantically equivalent VDM-SL specifications because they both describe the same *total* relation. However, I use an explicit precondition *true* to denote a query for a total function while I use the variant without precondition for features.

The definition of conditional plug-in match could be further strengthened by dropping the guard, i.e., replacing pre_c by *true*. This variant, called *plug-in post* in [Moo96, MW97b], could be considered as a more robust variant of the conditional plug-in match, similar to the relation of strict plug-in match to proper match. However, this kind of robustness is defined as proper behavior over *larger* domains than explicitly stated and is thus at odds with the retrieval policy of approximate retrieval, proper behavior over *smaller* domains.

Conditional plug-in domain match obviously uses only the postcondition of the query and not the usual full *(pre, post)*-pair. However, the precondition of the query can be used to additionally restrict the extent of the domain for which the component must return the required results. pre_q plays then essentially the same guarding role as in the proper plug-in match (cf. Def. 3.3.1).

Definition 3.4.2 (partial domain match) *For a query q and a component c,* partial domain match $\mu_{partial\text{-}dom}$ *is defined as*

$$\begin{aligned}&\forall \vec{x}_q \forall \vec{y}_q \forall \vec{x}_{c,u} \forall \vec{x}_{c,r} \forall \vec{y}_c \cdot \\ &\quad T_{\mathrm{dom}}(\vec{x}_q, \vec{x}_{c,r}) \wedge T_{\mathrm{cod}}(\vec{y}_q, \vec{y}_c) \Rightarrow \\ &\qquad \big(pre_q(\vec{x}_q) \wedge pre_c(\vec{x}_c) \wedge post_c(\vec{x}_c, \vec{y}_c) \Rightarrow post_q(\vec{x}_q, \vec{y}_q)\big)\end{aligned}$$

Consider for example the component

$$\begin{aligned}&\mathit{extract\text{-}palindrome\text{-}stem}\ (l : \mathit{list})\ \ r : \mathit{list} \\ &\mathsf{pre}\ \ l = \mathit{reverse}(l) \\ &\mathsf{post}\ l = r \frown \mathit{reverse}(r) \vee \exists\, i : \mathit{item} \cdot l = r \frown [i] \frown \mathit{reverse}(r)\end{aligned}$$

which extracts the stem from a palindrome (i.e., the segment of the palindrome which occurs in original as well as in reversed form). Clearly, *extract-palindrome-stem* has the *segment-feature*-property (i.e., matches under the conditional plug-in match against the query *segment-feature*). However, if the stronger query

$$\begin{aligned}&\mathit{proper\text{-}segment}\ (l : \mathit{list})\ \ r : \mathit{list} \\ &\mathsf{pre}\ \ l \neq [\,] \\ &\mathsf{post}\ \exists\, l_1, l_2 : \mathit{list} \cdot l = l_1 \frown r \frown l_2 \wedge (l_1 \neq [\,] \vee l_2 \neq [\,])\end{aligned}$$

is used to retrieve functions which return *proper* segments, the situation becomes more complicated. Under conditional plug-in match, *extract-palindrome-stem* does no longer match because the empty list is a palindrome and thus falls within the domain of *extract-palindrome-stem* but it does not admit a proper subsegment.[12] Under proper match, *extract-palindrome-stem* does not match either, now because the domain rule does not hold: not every non-empty list is a palindrome. Hence, the only way to handle this situation adequately (and to retrieve *extract-palindrome-stem*) is to use both preconditions as assumption, as in the partial domain match.

[12]Hence, a *proper-segment-feature* without explicit precondition would be non-implementable.

However, the results of the component are only checked on the part of the domain which is common to query and component. If that part is empty, the match relation $\mu_{\text{partial-dom}}$ holds trivially, leading to "unfit" components being retrieved and thus to a loss of precision. Such empty common domains may result from contradictory preconditions, e.g., *sorted-strictly-ascending* vs. *sorted-strictly-descending*; these contradictions can in general not be prevented by the user because the components' preconditions are unknown by definition—if they were known, there would be no need for component retrieval.

The *common domain match* prevents such empty common domains by explicitly requiring the existence of at least one compatible (w.r.t. the type compatibility predicate) element for which both preconditions hold in addition to the proper match.[13]

Definition 3.4.3 (common domain match) *For a query q and a component c,* common domain match $\mu_{\textit{partial-dom}}$ *is defined as*

$$
\begin{aligned}
&\exists \vec{x}_q \exists \vec{x}_{c,u} \exists \vec{x}_{c,r} \cdot T_{\text{dom}}(\vec{x}_q, \vec{x}_{c,r}) \wedge \mathit{pre}_q(\vec{x}_q) \wedge \mathit{pre}_c(\vec{x}_c) \\
&\wedge \\
&\forall \vec{x}_q \forall \vec{y}_q \forall \vec{x}_{c,u} \forall \vec{x}_{c,r} \forall \vec{y}_c \cdot \\
&\quad T_{\text{dom}}(\vec{x}_q, \vec{x}_{c,r}) \wedge T_{\text{cod}}(\vec{y}_q, \vec{y}_c) \Rightarrow \\
&\qquad \big(\mathit{pre}_q(\vec{x}_q) \wedge \mathit{pre}_c(\vec{x}_c) \wedge \mathit{post}_c(\vec{x}_c, \vec{y}_c) \Rightarrow \mathit{post}_q(\vec{x}_q, \vec{y}_q)\big)
\end{aligned}
$$

Partial and common domain match, respectively, are particularly well suited to handle libraries which apply VDM-SL's state invariants or the similar concept of *predicative subtyping* [ROS98]. In fact, since $\mathit{pre}_{\textit{extract-palindrome-stem}}$ is just the characteristic type predicate of a subtype of *list*, the specification

extract-palindrome-stem-2 $(l : \mathit{palindrome})\ r : \mathit{list}$
pre true
post $l = r \frown \mathit{reverse}(r) \vee \exists\, i : \mathit{item} \cdot l = r \frown [i] \frown \mathit{reverse}(r)$

is equivalent to *extract-palindrome-stem*; hence, working with *extract-palindrome-stem-2* without taking the characteristic predicate of *palindrome* into account leads to unsound results.

[13] In principle, it is also possible to encode the existence of any finite number of different elements in the common domain, even though the proof task then becomes rather complicated. This degree of domain coincidence can then be used to order the retrieved components.

Chapter 4

The System NORA/HAMMR

In this thesis, I investigate deduction-based software component retrieval, starting with its theoretical foundations. But it is not only a theoretical thesis. Large parts of the actual dissertation work were devoted to building the system NORA/HAMMR which allowed an experimental evaluation of the theoretical considerations.

However, NORA/HAMMR is not only an experimental evaluation vehicle. Although still a prototype, its design was heavily influenced by practical considerations—in fact, exploring viable design alternatives was an equally important topic. This chapter describes this exploration process and its results. Section 4.1 discusses some practical aspects of deduction-based retrieval in detail while Section 4.2 describes the resulting architecture. The final Section 4.3 describes the setup for the experimental evaluation and the handling of VDM-SL.

4.1 Making Deduction-Based Retrieval Practical

The idea to employ formal reasoning in order to show an "implementation" or "refinement" relation between two specifications is central to almost all formal (e.g., algebraic or model-oriented) software development methods and can thus be traced back as far as the mid-70's. Since the mid-80's, environments have been developed which incorporated at least rudimentary deduction-based retrieval components, e.g., PARIS [KRT87] or Inscape/Inquire [Per89, PP93a], and since the early 90's, deduction-based retrieval has been investigated on its own and more thoroughly.

Yet, the results are disappointing. To the best of my knowledge, no system actually left its own lab and deduction-based retrieval is quite often perceived as an "interesting failure." This failure or, more precisely, the perception of a failure had a variety of reasons, e.g.,

- insufficient deductive power,
- inadequate calculi,
- incomprehensible notation,
- inappropriate architecture.

This perception was reinforced by a general disapproval of formal methods and a looming concern about scale-up.

However, automated deduction has made significant progress in the past few years, both in raw inference rates and in calculi, and recent ATP competitions [SS97, SS98] indicate that the "deductive bottleneck" of the earlier approaches has been widened sufficiently[1] such that a new attempt is justified.

4.1.1 User Requirements

Deduction-based retrieval is still a demanding theorem proving application but if the goal is not only to generate realistic prover benchmarks but to make retrieval practical, the technical requirements must follow (and serve) the users' requirements. Yet, even the most basic and obvious user requirement is sometimes lost:

> *"Components, not proofs!"*

That is, in contrast to, for example, program verification, the proof itself is no object of interest. Hence, the proof *process* must be hidden completely—in fact, even the presence of a theorem prover should be hidden. The main steps to achieve this are

- the choice of a comprehensible contract language,
- an intuitive user interface, and
- full automation.

The first step is necessary to gain acceptance by software engineers. Obviously, the different clausal normal form syntaxes still employed by many provers are much too rudimentary but even full first-order syntaxes, e.g., the DFG-syntax [HKW96], are still not suitable for a convenient formulation of contracts. A contract language should thus be

- *prover-independent* to facilitate smooth prover switches,

[1] Unfortunately, there is no published hard data to support this claim but current theorem provers solve formerly "difficult" problems (i.e., requiring several minutes) in some seconds [Sch98] and the total number of problems solved over the TPTP benchmark collection [SSY94] is increasing steadily [Sut98].

- *component-oriented* to facilitate the easy specification of components in contrast to plain logic formulas,
- *sufficiently expressive* and provide notations for, e.g., lists and sets,
- *extensible*, e.g., via modules or traits, to facilitate the definition of local sub-vocabularies or *custom logics*.

Most modern specification languages, e.g., Larch [GHW85, GH86], Z [BN92, Spi92], or VDM-SL [A$^+$93, Daw91, PL92] satisfy these conditions. Additionally, using such a specification language as contract language ties software development and reuse closer together which offers some pragmatic advantages. Obviously, up-front investments, e.g., teaching or component indexing, can now be amortized over two phases. More importantly, the contracts for reuse then arise automatically from the development process and do not impose any extra work on the developers. Hence, retrieving components becomes an alternative to any single development (i.e., refinement) step—reuse is built into the process from the very beginning. This does not only increase productivity as the steps become larger but also enables vertical prototyping and thus increases the confidence into the validity of the system being built.

An intuitive user interface can provide an additional layer of separation between the user and the deductive machinery. Its main purpose is to allow the user a concise and consistent, prover-independent control of the retrieval process. This control is conceptually exercised on two levels. On the global level *behavioral control parameters* affect what proof conditions are generated. This group includes parameters as for example the libraries to be searched, the query, the match conditions, and the applied filters. The behavioral control parameters are the parameters of primary interest to the end-users because they specify what should be retrieved; the user interface should thus be tailored towards them. On the local level, *performance control parameters* affect what components are retrieved. This group includes the technically more specific parameters as for example simplification methods, time limits, and lemma libraries. These parameters are only of minor interest to the end-users, unless—obviously—the performance of the retrieval tool degrades. In that case, however, the more expert *reuse administrator* is supposed to take over. Hence, they can be subordinated to the behavioral control parameters. Section 4.2.2 describes NORA/HAMMR's user interface in some more detail.

However, the most important step is the full automation of the entire proof process. Here, the emphasis is on *full* and *entire*—an unsuspicious re-user whose only intention is to locate a component cannot be expected to advise a stalled interactive prover to "`resume by induction on q`" or even to invent additional lemmas which facilitate a proof. The only exception from this user model is the reuse administrator who is in charge to maintain the library *and* the retrieval

tool. The process of reuse administration is discussed in some more detail in Section 4.2.3.

Next to finding components at all, the other basic user requirement is to find them "sufficiently fast". The question, however, is how fast is sufficiently fast. Even for medium-sized libraries, sub-second response times are still far out of reach. Fortunately, such response times are not necessary for a practical retrieval system because it offers large benefits (i.e., fully implemented, tested, and probably even verified components) and can thus settle for a more relaxed requirement:

> *"Results while-u-wait!"*

In practice, the response time of a deduction-based retrieval tool depends not only on the size of the library but also on the required recall level. The optimal trade-off between response times and recall level is determined by the reuse situation which in turn is affected by a variety of factors, e.g.,

- reuse and retrieval policies,
- number of relevant components,
- origin and granularity of the query.

In black-box reuse, retrieved components are not subject to any modification. Each of the retrieved components fully satisfies the retrieval goal, at least for exact and proper retrieval, i.e., the components become indistinguishable and the choice of any particular component becomes irrelevant. But then the actual recall level becomes also irrelevant, as long as at least *one* component is retrieved at all. Consequently, the time elapsed until the first component is retrieved is more important than the total time required to retrieve all matching components. Hence, more time can be spent on any single proof task. In white-box reuse, the system is built to take maximal advantage of the library. Retrieved components must be inspected and—possibly—modified manually before they can be reused. This requires high recall levels to ensure a sufficient supply of candidates to chose from. The maximal prover time limit for each proof task can then be derived from the relation between the inspection and modification times on one hand and the recall and precision of the answer set on the other hand. In practice, the inspection and modification times dominate this relation which allows generous time limits for the ATPs.

4.1.2 Technical Requirements

The most stringent technical requirements follow from the full automation dictum. It is not sufficient just to use an *automated* theorem prover—it is also necessary to generate the input for the automated prover automatically. However,

this involves several preprocessing steps which are usually done by "experienced experimentators."

The first of these preprocessing steps is *decustomization* which denotes the translation from the custom logic via the contract language down to the actual theorem prover input format. Since this translation usually spans a wide semantical gap, it is advisable to break it down into several substeps.

1. *Compilation into a core language*: this step takes the original, abstract user input written in the contract language (e.g., query, protocol description, or program with assertions), checks its well-formedness and translates it into a small core language which represents a domain-specific logic (e.g., many-valued logic or modal logic). It eliminates for example references to the custom logic or scoping and derived binding constructs. This step usually requires only standard compiler technology but it is the most labor intensive step because good contract languages provide by definition a variety of such constructs. For NORA/HAMMR this is discussed in some more detail in Section 4.3.3; however, such a step is not specific to deduction-based retrieval but occurs in most ATP-applications in software engineering, e.g., predicate transformers in program verification.

2. *Logic translation*: usually, the domain-specific logic used as core languages does not have adequate automatic reasoning support. Such support can be enabled by a translation or embedding into a more common logic, usually a FOL-variant which then allows to reuse existing theorem provers. Section 4.3.3 also contains a description of NORA/HAMMR's logic translation step which translates from LPF [BCJ84], the logic of partial functions underlying VDM-SL, to order-sorted FOL with equality.

3. *Task completion*: the first two decustomization steps generate a pure proof problem which needs to be completed before it can be submitted to a theorem prover. The most important completion is the addition of an axiomatization for the custom logic to the proof task which is discussed in detail in Chapter 9. Most other completion steps, e.g., selection of search control parameters, are usually more prover-specific.

4. *Task adaptation*: the different theorem provers support the order sorted FOL used as core logic to different extents (cf. Section 4.3.4); it is thus necessary to adapt the tasks to the specific provers. In many cases, the adaptation can be achieved by "wrapping" the provers into additional off-the-shelf components, e.g., for clausification or equality handling. In other cases, the adaptation can be achieved by a prover-specific task completion, e.g., adding sort axioms. The most complicated situation, however, arises when peculiarities of the prover's calculus require a major modification of the entire proof task structure; inductive theorem provers for example work better with explicitly recursive axiomatizations and tasks.

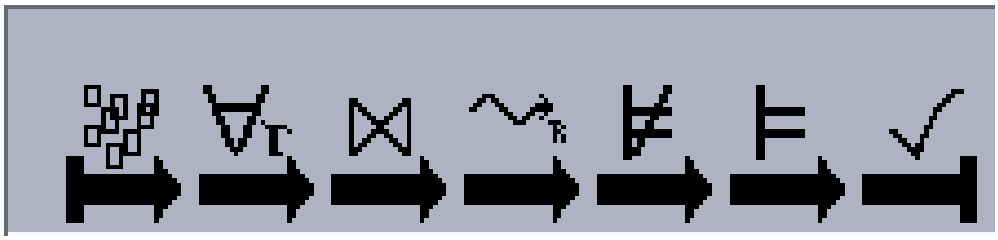


Figure 4.1: Typical filter pipeline (Detail from Figure 4.2)

In principle, the four decustomization steps are independent of each other; the later steps can be combined with different "front ends" (i.e., compilation steps) and thus be reused across different software engineering applications. In practice, however, the adaptation for a particular prover may have significant influence on the preceding steps, especially on the task completion. In NORA/HAMMR, the prover-specific task adaptation is thus more or less almost reduced to cases which can be implemented with the task completion.[2] That is, all prover specifics must be encoded into different lemma libraries and lemma selection mechanisms.

4.2 System Architecture

The requirements described in the previous section prompt a radical departure from the traditional monolithic, batch-oriented architectures. For a practical system, a flexible and open architecture which supports incremental retrieval is necessary.

4.2.1 Filter Pipeline

The central element of NORA/HAMMR's system architecture is a customizable pipeline of independent filters through which the candidates are fed. Each filter performs only a dedicated task, e.g., proof task generation, rewrite-based rejection, or final confirmation. A filter typically inspects one component at a time, makes a local decision and either rejects the component or passes it on to the next filter. Hence, components which are ruled out upstream never arrive at downstream filters. In the different filters, the components are represented in different forms. Some of the upstream filters require only database handles but the downstream filters generally pass along the entire proof task associated with a candidate. This is required to communicate local changes of the proof task (e.g., simplifications) to subsequent filters.

A typical pipeline configuration which was also used in the experiments described in the subsequent parts of the thesis is shown in Figure 4.1. This pipeline

[2] Of course, wrapping external functionality around the ATPs is supported; however, this is conceptually closer to modifying the applied prover than to adapting the proof task.

comprises six filters and a final dummy filter to inspect the final result. The first three pre-filters are required to set up the proof tasks; the first and third filter are no "filters" in the stricter sense of the word because they never reject a component.

- The *library access filter* currently just acts as a "dumb" pipeline source. More intelligent versions may incorporate indexing algorithms as described in Section 2.2
- The *signature matching filter* rules out components with incompatible calling conventions and instantiates the type compatibility predicates appropriately.
- The *join filter* builds the actual proof tasks from the query and the components' specifications, as described in Chapter 3.

These pre-filters could also have been integrated into a single pipeline source; however, individual filters fit better into the overall architecture and allow faster and more fine-grained system adaptation. The next three filters constitute the core of the semantic retrieval process and do the actual deductive work.

- The *rewrite-based rejection filter* takes the raw proof tasks mechanically generated by the join filter and applies the simplifications described in Chapter 5. Since all simplifications are sound, the filter may pass the modified proof task down in any case, even if it is not reduced to *false* (i.e., the component is not rejected). This "side effect" is of course the intended main effect of the filter. However, in order to prevent excessive formula sizes, the results of the quantifier unrolling strategy (cf. Section 5.3) are not propagated unless the tasks are reduced to *true* or *false*.
- In the *counter-model based rejection filter* the simplified proof tasks are evaluated over a specific structure. If this yields *false*, the component is rejected, otherwise the component is propagated with the original proof task.
- The *confirmation filter* uses an ATP to implement the proper specification matching step. It follows a simple "all-out match" approach, i.e., attempts to prove all incoming tasks separately and rules out all components associated with tasks the ATP fails to prove. Since the ATPs are sound, this approach guarantees a 100%-precision if the match relation is logically weaker than the relevance relation. This filter also includes all prover-specific preprocessing steps, e.g., sort encoding or lemma selection.

The final pipeline element is a simple sink. It is only required to collect and display the final result (cf. Figure 4.3).

In the current NORA/HAMMR-implementation the pipeline works only on a single processor. A coarse-grained parallelization on the task level can, however, easily be achieved because the tasks are independent of each other and because the order in which each single filter processes the incoming tasks is irrelevant. Hence, scale-up to larger component libraries can be achieved in a brute-force manner by adding more processors.

In an experimental extension of NORA/HAMMR the ILF-system [DG+97] was successfully used to distribute proof tasks over a local-area network [BF98, BFF99]. The experiments showed that the distribution overhead was insignificant which also makes competition between different provers feasible. Moreover, this prototype has also been used to experiment with proper prover cooperation which can substantially increase the recall rates (cf. the results reported in [BFF98, BFF99]).

In the current implementation the pipeline is also completely linear, i.e., allows (except for the parallelization provided by ILF) no branches through which proof tasks could be routed alternatively. This results in the typical *single point of failure* behavior mentioned earlier and increases the potential loss of recall by application of unsound rejection filters and incomplete confirmation filters. This potential can be mitigated by branching architectures implementing more complicated decision schemas as *majority vote* or *vetoing*.

4.2.2 User Interface

The recent example of model checking has shown that simple usability ("push-button-technology") is a crucial success factor, even for a formal development method as for example hardware or protocol verification. It is even more important in deduction-based retrieval where the underlying tools (i.e., provers) have not yet reached the necessary level of maturity.

NORA/HAMMR thus features a graphical user interface which completely hides the entire deductive machinery. Hence, to use NORA/HAMMR as retrieval tool, a user needs to know only the target language for signature matching and VDM-SL for specification matching.

Figure 4.2 shows the main window of the tool. It allows a simple, single-button operation in a VCR-like style. The retrieval process may be interrupted at any time, e.g., for the inspection of intermediate results, but can be resumed again, even if search (e.g., query or compatibility) or control parameters (e.g., prover or axiom selection heuristics) have been changed. The icon pad allows an easy configuration of the pipeline. To add a new filter, its class is selected via menu; a class-specific dialog window then asks for the initial values of the filter control parameters. This setup process can also be bypassed and complete pipelines can be saved to and read back from file, respectively. Search keys are divided into different categories (currently signature and contract); an additional "virtual" category allows the on-the-fly extension of the custom logic in the form

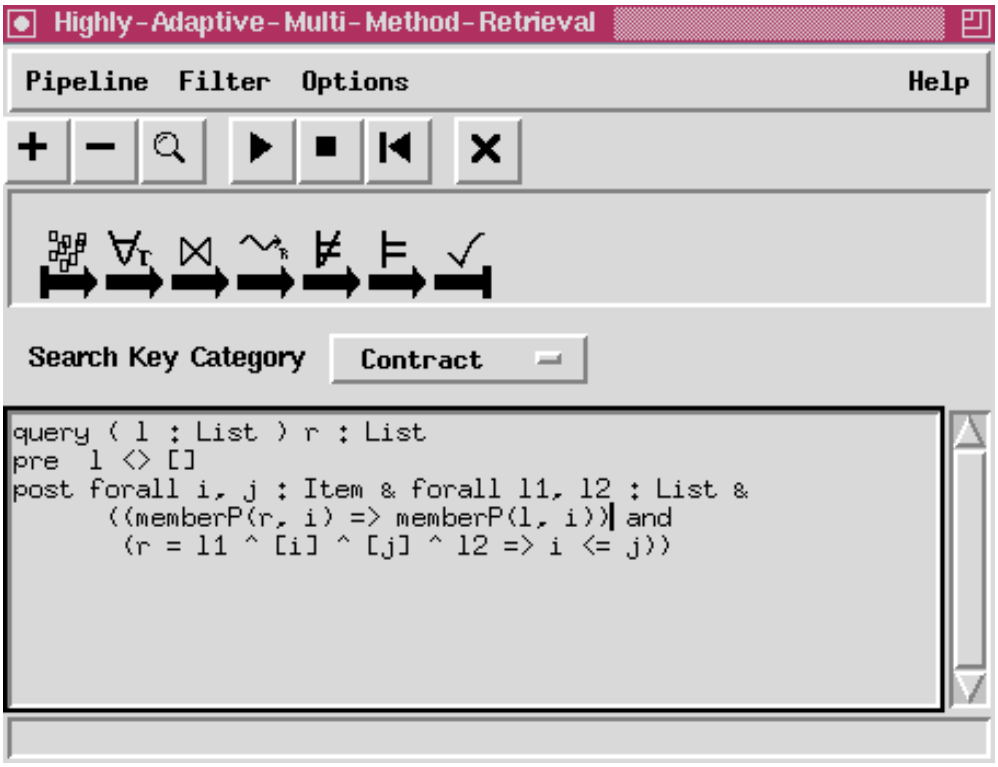


Figure 4.2: GUI: Main window

of additional VDM-SL modules which are imported into the query.

NORA/HAMMR not only presents the final result but provides direct access to the pipeline for the early inspection of intermediate results at every filter. The result inspectors grant not only further access to the retrieved components (cf. Figure 4.3) but also provide persistence. Hence, intermediate results can be used as "libraries" for further queries which supports a convenient exploration of large libraries.

4.2.3 Reuse Administration

Reuse administration is always necessary, independently of the applied reuse and retrieval approaches. Typical administration tasks include

- *quality assurance*, e.g., enforcement of coding, testing, and documentation standards,

- *library coherence maintenance*, i.e., keeping the library well-defined and maintaining a balance between library size and individual component reuse frequency,

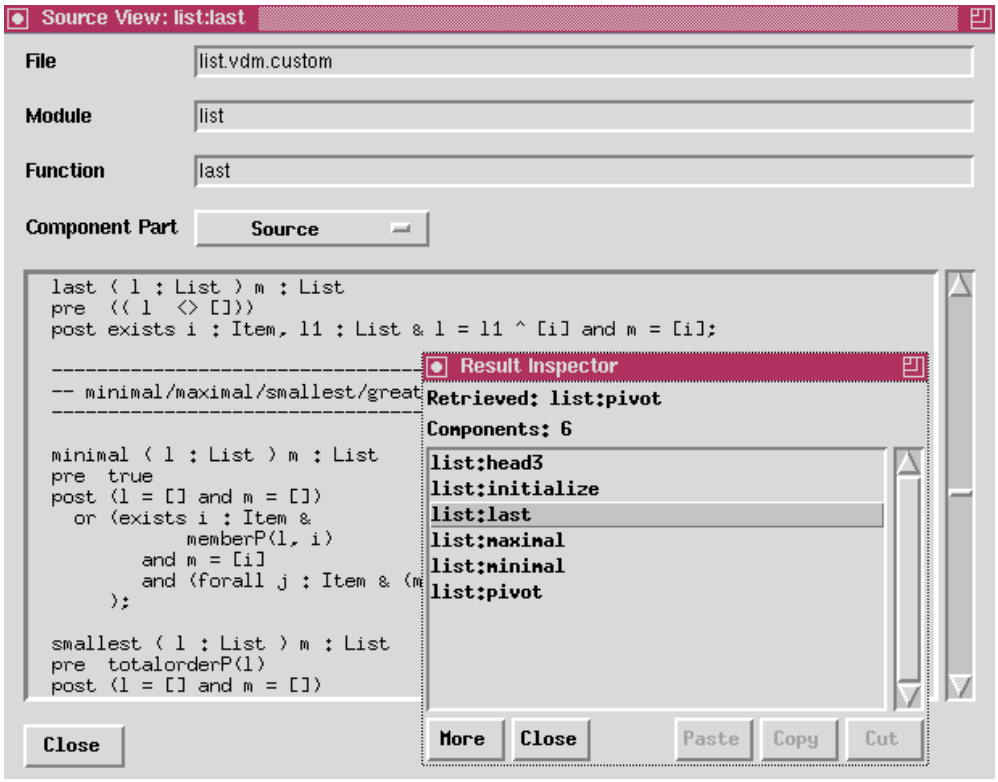


Figure 4.3: GUI: Result inspector

- *local sub-vocabulary definition* to ensure that all clients refer to the same concepts by the same name,
- *system tuning*, e.g., maintenance of stop lists.

Except for quality assurance which is largely necessary upon incorporation of a single component into the library, all administration tasks must be performed repeatedly throughout the entire life time of the library—without administration, the quality of the library degrades and it soon becomes unusable. These tasks also account for the additional up-front investments which can easily double the cost to develop a reusable component instead of a "write-once" component.

In the deduction-based case, reuse administration and in particular system tuning become even more important because the retrieval method is computationally more expensive and, unfortunately, still more brittle. In addition to the general tasks, three activities can be identified which are specific to and particularly important to deduction-based retrieval.

- *Definition of a custom logic*: in the deduction-based case, sub-vocabularies cannot be plain word lists but must be extensions of the specification language. Such an extension is called a *custom logic*. Its definition comprises a set of functions and predicates together with the respective axioms. Often,

a custom logic reflects the core functionality of the library, e.g., basic list predicates or basic date functions. Typically, it can be realized as a set of modules or traits in the specification language. This step is particularly important for a practical system because it reduces at the same time the complexity of the specifications which are visible to the user (i.e., queries) and the complexity of the emerging proof tasks.

- *Organization of a lemma library*: additional lemmas often allow much shorter proofs; if the lemmas are chosen judiciously, the provers can actually profit from them and do not get lost in the additional search space.

- *Prover performance tuning*: automated theorem provers usually provide a wide array of options and parameters which can be "tweaked" to adapt their underlying general calculi to specific tasks or domains. While there has been some progress in automatically learning successful option settings and parameter values from the tasks [Fuc95, FF98], careful monitoring of the prover performance coupled with human insight into the applied calculi is still the most common tuning approach.

In the practical experiments with NORA/HAMMR, reuse administration was not a separate activity but integrated into the entire system development. Most reuse administration efforts went into the definition of the custom logic and its accompanying lemma library.

4.3 Experimental Setup

A reliable evaluation of a retrieval system requires a large number of retrieval experiments; evaluation of different system variants requires the exact repetition of these experiments.

In the specification-based case, these repeated experiments can conveniently be generated: since component surrogate and queries coincide, components can also serve as queries. However, this assumes that the library contains—besides "real" component surrogates—also a sample of surrogates which adequately represent queries.

4.3.1 Library

For the retrieval experiments described in Chapters 7 to 9 I constructed a medium-sized library of 119 list-processing components. These can be divided into

- 30 queries with a total of 43 variants,

- 26 (proper) functions of type list to list or $list \rightarrow item$ with a total of 44 variants, and

- 19 predicates on *list* × *list* with a total of 32 variants.

However, the difference between queries and predicates is small, because both are possibly underspecified; for the above numbers, I essentially considered nondeterministic specifications which refer only to the argument or the result as queries (e.g., *insert_some*) and those which refer to both as predicates (e.g., *segment*).

Since a full signature matching was not available for the specification matching experiments, the components have been "twisted" slightly:

- Items are identified with singleton lists; hence, even *head* has the type *list* → *list*.
- The binary predicates are identified with unary nondeterministic functions; hence, *segment* also has the type *list* → *list*. As usual, this is interpreted such that the predicate *segment* is *true* on (l, m) if m is a possible result of the function call *segment*(l).

Task Generation

Following the basic idea, each component was then used as query against the entire library (including itself), using plug-in compatibility as match condition. For simplicity, I did not apply any internal library organization or semantic filtering but used the full cross match as test set. It thus comprises $119 \times 119 = 14161$ proof tasks. A manual inspection of these tasks revealed that 1838 or 13.0% of them are valid. However, the matches are not evenly distributed over the queries. The average number of matches per query is 15.45 with a standard deviation of 22.82 and a maximum of 116.

Relevance Judgments

For simplicity, and following an established tradition in deduction-based retrieval [MMM98, Pen98], I deliberately identified match condition and relevance condition. Hence, the general retrieval factors of the test library are $\overline{\gamma} = \underline{\gamma} = 0.13$ under the query-oriented and the document-oriented view, respectively. Both values coincide only by accident; in general, they may differ significantly.

As a consequence of the entire setup, $\mathrm{REL}(q)$ is never empty—it contains at least q itself. However, it is quite small quite often ($|\mathrm{REL}(q)| = 1$ for 33 queries) and also unevenly distributed ($\sigma_{\mathrm{REL}} = 0.19$). Hence, the query-oriented precision average may significantly differ from the document-oriented average. Especially for sound retrieval algorithms it is usually lower.

Caveats

Of course, this experimental setup (which has initially been published in [SF97]) has a number of possible consequences:

- Although the library deliberately contains multiple surrogates for some components (i.e., equivalent specifications), the specifications may be more uniform in style than a query sample generated by different users ("real life") would be. Hence, the generated proof tasks may be more homogeneous than in practice, but it is difficult to predict whether this is in favor of the theorem provers or not.

- Plug-in-match induces larger formulas than many other match definitions; this may in turn induce harder proof tasks.

- The difficulties induced by signature matching are not properly taken into account since the type compatibility predicate is restricted to the equality predicate. Furthermore, the short argument lists generate "smaller" problems than to be expected in real life.

- The applied VDM-SL-subset (i.e., lists, orders, equality) and the flat, many-sorted sort structure are easy compared to the full language (e.g., sets, arithmetic, user-defined datatypes with many generators as for example abstract syntax trees).

- Queries are not only checked against components but also against other queries which induces simpler proof tasks. Although this is countered by the harder proof tasks stemming from checking components against components, the net effect is a larger spread in the complexity of the tasks than in reality.

- The relevance condition may be too restrictive, yielding improper recall and precision values but again it is difficult to predict whether more relevant components lead to better results.[3]

While all the above uncertainties and simplifications must be kept in mind, the large number of generated tasks and the convenient "replay mechanism" justify the setup. It has thus also been used in the evaluation of other deduction-based retrieval systems [Pen98].

4.3.2 Custom Logics

Although VDM-SL already provides a wide variety of notations to describe list-processing components succinctly (e.g., sublist indexing), the test library contains a number of "specification idioms". These idioms can be translated into auxiliary predicates and functions which then form a *custom logic* (cf. Section 4.2.3). The custom logic needs of course to be formulated within VDM-SL in order to

[3]This is supported by the results of J. Penix [Pen98] who reports higher recall but lower precision for "relaxed match" (i.e., relaxed relevance conditions in the terminology used here).

make it accessible for user queries. The particular variant used in the retrieval experiments described here is realized as a VDM-SL-module containing a set of total, explicit, non-recursive function definitions but none of these properties is required. In this setup, however, the custom logic is a definitiorial extension of the base logic which in principle allows to eliminate the custom symbols by rewriting even if this leads to significantly more complicated proof tasks, as discussed in Sections 5.2.1 and 8.1.

The custom predicates used in the experiments can roughly be divided into three groups. The first group of predicates describes different relations between a list and a single element, e.g.,

$$memberP : list \times item \to \mathbb{B}$$

$$memberP\,(l, i) \triangleq \\ \quad \exists\, l_1, l_2 : list \cdot l = l_1 \frown [i] \frown l_2$$

The second group of predicates describes different sublist relations between two lists, e.g.,

$$prefixP : list \times list \to \mathbb{B}$$

$$prefixP\,(l, m) \triangleq \\ \quad \exists\, l_1 : list \cdot l = m \frown l_1$$

and, similarly, *suffixP* and *segmentP*. The last group of predicates captures different ordering properties of lists, e.g.,

$$totalorderedP : list \to \mathbb{B}$$

$$totalorderedP\,(l) \triangleq \\ \quad \forall\, i, j : Item, l_1, l_2 : list \cdot l = l_1 \frown [i] \frown [j] \frown l_2 \;\Rightarrow\; i \leq j$$

and, similarly, *strictorderedP* and *equalelementsP*. Other predicates in this group as for example

$$totalorderP : list \to \mathbb{B}$$

$$totalorderP\,(l) \triangleq \\ \quad \forall\, i, j : Item, l_1, l_2, l_3 : list \cdot l = l_1 \frown [i] \frown l_2 \frown [j] \frown l_3 \;\Rightarrow\; i \leq j$$

capture the fact that a list is orderable at all which is trivially true only if the entire *item* type is totally ordered.

However, the custom logic also needs to be represented in FOL in order to make it accessible to the prover. This can be done in two ways. The first approach relies on the decustomization process briefly described in the next section to translate the function definitions into axioms. The major drawback of this approach is that it does not allow for a convenient formulation of additional lemmas since only the actual function definition is translated; it is possible to "fold" lemmas as for example $\forall i \neg memberP([\,], i)$ into the definition but that soon leads to rather unwieldy specifications as for the example

$$memberP : list \times item \to \mathbb{B}$$
$$memberP\,(l, i) \triangleq$$
$$\quad l \neq [] \wedge \exists\, l_1, l_2 : list \cdot l = l_1 \frown [i] \frown l_2$$

where the lemmas clutter the original functionality.

The preferred second approach (which is also used in NORA/HAMMR) thus relies on the rule library to axiomatize the custom logic. The only slight problem here is that the reuse administrator has to keep the two variants consistent with each other.

4.3.3 Handling VDM-SL and LPF

VDM-SL is a very expressive specification language which is tailored much more towards specifying real-world applications than towards automatically proving properties of these specifications. The first decustomization steps thus deal with the translation of VDM-SL into FOL.

```
module some
  imports
    from custom-logics-list
      types item;
            list
      functions memberP
  exports
    functions some : list~→list
  functions
    some (l : list) r : list
    pre  l ≠ [ ]
    post ∃ i : item · memberP(l, i) ∧ r = [i]
end some
```

Figure 4.4: Component representation in NORA/HAMMR

NORA/HAMMR represents each component or query as a self-contained VDM-SL module (cf. Figure 4.4 for an example). The component modules are automatically extracted from the original specifications by a specialized VDM-SL front-end. The core of such a *component module* is of course the definition of the component itself (here the function *some*); however, it may also contain arbitrary

local definitions. The custom logic is referenced via the standard module import mechanism; this allows to feed the component modules into other off-the-shelf VDM-SL tools, e.g., for validation and verification purposes. In the example, the component module *some* imports the two base types *item* and *list* and the predicate *memberP* from the list custom logic. The front-end "knows" about the employed custom logic and thus resolves only the remaining "non-custom" imports. Additional information as for example time stamps, source locations, and type information is also extracted automatically and stored alongside with the component modules.

The join filter (cf. Section 4.2.1) takes two component modules and constructs a VDM-SL abstract syntax tree (AST) which represents the proof task according to the selected match condition (cf. Chapter 3), and a common symbol table which contains the appropriately renamed local definitions as well as the custom logic references. The proof task AST is then rewritten into a simpler normalized form which eliminates most of the syntactic variety VDM-SL allows. This step eliminates multiple bindings at a single quantifier or let-construct, pattern- and set-bindings, unique existential quantifiers, elsif- and cases-constructs and relativizes types with their invariants. This simplified AST is then translated into a two-valued form. This translation is based on the interpretation of LPF and its subsequent embedding into a classical (i.e., two-valued) logic given in [Mid93, JM94]; the embedding has been optimized to reduce the size of the resulting formula. The translation uses two mutually recursive functions $\langle\![\cdot]\!\rangle^{\mathsf{tt}}$ and $\langle\![\cdot]\!\rangle^{\mathsf{ff}}$, called the *truth* and *falsehood* conditions, respectively (cf. Figure 4.5).

$$
\begin{array}{rlrl}
\langle\![\neg A]\!\rangle^{\mathsf{tt}} &= \langle\![A]\!\rangle^{\mathsf{ff}} & \langle\![\neg A]\!\rangle^{\mathsf{ff}} &= \langle\![A]\!\rangle^{\mathsf{tt}} \\
\langle\![A \wedge B]\!\rangle^{\mathsf{tt}} &= \langle\![A]\!\rangle^{\mathsf{tt}} \wedge \langle\![B]\!\rangle^{\mathsf{tt}} & \langle\![A \wedge B]\!\rangle^{\mathsf{ff}} &= \langle\![A]\!\rangle^{\mathsf{ff}} \vee \langle\![B]\!\rangle^{\mathsf{ff}} \\
\langle\![A \vee B]\!\rangle^{\mathsf{tt}} &= \langle\![A]\!\rangle^{\mathsf{tt}} \vee \langle\![B]\!\rangle^{\mathsf{tt}} & \langle\![A \vee B]\!\rangle^{\mathsf{ff}} &= \langle\![A]\!\rangle^{\mathsf{ff}} \wedge \langle\![B]\!\rangle^{\mathsf{ff}} \\
\langle\![A \Rightarrow B]\!\rangle^{\mathsf{tt}} &= \langle\![A]\!\rangle^{\mathsf{ff}} \vee \langle\![B]\!\rangle^{\mathsf{tt}} & \langle\![A \Rightarrow B]\!\rangle^{\mathsf{ff}} &= \langle\![A]\!\rangle^{\mathsf{tt}} \wedge \langle\![B]\!\rangle^{\mathsf{ff}} \\
\langle\![\forall x : T \cdot A]\!\rangle^{\mathsf{tt}} &= \forall x : T \cdot \langle\![A]\!\rangle^{\mathsf{tt}} & \langle\![\forall x : T \cdot A]\!\rangle^{\mathsf{ff}} &= \exists x : T \cdot \langle\![A]\!\rangle^{\mathsf{ff}} \\
\langle\![\exists x : T \cdot A]\!\rangle^{\mathsf{tt}} &= \exists x : T \cdot \langle\![A]\!\rangle^{\mathsf{tt}} & \langle\![\exists x : T \cdot A]\!\rangle^{\mathsf{ff}} &= \forall x : T \cdot \langle\![A]\!\rangle^{\mathsf{ff}}
\end{array}
$$

Figure 4.5: Translation of LPF into FOL

Both functions are derived in a straightforward manner from the three-valued interpretation of the logical operators and quantifiers. Their intuition is that the formula A has the truth value tt in LPF (i.e., A is defined and true) iff the classical formula $\langle\![A]\!\rangle^{\mathsf{tt}}$ is provable in classical logic. The basic idea here is to modify the LPF-formula A by inserting the preconditions for the partial functions in such a way that both resulting classical formulas $\langle\![A]\!\rangle^{\mathsf{tt}}$ and $\langle\![A]\!\rangle^{\mathsf{ff}}$ become (classically) unprovable whenever a (sub-) term is possibly non-denoting. In practice, this requires a (partial) flattening of the term structures by introducing fresh variables

for subterms in a way similar to the STE-modification [Bra75]. This flattening and the precondition insertion are accomplished by a third translation function $\langle\!|t|\!\rangle^{v}$ which can intuitively be interpreted as "t is defined and evaluates to v".

$$
\begin{array}{lcl}
\langle\!|f(v_1,\ldots,v_n)|\!\rangle^{v} & = & \exists v_1:T,\ldots,v_n:T_n\cdot \langle\!|t_1|\!\rangle^{v_1}\wedge\ldots\wedge\langle\!|t_n|\!\rangle^{v_n}\wedge \\
 & & \quad pre_f(v_1,\ldots,v_n)\wedge post_f(v_1,\ldots,v_n,v) \\
\langle\!|p(v_1,\ldots,v_n)|\!\rangle^{\mathsf{tt}} & = & \exists v_1:T,\ldots,v_n:T_n\cdot \langle\!|t_1|\!\rangle^{v_1}\wedge\ldots\wedge\langle\!|t_n|\!\rangle^{v_n}\wedge p(v_1,\ldots,v_n) \\
\langle\!|p(v_1,\ldots,v_n)|\!\rangle^{\mathsf{ff}} & = & \exists v_1:T,\ldots,v_n:T_n\cdot \langle\!|t_1|\!\rangle^{v_1}\wedge\ldots\wedge\langle\!|t_n|\!\rangle^{v_n}\wedge \neg p(v_1,\ldots,v_n)
\end{array}
$$

Figure 4.6: Translation of LPF into FOL: precondition insertion

Figure 4.6 shows for an implicitly defined partial function f and a predicate p,[4] how $\langle\!|\cdot|\!\rangle^{v}$ and $\langle\!|\cdot|\!\rangle^{\mathsf{tt}}/\langle\!|\cdot|\!\rangle^{\mathsf{ff}}$ interact under the assumption of deterministic specifications (i.e., no loose constructs as for example let-be-such-that).[5] However, a simple-minded flattening blows up the size of the formula substantially, as the example translation in Figure 4.7 shows.

$$
\begin{array}{ll}
 & \langle\!|\forall l:list\cdot l=[hd\ l]\curvearrowright tl\ l|\!\rangle^{\mathsf{tt}} \\
\equiv & \forall l:list\cdot\exists v_1:list\cdot l=v_1\wedge\langle\!|[hd\ l]\curvearrowright tl\ l|\!\rangle^{v_1} \\
\equiv & \forall l:list\cdot\exists v_1,v_2,v_3:list\cdot l=v_1\wedge true\wedge v_1=v_2\curvearrowright v_3\wedge\langle\!|[hd\ l]|\!\rangle^{v_2}\wedge\langle\!|tl\ l|\!\rangle^{v_3} \\
\equiv & \forall l:list\cdot\exists v_1,v_2,v_3,v_5:list,v_4:item\cdot \\
 & \quad l=v_1\wedge true\wedge v_1=v_2\curvearrowright v_3\wedge true\wedge v_2=[v_4]\wedge\langle\!|hd\ l|\!\rangle^{v_4}\wedge v_5\neq[\,]\wedge \\
 & \quad v_3=tl\ v_5\wedge\langle\!|l|\!\rangle^{v_5} \\
\equiv & \forall l:list\cdot\exists v_1,v_2,v_3,v_5,v_6:list,v_4:item\cdot \\
 & \quad l=v_1\wedge true\wedge v_1=v_2\curvearrowright v_3\wedge true\wedge v_2=[v_4]\wedge v_6\neq[\,]\wedge v_4=hd\ v_6\wedge \\
 & \quad \langle\!|l|\!\rangle^{v_6}\wedge v_5\neq[\,]\wedge v_3=tl\ v_5\wedge l=v_5 \\
\equiv & \forall l:list\cdot\exists v_1,v_2,v_3,v_5,v_6:list,v_4:item\cdot \\
 & \quad l=v_1\wedge true\wedge v_1=v_2\curvearrowright v_3\wedge true\wedge v_2=[v_4]\wedge v_6\neq[\,]\wedge v_4=hd\ v_6\wedge \\
 & \quad l=v_6\wedge v_5\neq[\,]\wedge v_3=tl\ v_5\wedge l=v_5
\end{array}
$$

Figure 4.7: Translation of LPF into FOL: example

This blow-up can be mitigated if $\langle\!|\cdot|\!\rangle^{v}$ is modified to flatten only subterms

[4] For total functions f, pre_f is $true$; for explicit functions and operators of the custom logic, $post_f(v_1,\ldots,v_n,v)$ can be replaced by $f(v_1,\ldots,v_n)=v$.

[5] For loose specifications,VDM-SL's semantics requires that the predicate holds or does not hold, respectively, for all possible assignments; this substantially increases the size of the resulting formulas (cf. [Mid93] for details).

which are argument of a partial function. Moreover, since free and bound variables are always denoting in LPF (i.e., assignments and quantifiers range over elements of the domain and not over terms), flattening can be restricted to non-variable subterms. The modified translation function yields

$$\langle\!\langle \forall l : list \cdot l = [hd\ l] \frown tl\ l \rangle\!\rangle^{\mathsf{tt}} \equiv \forall l : list \cdot l \neq [\,] \wedge l \neg [\,] \wedge l = [hd\ l] \frown tl\ l$$

i.e., introduces no additional existentially quantified variables and only a single redundant conjunct which is easily eliminated during proof task simplification (cf. Chapter 5).

4.3.4 Applied Systems

The simplifier is described in more detail in Chapter 5. The other systems are essentially used as off-the-shelf components and were run in autonomous mode or a fixed setting provided by their developers, unless explicitly stated otherwise. All systems are public domain, actively maintained and designed for high performance. In particular, each of the theorem provers ranked under the "top five" in recent CADE ATP system competitions [SS97, SS98].

OTTER

OTTER (Organized Techniques for Theorem proving and Effective Research) [McC94b, MW97a] is an automated theorem prover for first-order logic with equality based on the resolution calculus. It implements several inference rules including binary resolution and paramodulation, hyperresolution, and UR-resolution, and a variety of reduction techniques including factoring, forward and back subsumption and demodulation, respectively, orderings and weights, and Knuth-Bendix completion. OTTER's main inference loop is based on the given-clause-algorithm which can be considered as a simple implementation of the set-of-support-strategy. Besides a huge number of individual parameters to control the proof search, OTTER also offers two different autonomous modes in which it selects a parameter setting after a simple syntactic analysis of the proof task.

For the experiments, I used OTTER 3.0.5. The proof tasks were given in first-order form; clausification was done by OTTER's built-in routine. Sorts were encoded by terms, following the approaches described in [Dah96, Mel88, RM93, SW89]; for each sort structure, the optimal encoding has been selected. In the usual many-sorted case, sorts are thus replaced by injection functions. The reflexivity of equality was explicitly added to the axioms. The entire task (i.e., axioms and negated conjecture) was put into the usable-list, no demodulators or term orderings were given explicitly. However, both autonomous modes reorganized the original arrangement.

GANDALF

GANDALF [Tam97a] is an automated theorem prover for first-order logic with equality based on the resolution calculus. As OTTER, it implements several inference rules including unit and binary resolution, hyperresolution, and paramodulation, and a variety of reduction techniques including forward and back subsumption and demodulation, respectively, orderings and weights, and tautology elimination. GANDALF's main inference loop is also based on the given-clause-algorithm; however, it periodically selects the first clause in the set-of-support-queue instead of the lightest clause, thus implementing a combination of best-first and breadth-first search. As OTTER, GANDALF also offers not only a variety of individual parameters to control the proof search but also an autonomous mode. However, it works slightly differently. Here, the proof task is also analyzed syntactically but, instead of a single parameter setting, GANDALF selects several settings and strategies together with a fraction of the total time which is available for the proof attempt. The variants are then ordered and run one after the other. If the time slice (or memory) allotted for a variant runs out, GANDALF switches to a special "end-game mode" in which certain inferences get higher priority. Intermediate results (e.g., unit clauses) are carried over from one variant to the next. As a consequence of this time-sharing scheme, results obtained by runs with longer timeouts cannot be trimmed down to shorter timeouts but the experiments must be repeated with the actual timeouts.

For the experiments, I used GANDALF c-1.0d. The proof tasks were given in first-order form; clausification was done by FLOTTER [NRW98], the clausifier provided by the SPASS-system. The remaining setup (i.e., sort and equality handling and task layout) was the same as for OTTER.

SPASS

SPASS (Synergetic Prover Augmenting Superposition with Sorts) [WGR96, Wei97] is an automated theorem prover for first-order logic with equality and sorts based on the superposition calculus [BG94, GMW97].

For the experiments, I used SPASS 0.92. The proof tasks were given in first-order form; clausification was done by FLOTTER. Sorts were usually relativized by unary predicates, following the standard approach [Obe62, Obe89]; all sort predicates were explicitly tagged as such but beyond that, no other unary predicates were tagged. The relativation requires a number of additional axioms to represent some properties explicitly which are implicitly assumed in (and hard-coded into) the term encoding approaches:

- subsort relations,
- sort inhabitation, and
- signature information for the function symbols.

These axioms are generated automatically by the NORA/HAMMR-system (cf. Section 9.3) and added to the proof task.

SETHEO

SETHEO (Sequential Theorem Prover) [LS+92, MI+97] is an automated theorem prover for pure first-order logic (i.e., without equality and sorts) based on the model elimination calculus [Lov68]. However, SETHEO extends the base calculus by a variety of inference and control rules in order to increase the its efficiency. The most important extensions are folding which allows proven subgoals to be used as lemmas "on the fly" and redundancy elimination using regularity, tautology, or subsumption criteria. Equality is treated either axiomatically by adding (the corresponding instances of) Birkhoff's equality axioms and schemes [Bir35] or by Brand's STE-modification [Bra75]. SETHEO's implementation follows the PTTP-technology [Sti88, Sti92]; it requires a transformation of the clauses into sets of contrapositives.

For the experiments, I used a "customized" variant which builds on top of SETHEO V3.3. This variant also encodes sorts as terms using the optimal representation for any given sort structure but relies on a different implementation of the term encoding technique (which is part of the PROTEIN-distribution [BF94]) than OTTER and GANDALF. Equality is handled axiomatically but with optimizations for the sort representation: the congruence axioms are generated in sorted form and no axioms are generated for the function symbols introduced by the sort encoding.

Method of Measurement

I did not use the GUI for the experiments because it makes an accurate timing for the single tasks almost impossible. Instead, I used a batch version which bypasses the GUI, and generates the respective variant of the proof tasks and writes them into files. These files were then—again in a batch-like fashion—processed by the provers.

All experiments were run on Sun ULTRA 1/170 workstations with at least 128 MB RAM running SunOS 5.5 or later. Run times were extracted from the protocol files generated by the respective provers; no claim is made about the accuracy of these timings. Due to the excessive number of proof tasks averaging over multiple prover runs was impossible.

Unless otherwise stated, all times refer only to the CPU-time spent by the prover and exclude preprocessing and "gluing" steps, e.g.:

- I/O,
- task generation and simplification,
- sort and equality encodings, and

- clausification.

These steps can add a significant overhead, especially in SETHEO's case.

Competition experiments between different variants or provers were not actually run on parallel processors but only simulated afterwards, using the log files of the involved variants: for each task the best reported individual timing was selected as timing under the competition. However, this systematically accumulates small timing differences (e.g., due to measurement inaccuracies or slight load differences between the different runs) and thus slightly biases competition timing.

Chapter 5

Simplification-based Rejection

Unlike the problems in theorem proving benchmark collections as the TPTP [SSY94] the proof tasks emerging in deduction-based retrieval still contain significant redundancy, e.g., the propositional constants *true* and *false* which may be part of the original queries. Hence, proof task simplification is always necessary as a complexity reduction technique before the actual prover is invoked.

However, simplification can also be used as a low-cost rejection filter: if a proof task $\mathcal{T}[q, c]$ can be simplified to *false*, the associated component c may be rejected. Obviously, this rejection filter is recall-preserving as long as the applied simplifications are sound.

Simplification can apply arbitrarily complex procedures, e.g., arithmetic equation solvers or decision procedures. In NORA/HAMMR, however, simplification is currently based on term rewriting only. To achieve efficiency, NORA/HAMMR provides a set of functions which implement "generic" rewrite rules (e.g., distributivity of one operator over another) modulo associativity and commutativity through direct term manipulations and without explicitly constructing substitutions. These functions are not only used to implement the simplifications of the core logic which I describe in Section 5.1 but also for the additional domain-specific simplifications (cf. Section 5.2). For these, the lemma library is analyzed for instances of the generic rules. In sections 5.3 and 5.4 I show how these can be turned into an efficient rejection filter; the results of this filter will be presented in Section 5.5.

Rewrite-based simplification has the big advantage over rewrite-based *theorem proving* that the applied rewrite system $\mathcal{R}$ need not be convergent: termination (which is necessary in practice) can—by virtue of an arbitrary control strategy—be forced at the expense of confluence because subsequent filters can still deal with multiple normal forms. Hence, completion of $\mathcal{R}$ is not necessary in practice.

5.1 Core Logic Simplifications

The core logic simplifications deal with the operators which are common to all proof tasks, e.g., propositional constants, equality, or quantifiers. Since the preceding preprocessing step translated the proof-tasks (provability-preserving) into sorted FOL with equality, the usual rules can be applied, even if they are not valid for LPF, as for example the law of the excluded middle. In LPF

$$\forall l : List \cdot hd\ l = [\,] \vee \neg\,(hd\ l = [\,]) \qquad (*)$$

can not be simplified easily (and actually has the truth value *undefined*). However, (*) is translated into the FOL-formula

$$\forall l : List \cdot \exists l' : List \cdot l = l' \wedge l' \neq [\,] \wedge (hd\ l' = [\,] \vee \neg\,(hd\ l' = [\,])) \qquad (**)$$

and here the law of the excluded middle can safely be applied to the right conjunct because the left conjunct still captures the definedness constraint which prevented the simplification in (*).[1]

None of the following rules is particularly deep or "tricky". However, the presentation uses some of the basic concepts of term rewrite systems. General introductions into that field are for example [Klo92, DJ90, BN98]. The main purpose of this section is to document the "practical engineering" of a rewrite system with a specific application in mind.

Propositional Constants

For the propositional constants only the usual rules apply:

$$\begin{aligned} true \wedge x &\rightsquigarrow x & \qquad false \wedge x &\rightsquigarrow false \\ true \vee x &\rightsquigarrow true & \qquad false \vee x &\rightsquigarrow x \end{aligned} \qquad (5.1)$$

Since $\rightsquigarrow$ rewrites modulo associativity and commutativity, I present all rules in a "one-sided" version only.

Negation, Conjunction, and Disjunction

As explained above, the rules for the usual negation apply, i.e.,

$$\neg\, true \rightsquigarrow false \qquad \neg\, false \rightsquigarrow true \qquad (5.2)$$

$$x \wedge \neg\, x \rightsquigarrow false \qquad x \vee \neg\, x \rightsquigarrow true \qquad (5.3)$$

$$\neg\, \neg\, x \rightsquigarrow x \qquad (5.4)$$

[1] In the following section I show how the resulting formula $\forall l : List \cdot \exists : l' : List \cdot l = l' \wedge l' \neq [\,]$ can further be simplified to *false*.

As usual, DeMorgan's laws are oriented such that a negation normal form is achieved.

$$\neg(x \wedge y) \rightsquigarrow \neg x \vee \neg y \qquad \neg(x \vee y) \rightsquigarrow \neg x \wedge \neg y \tag{5.5}$$

For conjunction and disjunction we have (besides associativity and commutativity) the idempotency laws

$$x \wedge x \rightsquigarrow x \qquad x \vee x \rightsquigarrow x \tag{5.6}$$

but also the two troublesome distributivity laws

$$x \vee (y \wedge z) \equiv (x \vee y) \wedge (x \vee z) \tag{†}$$
$$x \wedge (y \vee z) \equiv (x \wedge y) \vee (x \wedge z) \tag{‡}$$

which prevent any term rewriting system for Boolean algebras (BA) from being convergent [SA91]. Since most theorem provers work on clausal normal forms, it seems to be advisable to orient (†) from left to right and (‡) from right to left or to drop it altogether. However, since most (but not all) provers also work in a refutational style and negate the conjecture before they start their clausification procedure this orientation may be counterproductive. Moreover, the choice of either (†) or (‡) as basic distributive law may result in a loss of simplification potential, depending on the internal structure of the proof tasks. In particular, the tasks are universally quantified on the outermost level such that the choice of (‡) prevents further reductions of the quantifier scopes. Consequently, NORA/HAMMR uses both variants but in different contexts, i.e., in two different rewrite systems.

$$x \vee (y \wedge z) \rightsquigarrow (x \vee y) \wedge (x \vee z) \tag{5.7}$$
$$x \wedge (y \vee z) \rightsquigarrow (x \wedge y) \vee (x \wedge z) \tag{5.7′}$$

(5.7) is used for simplification purposes, while (5.7′) is essentially used to generate the actual tasks which are fed into the provers. Section 5.5 contains the experimental data which justifies this approach.

To compensate for the resulting loss of simplification potential, the usual adjunctive laws

$$x \wedge (x \vee y) \rightsquigarrow x \qquad x \vee (x \wedge y) \rightsquigarrow x \tag{5.8}$$

and their negated counterparts

$$\begin{aligned} x \wedge (\neg x \vee y) &\rightsquigarrow x \wedge y & x \vee (\neg x \wedge y) &\rightsquigarrow x \vee y \\ \neg x \wedge (x \vee y) &\rightsquigarrow \neg x \wedge y & \neg x \vee (x \wedge y) &\rightsquigarrow \neg x \vee y \end{aligned} \tag{5.9}$$

proved to be valuable. Moreover, in practice the respective disjunctive variants often also preempt the remaining expensive distributivity law.

It is obvious, that the rules (5.1) to (5.9) are correct w.r.t. $=_{BA}$ (i.e., the equational theory of Boolean algebras) and that the two corresponding rewrite relations $\rightsquigarrow$ terminate and are strongly confluent. However, as to be expected from the results in [SA91], the respective $\mathcal{R}$-normal forms are not "minimal" as for example the $\mathcal{R}_{CNF}$-normal form

$$(x \vee y) \wedge (x \vee \neg y) \wedge (\neg x \vee z) \wedge (\neg x \vee \neg z)$$

is *BA*-equal to *false*. But in practice, $\rightsquigarrow$ is strong enough and giving up additional rules as

$$(x \vee y) \wedge (\neg x \vee y) \rightsquigarrow y$$

saves a large number of expensive equality tests modulo associativity and commutativity.

An alternative to the application of $\mathcal{R}$ would be to use J. Hsiang's [Hsi85] rewrite system for Boolean rings which remains convergent even if the classical disjunction with the rule

$$x \vee y \rightsquigarrow x * y + x + y$$

is added. However, this causes an exponential blow-up of the formula size. Even worse, if the ring normal forms are translated back into *BA*-terms (which is necessary because theorem provers cannot deal with the ring addition or *exclusive or* directly), a second exponential blow-up occurs and many $\mathcal{R}$-reducible forms are re-introduced.

Quantifiers

Rewrite rules for quantifiers (and other similar binding constructs) need to take the bound variables into account. Although explicit substitution calculi [AC$^+$91] have become fashionable for such formalizations, I stick to the "traditional" meta-level formalization using implicit eigenvariable conditions.

Some equivalences for the quantifiers as for example

$$\forall x \cdot \mathcal{F}[x] \wedge \mathcal{G}[x] \equiv (\forall x \cdot \mathcal{F}[x]) \wedge (\forall x \cdot \mathcal{G}[x]) \qquad (\S)$$

induce similar termination problems as the distributive laws (†) and (‡) above. But in this case, the choice of an orientation is not so obvious because both alternatives have mutually exclusive advantages:

- Orientation from left to right (i.e., anti-prenex form) allows a quantifier elimination using the techniques described in the next section.
- Orientation from right to left (i.e., prenex form) allows more simplifications in the larger bodies and ultimately produces better clause normal forms due to the smaller number of different emerging skolem functions.

I follow the usual solution in simplification (cf. e.g., [Käu88]) and use both orientations but in different phases. Rules (5.10) to (5.12) are used for the anti-prenex phase, while their reverse orientations (except for rule (5.12)) are used in the prenex phase.

Finally, since the core logic is sorted, the standard rules must be slightly modified to take the sorts into account. While this is straightforward for the many-sorted case, true subsorts allow for more elaborated variants. For example, the right-to-left orientation of (§) may become

$$(\forall x : T_1 \cdot \mathcal{F}[x]) \wedge (\forall x : T_2 \cdot \mathcal{G}[x]) \rightsquigarrow \forall x : T \cdot (p_{T_1}(x) \Rightarrow \mathcal{F}[x]) \wedge (p_{T_2}(x) \Rightarrow \mathcal{G}[x])$$

for $T_1, T_2 < T$ and type relativation (w.r.t. to T) predicates p_{T_1}, p_{T_2}. However, since this defeats the purpose of sorted quantifiers by expanding the body of the quantifier and the range of the bound variables, I use only the many-sorted simplifications. However, some of them involve a "relativation residuum" to capture empty domains adequately. In such cases I use the notation "$\in T$" as type relativation predicate for T.

In the unsorted case or if T is a priori known to be inhabited (e.g., for generated datatypes, cf. Section 5.2.2), two rules suffice to remove conjuncts and disjuncts, respectively, which are alien from quantifier scopes. In the general sorted case, however, the following set is better suited because it minimizes the number of emerging relativation residua.

$$\begin{aligned} \forall x : T \cdot \mathcal{F}[x] \wedge \mathcal{G}[x] &\rightsquigarrow (\forall x : T \cdot \mathcal{F}[x]) \wedge (\forall x : T \cdot \mathcal{G}[x]) \\ \exists x : T \cdot \mathcal{F}[x] \vee \mathcal{G}[x] &\rightsquigarrow (\exists x : T \cdot \mathcal{F}[x]) \vee (\exists x : T \cdot \mathcal{G}[x]) \end{aligned} \tag{5.10}$$

$$Qx : T \cdot \mathcal{F}[x] \oplus \mathcal{G} \overset{x \notin v(\mathcal{G})}{\rightsquigarrow} (Qx : T \cdot \mathcal{F}[x]) \oplus \mathcal{G} \tag{5.11}$$

$$\begin{aligned} \forall x : T \cdot \mathcal{G} &\overset{x \notin v(\mathcal{G})}{\rightsquigarrow} \mathcal{G} \vee \neg \exists x \cdot x \in T \\ \exists x : T \cdot \mathcal{G} &\overset{x \notin v(\mathcal{G})}{\rightsquigarrow} \mathcal{G} \wedge \exists x \cdot x \in T \end{aligned} \tag{5.12}$$

The symbol $\oplus$ in rule (5.11) stands for either $\wedge$ or $\vee$. The rewrite machine only has to check the eigenvariable conditions attached to rules (5.11) and (5.12) before they are applied; after that, the bound variables are handled properly by ordinary rewriting.

Equality

Due to the specification style and the construction of the proof tasks (cf. Chapter 3) equality literals make up the largest part of all literals. Efficient equality handling is thus necessary already during the simplification phase. As usual, the main problems are to enforce termination and to balance the simplification effects against the efforts. Both problems can easily be solved if only such equalities are

used for rewriting where at least one side is a single variable occurring free in the remaining term:

$$x = t \wedge \mathcal{F}[x] \overset{x \notin v(t)}{\rightsquigarrow} x = t \wedge \mathcal{F}[t/x] \tag{5.13}$$

$$x \neq t \vee \mathcal{F}[x] \overset{x \notin v(t)}{\rightsquigarrow} x \neq t \vee \mathcal{F}[t/x] \tag{5.14}$$

The negated variant (5.14) is in practice very useful because the type compatibility predicates often boil down to single equations and the LPF-to-FOL-translation then exactly produces the left-hand sides of (5.14).

Among G. Birkhoff's axioms for equality [Bir35], only reflexivity gives rise to an additional rewrite rule

$$x = x \rightsquigarrow \mathit{true} \tag{5.15}$$

since commutativity and substitutivity are built into the term representation and rewrite mechanism, respectively, and transitivity is preempted by (5.13).

A number of rules take care of situations in which the equality literal is the only literal in the scope of a quantifier and the bound variable occurs one one side of the equation. Although this situation looks very restricted, even contrived, at first glance, these rules proved to be very valuable in practice because they capture common situations which are consequences of the specification style, the LPF-to-FOL-translation, and the quantifier scope reduction and variable elimination rules above.

The *witness rule* eliminates intermediate existentially quantified variables which are introduced by the LPF-to-FOL-translation ([JM94]; cf. also Section 4.3.3) and by VDM-SL's *let*-construct. Consider for example again the formula from p. 92:

$$\forall l : \mathit{List} \cdot \exists l' : \mathit{List} \cdot l = l' \wedge l' \neq [\,] \wedge (\mathit{hd}\ l' = [\,] \vee \neg\,(\mathit{hd}\ l' = [\,]))$$

After application of the variable substitution (5.13) and scope reduction (5.10–5.12) rules, this simplifies to

$$\forall l : \mathit{List} \cdot (\mathit{hd}\ l = [\,] \vee \neg\,(\mathit{hd}\ l = [\,])) \wedge \exists l' : \mathit{List} \cdot l = l' \wedge l \neq [\,]$$

in which the witness rule can be applied.

Definition 5.1.1 (witness rule) *The* witness rule *for any type T is*

$$\exists x : T \cdot x = t \overset{x \notin v(t)}{\rightsquigarrow} t \in T \tag{5.16}$$

The soundness of the witness rule is an immediate consequence of the semantics of FOL with equality.

Lemma 5.1.2 *The witness rule is sound for any type T.*
Proof: *$\exists x{:}T{\cdot}x = t$ holds iff the domain of T contains (under any interpretation) an element t' such that $(x = t)[x/t']$ holds, which reduces to $t' = t$ since $x \notin v(t)$; at the same time, the diagonal interpretation of the equality predicate requires t' to denote the same element as t. Hence, if t also denotes an element in the domain of T (i.e., $t \in T$ holds), t' can trivially been found by setting it to t; if $t \notin T$, no other element in the domain of T can be identical to the denotation of t' and thus $\exists x : T \cdot x = t$ cannot hold either.* $\triangle$

The residuum $t \in T$ must be checked dynamically, similarly to "runtime typechecking" in some programming languages. For generated datatypes T (cf. also Section 5.2.2), this overhead can be eliminated because the witness rule can then be strengthened to

$$\exists x : T \cdot x = t \overset{x \notin v(t)}{\rightsquigarrow} true$$

provided that the original formula is well-typed under a homogeneous equality (i.e., of type $\forall\alpha \cdot \alpha \times \alpha \to \mathbb{B}$) and does not contain occurrences of partial functions—under these assumptions the type relativation predicate $\in T$ is trivially true.[2] Fortunately, the latter constraint is not important in practice because the LPF-to-FOL-translation just isolates all applications of partial functions from their respective arguments such that they appear only in minimal scopes.[3]

Since the proof tasks do not contain uninterpreted constants and function symbols, non-variable equations (i.e., neither side is a single variable) need not to be used for rewriting as they are likely to be reduced by the domain-specific simplifications described in the next section.

5.2 Domain-Specific Simplifications

The core logic rules are not sufficient for simplification: to achieve significant reductive effects, more domain-specific rules are required. These rules can either be extracted form the lemma library (cf. Section 5.2.1) in a very straightforward manner or even be generated automatically if some meta-level information is provided by the reuse administrator. Section 5.2.2 describes this process for (freely) generated datatypes.

[2]More precisely, it is trivially true only for ground terms, but it also holds for non-ground terms because quantifiers only range over denoting terms.

[3]In effect, the witness rule undoes (together with the core logic and equality rules) to a large extent the blow-up caused by the LPF-to-FOL-translation. For properly stated formulas (i.e., formulas with truth value *true* or *false*), the formula size does usually not change.

5.2.1 Extracting Simplifications from a Lemma Library

The fully automatic extraction of a "useful" set of domain-specific simplification rules is a considerable engineering task. The main problem is to ensure termination of the extracted rewrite system. In NORA/HAMMR, I refrain from constructing a proper termination ordering. Instead, I assume that the termination ordering is implicitly given by the reuse administrator, i.e., equivalences and equalities are always oriented from left to right.

Equivalences and Equalities

Equivalences can obviously be interpreted as equalities on the datatype of truth values and thus be handled by the rewrite machinery in the same way as equalities. However, they need not be as obvious as equalities because they may be split up into two separate implications which are "scattered" throughout the lemma library. The most common case is that one implication is defined in a core theory at the lower hierarchy levels (or even in the meta-logic) while the reverse implication is defined further up in the hierarchy. In the ideal case, the equivalences can be recovered by searching the lemma library for these two implications. In less ideal cases, however, the second implication is not formulated explicitly but can be inferred (easily). This is typically the case with injectivity axioms,, e.g., for the list constructor *cons*:

$$\forall i, i' : item \cdot \forall l, l' : list \cdot cons(i, l) = cons(i', l') \Rightarrow i = i' \wedge l = l'$$

Here, the reverse implication need not be axiomatized explicitly because it is an instance of the congruence axioms schema for equality. Finding the reverse implication, however, may of course become arbitrarily complex and generally requires theorem proving capabilities.

Unit Clauses

Fully universally quantified unit clauses (i.e., axioms or lemmas consisting of a universal prefix followed by a single literal) can easily be rewritten as equivalences and then be handled as described above; the right-hand side of the equivalence is just the polarity of the literal (i.e., *true* for positive literals, *false* for negative literals). Non-universal unit clauses can obviously not be converted into (ordinary) rewrite rules because the existentially quantified variables may not be instantiated arbitrarily during rewriting. For example, it is obviously wrong to extract a rewrite rule $segmentP(l', l) \rightsquigarrow true$ from the lemma $\forall l : list \cdot \exists l' : list \cdot segmentP(l', l)$. However, non-universal unit clauses can still be used for simplification if they are not interpreted as rules for the top-level functor of the literal but as rules for the existential quantifier. Hence, the lemma gives rise to the correct rule $\exists l' : list \cdot segmentP(l', l) \rightsquigarrow true$ which can be used to simplify

$\forall l : list \cdot (\exists l' : list \cdot segmentP(l', l)) \Rightarrow l \neq [\,]$ into $\forall l : list \cdot true \Rightarrow l \neq [\,]$ and ultimately into *false* using the quantifier splitting approach described in Section 5.3. Non-unit clauses can sensibly be converted only into conditional rewrite rules (if at all). However, since these do not allow a simple and fast execution, which was the original motivation for rewrite-based simplification, non-unit clauses are not used for simplification purposes by NORA/HAMMR.

5.2.2 Handling Generated Datatypes

Inductively defined or (freely) generated datatypes abound in software specifications (e.g., lists, numbers, or sets) and thus also in deduction-based retrieval. Fortunately, their regular structure allows significant simplifications.

Inductive definedness is a higher-order property. This becomes apparent in the following definitions. They generalize the approach of J. Harrison [Har95], which is based on least fixpoints (*lfp*), to the many-sorted case.[4]

Definition 5.2.1 (inductively defined set) *A set $T \subseteq X$ is called* inductively defined by the *generator predicate $\mathcal{P}$ iff $\forall x \cdot \mathcal{P}(T, x) \Rightarrow x \in T$ holds, the function $f(S) = \{x \mid \mathcal{P}(S, x)\}$ is monotone on $2^X \rightarrow 2^X$ and $T = lfp(f)$.*

For the special case of *generated datatypes*, $\mathcal{P}$ makes only very limited assumptions such that it can even be derived automatically from a signature. Moreover, the function f then is monotone by construction. However, the definitions are complicated by the many-sorted setup, i.e., by the fact that the arguments of a generator function may be of a sort other than the one being defined inductively.

Definition 5.2.2 (generated datatype) *Let π_T be defined as*

$$\pi_T(S, U) = \begin{cases} S & U = T \\ U & else \end{cases}$$

A type T is generated *by the signature $\Gamma_T = \{g_i \mid g_i : T_{g_i,1} \times \ldots \times T_{g_i,n} \rightarrow T\}$ iff T is inductively defined by*

$$\begin{aligned} \mathcal{P}(S, y) \quad \equiv \quad & \bigvee_{g \in \Gamma_T} \exists x_1, \ldots, x_{g,n} \cdot \\ & x_1 \in \pi_T(S, T_{g,1}) \wedge \ldots \wedge x_{g,n} \in \pi_T(S, T_{g,n}) \wedge \\ & y = g(x_1, \ldots, x_{g,n}) \end{aligned}$$

The functor symbols g in Γ_T are called the generators *or* constructors *of T.*

[4] An alternative formalization of inductively defined datatypes is pursuit by the algebraic specification school, (cf. e.g., [LEW96]), but since it relies more on the model-theoretic notion of generated algebras, the meta-reasoning is harder to automate than in the higher-order logic approach followed here.

The purpose of the projection function π_T is to "curb" the domain of the generator functions in the right arguments from the datatype being defined to the set S calculated up to the current iteration and thus to facilitate the fixpoint calculation. This in turn requires the use of unsorted existential quantifiers, i.e., a relativized formalization. However, monotonicity of f is then easy to show.

Lemma 5.2.3 *Let T be generated by Γ_T. Then the function f as defined in Definition 5.2.1 is monotone on $2^X \to 2^X$.*
Proof: *Assume $S \subseteq S' \subseteq X$ and $y \in f(S)$; hence, $\mathcal{P}(S, y)$ holds. Let $x_i \in \pi_T(S, T_{g,i})$. If $T_{g,i} \neq T$, $\pi_T(S, T_{g,i}) = \pi_T(S', T_{g,i})$, otherwise $\pi_T(S, T_{g,i}) = S \subseteq S' = \pi_T(S', T_{g,i})$. Hence, $x_i \in \pi_T(S', T_{g,i})$ by which $\mathcal{P}(S', y)$ also holds, which in turn implies $y \in f(S')$ and thus monotonicity.* $\triangle$

Hence, the notion of datatypes generated by a set of constructors is well-defined even in the many-sorted case. Consider for example the natural numbers $\mathbb{N}$ and (monomorphic) lists with generators $\Gamma_{\mathbb{N}} = \{0 : \mathbb{N}, \mathit{succ} : \mathbb{N} \to \mathbb{N}\}$ and $\Gamma_{\mathit{list}} = \{\mathit{nil} : \mathit{list}, \mathit{cons} : \mathit{item} \times \mathit{list} \to \mathit{list}\}$, respectively. $\mathbb{N}$ is generated by $\Gamma_{\mathbb{N}}$ in the usual—algebraic—sense because the *succ*-constructor has only a single argument which is of type $\mathbb{N}$. The case of *list*, however, is more complicated. It is not generated by Γ_{list} in the usual sense because Γ_{list} does not contain any constructors for the abstract *item*-type; even worse, to keep *item* truly abstract, no assumptions about its internal structure should be made at all. [LEW96] thus apply a trick to generalize the notion of generated algebras to "algebras generated in a sort" and use variables as the artificial generators of the abstract type. This trick is not necessary here, due to the more careful Definition 5.2.2: *list* is generated by Γ_{list}. Its generator predicate simplifies to

$$\mathcal{P}(S, y) \equiv y = \mathit{nil} \vee \exists x_1 : \mathit{item}, x_2 \in S \cdot y = \mathit{cons}(x_1, x_2)$$

which captures the essence of the variable trick cleanly. Of course, the generator predicate collapses to *false* (and the least fixpoint into the empty set) if the *item*-type is empty, i.e., $\exists x_1 : \mathit{item}$ is equivalent to *false*.

An immediate consequence of the fixpoint definition of T is that it admits a *cases theorem* $\forall x \cdot x \in T \Leftrightarrow \mathcal{P}(T, x)$ which "lists the ways an element of the inductively defined set can arise" [Har95]. For the natural numbers, generated by $\Gamma_{\mathbb{N}}$ as above, this yields $\forall n \cdot n \in \mathbb{N} \Leftrightarrow (n = 0 \vee \exists n' : \mathbb{N} \cdot n = \mathit{succ}\ n')$. The $\Rightarrow$-direction of the cases theorem can also be interpreted as the result of relativizing a sorted universal quantifier with the characteristic predicate "$\in T$". Reversing this relativation step thus yields the expected surjectivity axiom for the datatype in the correct many-sorted formulation.

Corollary 5.2.4 (surjectivity axiom) *For any datatype T generated by Γ_T the* surjectivity axiom

$$\forall x : T \cdot \bigvee_{g \in \Gamma_T} \exists x_1 : T_{g,1}, \ldots, x_{g,n} : T_{g,n} \cdot x = g(x_1, \ldots, x_n)$$

holds.

The fixpoint definition of inductively defined types also yields a general induction theorem; for generated datatypes this boils down to the well-structural induction schema ([Har95] for more details).

More structure can be superimposed upon a generated datatype by stipulating that it is also *freely* generated. Intuitively, this means that each element of the datatype is uniquely represented by a unique term using the constructors only. Formally, this is achieved by replacing the existential quantifiers by *unique* existential quantifiers.

Definition 5.2.5 (freely generated datatype) *A type T is* freely generated *by the signature $\Gamma_T = \{g_i \mid g_i : T_{g_i,1} \times \ldots \times T_{g_i,n} \to T\}$ iff T is inductively defined by*

$$\begin{aligned}\mathcal{P}(S, y) \quad \equiv \quad & \exists_1 g \in \Gamma_T, x_1, \ldots, x_{g,n} \cdot \\ & x_1 \in \pi_T(S, T_{g,1}) \wedge \ldots \wedge x_{g,n} \in \pi_T(S, T_{g,n}) \wedge \\ & y = g(x_1, \ldots, x_{g,n})\end{aligned}$$

The usual properties of the free constructors—injectivity and disjointedness—are than immediate consequences of the unique existential quantifiers.

Corollary 5.2.6 (injectivity axioms, disjointedness axioms) *For any datatype T freely generated by Γ_T the* injectivity axioms

$$\begin{aligned}&\forall x_1, y_1 : T_{g,1}, \ldots, x_{g,n}, y_{g,n} : T_{g,n} \cdot \\ &\quad g(x_1, \ldots, x_n) = g(x_1, \ldots, x_n) \Rightarrow x_1 = y_1 \wedge \cdots \wedge x_n = y_n\end{aligned}$$

and the disjointedness axioms

$$\begin{aligned}&\forall x_1 : T_{g,1}, \ldots, x_{g,n} : T_{g,n} y_1 : T_{f,1}, \ldots, y_{f,m} : T_{f,m} \cdot \\ &\quad \neg(g(x_1, \ldots, x_n) = f(y_1, \ldots, y_m))\end{aligned}$$

hold for all $f, g \in \Gamma_T$.

Free generation of the datatype also implies its finite generation, i.e., each element of the datatype is of finite (but unbounded) size. However, this property is difficult to formalize and to prove and not very useful for automated theorem provers. The slightly simplified variant of *acyclicity* is more useful in practice. The name acyclicity stems from the fact that the obvious implementation of infinite terms uses cyclic directed graphs; however, acyclicity rules out only immediate (i.e., starting from the root) cycles in the term graph. It is a syntactic restriction similarly to the "no constructor axioms" rule also used to characterize freely generated datatypes syntactically.

Lemma 5.2.7 (acyclicity axioms) *For any datatype T generated by Γ_T the* acyclicity axioms

$$\forall x_1 : T_{g,1}, \ldots, x_{g,n} : T_{g,n} \cdot x_i \in T \Rightarrow \neg(x_i = g(x_1, \ldots, x_n))$$

hold for all $g \in \Gamma_T$, provided that Γ_T contains at least one constant.

The proof proceeds by structural induction over x_i which makes the proviso necessary. The theorem can, however, be generalized to the case where Γ_T contains proper constructor functions only.

Proof: Since Γ_T contains a constant c and g cannot be a constant, c and g are different and the base case holds by the disjointedness axiom. For the step case, only the case of $x_i = g(x_1, \ldots, x_n)$ has to be considered, since all other cases hold trivially, again by disjointedness. Hence, applying surjectivity over x_i after substitution yields

$$\begin{array}{l} \forall x_1 : T_{g,1}, \ldots, x_{g,n} : T_{g,n} \cdot x_i \in T \Rightarrow \\ \quad \neg\exists y_1 : T_{g,1}, \ldots, y_{g,n} : T_{g,n} \cdot \\ \qquad g(y_1, \ldots, y_n) = g(x_1, \ldots, x_{i\text{-}1}, g(y_1, \ldots, y_n), x_{i+1}, \ldots, x_n) \end{array}$$

which reduces by injectivity and repeated application of the witness rule 5.16 to the induction assumption. $\triangle$

5.3 Quantifier Splitting

Although $\rightsquigarrow$ already detects some non-obvious inconsistencies, it is not yet sufficient as simplification filter: to achieve significant reductive effects, more hidden inconsistencies must be exposed. Consider for example an $\mathcal{R}$-irreducible (but inconsistent) formula

$$\forall x, y : T \cdot x < y \wedge \mathcal{F}[x, y] \qquad (*)$$

To detect the inconsistency it is sufficient to consider the case $x = y$ only (provided that $x < x \rightsquigarrow \mathit{false}$.)

Quantifier splitting techniques are rewrite-based implementations of such case distinctions, although in a brute-force manner. Hence, splitting is non-terminating in general and requires an additional control strategy. The generic formulations of the splitting rules (w.r.t. a *split predicate* p) are

$$\begin{array}{l} \forall \vec{x} \cdot \mathcal{F}[\vec{x}] \rightsquigarrow \forall \vec{x} \cdot (p(\vec{x}) \Rightarrow \mathcal{F}[\vec{x}]) \wedge (\neg p(\vec{x}) \Rightarrow \mathcal{F}[\vec{x}]) \\ \exists \vec{x} \cdot \mathcal{F}[\vec{x}] \rightsquigarrow \exists \vec{x} \cdot (p(\vec{x}) \wedge \mathcal{F}[\vec{x}]) \vee (\neg p(\vec{x}) \wedge \mathcal{F}[\vec{x}]) \end{array} \qquad (**)$$

It is easy to see that quantifier splitting is an equivalence-preserving simplification. This follows from equivalence preservation of the distributive and adjunctive laws of the propositional calculus.

Lemma 5.3.1 *Quantifier splitting is equivalence-preserving for any split predicate p.*
Proof: *For the universal splitting rule we have*

$$\begin{aligned}
& \forall \vec{x} \cdot (p(\vec{x}) \Rightarrow \mathcal{F}[\vec{x}]) \wedge (\neg p(\vec{x}) \Rightarrow \mathcal{F}[\vec{x}]) \\
\equiv\ & \forall \vec{x} \cdot (\neg p(\vec{x}) \vee \mathcal{F}[\vec{x}]) \wedge (p(\vec{x}) \vee \mathcal{F}[\vec{x}]) \\
\equiv\ & \forall \vec{x} \cdot ((\neg p(\vec{x}) \vee \mathcal{F}[\vec{x}]) \wedge p(\vec{x})) \vee ((p(\vec{x}) \vee \mathcal{F}[\vec{x}]) \wedge \mathcal{F}[\vec{x}]) \\
\equiv\ & \forall \vec{x} \cdot (p(\vec{x}) \wedge \mathcal{F}[\vec{x}]) \vee \mathcal{F}[\vec{x}] \\
\equiv\ & \forall \vec{x} \cdot \mathcal{F}[\vec{x}]
\end{aligned}$$

The equivalence preservation of the existential splitting rule follows by duality. △

Quantifier splitting resembles the propositional splitting which is for example used in the Davis-Putnam-Procedure [DP60, DLL62]. Of course, the practical problem of quantifier splitting is similar to the problem in the propositional case, viz. the "judicious" choice of the split predicate which must trigger enough simplifications. In NORA/HAMMR, I have experimented with two different predicates which I describe subsequently.

Diagonalization

Diagonalization is the most basic splitting technique. It makes no further assumptions about the domain of the bound variable and splits w.r.t. the equality predicate, as in the example (*) above.

Definition 5.3.2 (diagonalization rule) *The* diagonalization rules *for any type T are*

$$\begin{aligned}
& Q\, x : T \cdot \forall y : T \cdot \mathcal{F}[x, y] \rightsquigarrow Q\, x : T \cdot \mathcal{F}[x, x] \wedge \forall y : T \cdot x \neq y \Rightarrow \mathcal{F}[x, y] \qquad (5.17) \\
& Q\, x : T \cdot \exists y : T \cdot \mathcal{F}[x, y] \rightsquigarrow Q\, x : T \cdot \mathcal{F}[x, x] \vee \exists y : T \cdot x \neq y \wedge \mathcal{F}[x, y]
\end{aligned}$$

for an arbitrary quantifier Q.

Hence, the split predicate used for diagonalization is not equality but more precisely a curried version "$= y$" of equality; this currying-process also requires the second, nested quantifier in the formulation of the rule. The name "diagonalization rule" recalls the fact that (5.17) isolates the diagonal over $T \times T$ into a separate conjunct in which—hopefully—a contradiction becomes apparent; it thus also resembles the traditional diagonalization proof technique invented by Cantor.

In order to make diagonalization into a useful simplification technique, it should be applied only if the isolated conjunct (disjunct) ultimately becomes *true* (*false*), at least with a high probability. One easy and cheap way to ensure such a high probability of success is to exploit the syntactic structure of $\mathcal{F}$ and to follow these simple rules of thumb:

- Apply the universal diagonalization rule only if $\mathcal{F}$ contains a positive occurrence of an irreflexive predicate or a negative occurrence of a reflexive predicate.
- Apply the existential diagonalization rule only if $\mathcal{F}$ contains a positive occurrence of an reflexive predicate or a negative occurrence of a irreflexive predicate.

Instances of reflexive and irreflexive predicates, respectively, can easily be detected in the lemma library; moreover, the equality predicate is reflexive by definition and can thus be used in both variants.

Surjective Unrolling

Surjective Unrolling is a splitting technique which is applicable only for generated datatypes. It splits with respect to the surjectivity axiom of the datatype.[5]

Definition 5.3.3 (surjective unrolling rule) *The* surjective unrolling rules *for any generated type T with generators Γ_T are*

$$\begin{aligned} \forall x : T \cdot \mathcal{F}[x] &\rightsquigarrow \bigwedge_{f \in \Gamma_T} \forall \vec{x}_f : \vec{T}_f \cdot \mathcal{F}[f(\vec{x}_f)] \\ \exists x : T \cdot \mathcal{F}[x] &\rightsquigarrow \bigvee_{f \in \Gamma_T} \exists \vec{x}_f : \vec{T}_f \cdot \mathcal{F}[f(\vec{x}_f)] \end{aligned} \tag{5.18}$$

This definition of the surjective unrolling rule is not a direct instantiation of the generic splitting schema (**) on page 102 but a result of simplifying (**) w.r.t. the usual predicate calculus rules and noting that the second conjunct (resp. disjunct) can be dropped since $\neg p(\vec{x})$ is always *false*.

The surjective unrolling rule is defined for any generated datatype; however, in practice its usefulness depends to a large extent on the disjointedness axioms of freely generated datatypes to simplify the large formulae resulting from an unrolling step.

5.4 Rewrite Strategy

Since NORA/HAMMR does not apply completion, the generated term rewrite system $\rightsquigarrow$ is not necessarily confluent; moreover, quantifier splitting is nonterminating. Selection of a proper rewrite strategy may thus have major impacts on the efficiency of a rewrite-based rejection filter, not only in terms of runtime but also in terms of reduction rates.

[5] A proof rule similarly to the existential version of surjective unrolling has recently been proposed in the context of model-generation theorem provers for full first-order logic [Ahr00].

For pragmatic reasons, NORA/HAMMR distinguishes between reduction and expansion core simplifications and uses a mixed *top-down/bottom-up* rewrite strategy. Reduction simplifications (which reduce the size of the term or the number of variable occurrences) are applied top-down in order to maximize the effect of the rules (5.1) and (5.3); this is supported further by their internal ordering. Expansion simplifications as distributivity (which are terminating with respect to a more complicated termination ordering than just the term size) are applied bottom-up in order to minimize the growth in term size. As mentioned before (cf. Page 93), the rule set contains only one of the distributive laws to ensure termination.

The distinction between reduction and expansion rules also applies to the quantifier rules, especially to (5.10) and (5.11). The experiments have shown that good effects are already achieved if these rules are considered as expansion rules, i.e., applied bottom-up. This combines to some extent the advantages of both the prenex- and anti-prenex-orientation of the rules because the quantifier bodies are reduced first, potentially allowing more simplifications, before the quantifier scopes are reduced, potentially allowing quantifier elimination. However, NORA/HAMMR does currently not contain an extra post-simplification prenex-phase to merge quantifier scopes together.

Diagonalization and surjective unrolling are non-terminating and require an additional control strategy. NORA/HAMMR uses a very simple strategy. Simplification and diagonalization/unrolling are two separate alternating phases; if the term size after a simplification phase does not exceed a user-specified limit, all quantifier occurrences are unrolled once.

5.5 Experimental Results

Tables 5.1 and 5.2 summarize the results of the simplification-based rejection filters for different simplification levels. Here, $\overline{|t|}$ is the average size of the proof tasks, defined as the term size of the first-order term representing the conjecture, i.e., not counting the axiomatization. $\overline{T}_{task}$ and $\overline{T}_{query}$ are the average response times per task and per query, respectively; each query comprises 119 (i.e., $|\mathcal{L}|$) single tasks. The lower parts of both tables list recall, precision, and fallout as defined in Section 2.3. However, since all rewrite systems are sound, the rejection filters are recall-preserving and the recall is consistently 100%. For precision and fallout, I give both query-oriented and document-oriented averages. Finally, $\underline{red}$ is the *reduction factor* of the filter, i.e., the fraction of tasks which could be simplified to *false*. $\underline{red}^+$ is the *total reduction factor* which also includes tasks which are rewritten to *true*.

Table 5.1 gives the results of the base simplifications, i.e., without quantifier splitting. Four different simplification levels are listed. $\mathcal{R}_{FOL}$ contains the usual first-order simplifications (rules 5.1 to 5.9), $\mathcal{R}_{EQ}$ additionally contains simplifica-

tion for equality (rules 5.13 to 5.16), and $\mathcal{R}_{DOMAIN}$ is the full set of automatically generated domain-specific simplifications, including $\mathcal{R}_{EQ}$. $\mathcal{R}_{\emptyset}$ is the empty rewrite system (i.e., no simplification); it is included for reference only. Finally, the CNF- and DNF-entries in the second line indicate in which direction the distributive law has been oriented and, hence, which normal form is prepared.

	$\mathcal{R}_{\emptyset}$	$\mathcal{R}_{FOL}$	$\mathcal{R}_{EQ}$	$\mathcal{R}_{DOMAIN}$		
	CNF	CNF	CNF	CNF	DNF	comp.
$\overline{\lvert t \rvert}$	87.9	177.1	171.6	55.7	61.8	-
$\sigma_{\lvert t \rvert}$	29.4	137.1	176.0	64.9	37.2	-
$\overline{T}_{task}$ (sec.)	0.03	0.07	0.08	0.06	0.05	0.06
σ_T	0.01	0.04	0.06	0.04	0.02	0.04
T_{max}	0.09	0.46	1.25	0.44	0.29	0.44
$\overline{T}_{query}$ (sec.)	3.39	7.97	9.45	7.38	5.85	7.51
σ_T	0.59	3.61	5.05	3.60	1.84	3.54
T_{max}	5.68	21.59	27.63	21.65	14.48	21.54
$\underline{r}$ (%)	100.00	100.00	100.00	100.00	100.00	100.00
$\underline{p}$ (%)	12.98	12.98	12.98	17.59	13.47	17.59
$\underline{\delta}_p$	1.00	1.00	1.00	1.35	1.04	1.35
$\overline{p}$ (%)	12.98	12.98	12.98	17.48	14.08	17.48
σ_p	19.17	19.17	19.17	23.59	21.62	23.59
$\overline{\delta}_p$	1.00	1.00	1.00	1.46	1.06	1.46
$\underline{f}$ (%)	100.00	100.00	100.00	69.9	95.8	69.9
$\overline{f}$ (%)	100.0	100.00	100.00	69.3	94.5	69.3
σ_f	0.00	0.00	0.00	21.5	14.9	21.5
$\underline{red}$ (%)	0.00	0.00	0.00	26.19	3.67	26.19
$\underline{red}^+$	0.00	2.75	4.23	31.78	8.63	31.78

Table 5.1: Rewrite-Based Rejection: Base Simplifications

The most obvious result is that purely syntactic simplification are insufficient for task rejection: neither $\mathcal{R}_{FOL}$ nor $\mathcal{R}_{EQ}$ are able to identify a single non-theorem. Hence, the fallout is 100% in both cases. However, since both variants already reduce a significant fraction of the valid tasks to *true* ($\mathcal{R}_{FOL}$: 21.0%, $\mathcal{R}_{EQ}$: 32.6%), small total reduction factors are achieved.

The situation improves clearly if the domain-specific simplifications are added. Using $\mathcal{R}_{DOMAIN}$ to rewrite into CNF-style, more than 30% of the tasks are eliminated in total. But again, it performs better on the valid tasks (43.1%) than on the invalid ones (26.2%). Both fallout averages are close to 70% which indicates that the filter works equally well for all queries. The two rightmost columns show

that rewriting into CNF-style is not only clearly better than rewriting into DNF-style but that the former even subsumes the latter: the reduction factor does not change if both variants are run in a competitive mode. CNF-style remains superior even with unrolling although it than does no longer subsume DNF-style. However, this is unfortunate. Because most (refutation-oriented) provers can handle the tasks much better if they are in DNF-style,[6] in fact both variants need to be generated: CNF-style for rejection purposes, DNF-style for the actual proof attempts.

Table 5.2 gives the results of the quantifier unrolling strategy over $\mathcal{R}_{DOMAIN}$ for two different timeouts, using a maximal term size of 10.000 as additional termination criterion.

	CNF			DNF			pipe.		comp.	
T_{max} (sec.)	0.50	5.00	∞	0.50	5.00	∞	0.50	5.00	0.50	5.00
$\overline{T}_{task}$ (sec.)	0.24	1.48	9.39	0.25	1.17	13.59	0.25	1.42	0.23	1.40
σ_T	0.21	2.10	118.93	0.21	1.74	258.59	0.21	1.99	0.21	2.09
T_{max}	-	-	5292	-	-	26241	-	-	-	-
$\overline{T}_{query}$ (sec.)	28.63	175.59	1117	30.18	139.30	1616	29.20	169.02	26.93	166.14
σ_T	10.77	88.50	3042	11.40	85.41	10771	11.03	90.06	10.13	90.42
T_{max}	54.73	460.10	30708	56.92	405.78	106069	57.76	445.66	54.75	477.06
$\underline{r}$ (%)	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00
$\underline{p}$ (%)	30.06	35.37	37.23	28.79	33.50	34.41	29.10	36.02	32.19	36.83
$\underline{\delta}_p$	2.32	2.73	2.87	2.22	2.58	2.65	2.24	2.78	2.48	2.84
$\overline{p}$ (%)	26.29	28.98	30.16	24.24	28.08	28.62	25.11	29.55	26.87	30.28
σ_p	27.75	28.13	27.96	26.83	27.84	27.84	27.32	27.84	27.45	27.83
$\overline{\delta}_p$	3.79	5.05	5.38	3.25	5.19	5.32	3.33	5.43	4.41	5.70
$\underline{f}$ (%)	34.70	27.25	25.15	36.89	29.61	28.43	36.34	26.50	31.41	25.58
$\overline{f}$ (%)	35.11	28.27	26.32	37.57	30.66	29.52	36.76	27.62	32.18	26.76
σ_f	19.32	18.29	18.59	20.09	19.53	19.13	19.65	18.76	18.61	18.93
$\underline{red}$ (%)	56.83	63.31	65.14	54.92	61.25	62.28	55.40	63.96	59.69	64.76
$\underline{red}^+$	64.44	70.99	72.82	62.21	68.61	72.00	63.37	71.95	68.68	72.75

Table 5.2: Rewrite-Based Rejection: Unrolling, $|\,t\,|_{max} = 10000$

It is obvious that unrolling dramatically improves the filtering effect. Already within half a second per task, reduction factors of approximately 55% can be achieved, while the total reduction factors even reach approximately 65%. Moreover, this more than twofold increase is paid for with only moderate response

[6] In fact, SETHEO's normal form translator was not even able to handle the proof tasks in DNF-style, due to the excessive size of the emerging intermediate formulae.

times which amount to an average of roughly 30 seconds per query and are "sub-minute" even in the worst case. Further increases of the timeout still improve these results, although a saturation effect soon occurs. While reduction factors of almost 65% can be achieved for moderate timeouts, unrolling with termination enforced only via the maximal term size becomes prohibitively slow, due to a small number of tasks (cf. the large standard deviation for T_{task}).

As a consequence of the improved filtering effect, the fallout numbers decrease significantly, dropping below the 30%-mark already for moderate timeouts. Again, query-oriented and document-oriented averages lie within a small margin which indicates again that the filter works equally well for all queries. Since rewriting is sound, the precision of the answer set increases at the same time. Due to the large variation in the size of the relevance sets ($\sigma_{REL} = 19.2\%$), however, a peculiar difference between the query-oriented and document-oriented points of view can be observed. Although the system-wide, document-oriented average is for every timeout and combination higher than the query-oriented average, these order turns upside-down if the leverages are considered. Here, queries with very small relevance sets induce extraordinary large precision leverages which skew the average. This effect is even reinforced by the fact that such queries usually also induce a large number of "easy" non-matches which can be ruled out.

Another effect which is also evident from Table 5.2 is that CNF-style and DNF-style are now much closer to each other, even if the formal still performs slightly better. But in contrast to the base simplifications, CNF-style does not subsume DNF-style any longer. This can be utilized by combining both into a single filter.

For such a combination, two different approaches are possible, pipelining and true parallelism. Here, pipelining is implemented very straightforward, requiring only a single processor. For each task, the variants are simply checked after each other. If one variant succeeds, the remaining ones are dropped and the next task is taken. Depending on the chosen order and allocated time slices, quite different results can be achieved. For the experiments, I chose CNF-style as first filter in the pipeline, because it shows slightly better results, and for simplicity, I just split the total time into two equal slices. The results show that this is not the optimal division, at least not for short timeouts: for 0.5 seconds, the combined filter performs slightly worse than the pure CNF-style filter. However, with increased timeouts the actual time slices become less important and the effect of exploring the search spaces from different starting points begins to dominate. For 5 seconds better results are achieved for the pipeline than for the pure CNF-style filter. Although the improvements are relatively small, they should not be devaluated because they already contain a major part of the maximal possible (i.e., without time limits) improvements.

If multiple processors are available, further improvements can be achieved by a parallel competition between the variants. Here, the first processor which

arrives at a decision (i.e., rejection or acceptance) "wins" and aborts the others. A parallel rejection filter need not necessarily to be faster than its individual sub-filters because it has to wait for even the slowest sub-filter if none of them arrives at a decision. However, it returns the same results as a pipeline consisting of the same sub-filters would do but is always faster than that.

In practice, competition is at least faster than the slowest sub-filter but gives better results. The reduction factor increases up to almost 65% and is quite close to the maximum achieved without time limits. The fallout averages drop to slightly more than 25%, i.e., almost three out of four irrelevant components are filtered out, and the precision averages accordingly increase.

In fact, a rewrite-based rejection filter may already be considered as the final pipeline element. The pipeline is then guaranteed to be recall-preserving, but the retrieved components require manual inspection before they may be reused. Nevertheless the achieved precision level of roughly 30% is already competitive, even if it is compared to more traditional, information retrieval methods. For example, Y. Maarek et al. [MBK91] report for two different systems between 40% and 50% average precision at a 90%-recall level. However, since these were the maximum recall levels achieved by the systems, the results do not allow a definitive comparison of the approaches.[7]

The large differences between the rejection and total reduction factors indicate that the simplifications already reduce a significant number of proof tasks to *true*. This raises the question of how they perform as low-cost confirmation filters. Table 5.3 gives some answers to this question. Here, the timeouts were chosen to correspond with those used in the ATP-based filters.

	CNF		DNF		pipe.		comp.	
T_{max} (sec.)	1.00	90.00	1.00	90.00	1.00	90.00	1.00	90.00
$\overline{T}_{task}$ (sec.)	0.41	4.73	0.42	3.03	0.41	6.08	0.46	6.27
σ_T	0.43	12.18	0.42	10.61	0.43	12.95	0.43	15.42
$\overline{T}_{query}$ (sec.)	48.43	562	49.75	360	48.80	724	54.19	746
σ_T	20.62	517	21.32	784	20.70	655	22.10	901
# proofs	1081	1088	1042	1042	1128	1131	1131	1131
$\underline{r}$ (%)	58.81	59.19	56.69	56.69	61.37	61.53	61.53	61.53
$\overline{r}$ (%)	38.37	38.65	34.44	34.44	39.54	39.65	39.65	39.65
σ_r	37.86	37.77	37.96	37.96	37.67	37.64	37.64	37.64
$\overline{p}$ (%)	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00

Table 5.3: Rewrite-Based Confirmation: Unrolling, $|t|_{max} = 10000$

[7]Moreover, the IR-systems used a different test library and query set. Still, the numbers indicate that deduction-based methods become practical.

The most obvious answer is that rewriting using the quantifier unrolling strategy is a suitable and particularly fast confirmation filter: even within a single second per task, roughly 60% of the valid proof tasks are identified. Suitability even holds under the more critical user-oriented view: the response times per query are "sub-minute", and the query-oriented recall average is still between 35% and 40%. Hence, rewriting creates a strong bound for the application of theorem provers.

However, Table 5.3 also reveals that this approach does not scale any further. Even with a timeout increased by almost two orders of magnitude, only a neglectable number of additional proofs can be found. Moreover, further increases do not yield any further proofs (cf. the differences between $\underline{red}$ and $\underline{red}^+$ in Table 5.2). This is in contrast to the rejection case where further improvements can be achieved, although with a diminishing return.

Chapter 6

Counterexample-based Rejection

The simplification-based rejection filters described in the preceding chapter rule out only such components which are associated with contradictory proof tasks. Such filters are still too coarse: if $\mathcal{T}[q, c]$ is neither valid nor contradictory, rewriting fails to produce a definitive result. In such cases, counterexample-based techniques can be applied. If c is a non-match for q, $\mathcal{T}[q, c]$ must admit at least one *counterexample*, i.e., a structure, an interpretation (of the predicate and function symbols) and an assignment (of the free variables) under which it evaluates to *false*. Such counterexamples can be exploited by two different approaches.

- A number of structures and interpretation are fixed such that they are models of the background theory $\mathcal{A}$ (i.e., the axiomatization of the extralogical symbols). Under these, all assignments of $\mathcal{T}[q, c]$ are checked. If no counterexample is found, nothing can be concluded.

- $\mathcal{A} \Rightarrow \mathcal{T}[q, c]$ is checked exhaustively, i.e., under all structures, interpretations, and assignments. If no counterexamples are found, $\mathcal{T}[q, c]$ is valid in the background theory and c can safely be accepted.

Obviously, both approaches terminate only if the structures and thus the number of possible interpretations and assignments are finite. Unfortunately, and in contrast to the situation in hardware verification, most interesting structures in software engineering applications are infinite; in the test library, the list datatype induces infiniteness. Depending on the approach, different techniques are possible to ensure termination:

- Only a finite substructure is fixed, e.g., only the domain of a particular datatype, and only assignments over this substructure are checked. However, if $\mathcal{T}[q, c]$ cannot be mapped completely onto the substructure, checking cannot be done by exhaustive evaluation but still requires more general techniques, e.g., rewriting or theorem proving.

- Finiteness of the structures is enforced by application of appropriate approximations or abstractions. The domain elements are abstracted into a finite number of equivalent classes; lists for example may be abstracted into the classes $nil = \{[\,]\}$ and $non\text{-}nil = \{x \mid x \neq [\,]\}$. Similarly, the concrete functions and predicates are replaced by abstracted counterparts which work on the equivalence classes. Then the abstracted problem $[\![\, \mathcal{A} \Rightarrow \mathcal{T}[q, c] \,]\!]$ is checked exhaustively. This is also called *abstract model checking*.

Usually, such counterexample-based rejection filters are no longer recall-preserving because wrong substructures or unsound abstractions may produce spurious counterexamples. It is thus an engineering problem to balance this loss of recall against the rejective power of the filter.

This chapter describes the two counterexample-based rejection filters implemented in NORA/HAMMR. Both filters follow the "fixed substructure" approach but rely on different evaluation techniques. Section 6.1 deals with rewriting over finite *item*-domains while Section 6.2 describes the attempts to prove that tasks are contradictory, again under the assumption of various finite *item*-domains. Rejection filters based on model checking techniques turned out to be inferior and showed relatively low precision and recall rates in preliminary experiments. They are thus not investigated closer in this thesis (cf. Chapter 10.3.3 for some details).

6.1 Rewriting over Finite *item*-Domains

6.1.1 Counterexample Domains

The free generators *nil* and *cons* of the *list*-datatype allow surjective unrolling of all *list*-quantifiers. The *item*-datatype, which is more abstract, does not provide such information and admits only the less effective diagonalization rule. However, this abstractness can also be turned into an advantage because now any—and in particular any finite—number of *virtual generators* can be invented to describe a specific fixed structure. These virtual generators again allow unrolling of the *item*-quantifiers but this is only valid in the selected structure and not in general. Nevertheless, the resulting rejection filter can be recall-preserving if the abstract *item*-domain and the interpretation of the predicates over this domain are chosen judiciously. Obviously, this choice cannot be automated but requires the human insights of a reuse administrator.

In NORA/HAMMR, I have experimented with several sound and unsound prospective counterexamples:

1. Since all types can be assumed to be inhabited,[1] the smallest possible counterexample contains only a single item i_0. In this structure, some of the

[1] This is a well-formedness condition of VDM-SL-specifications.

functions and predicates of the domain theory become trivial and can thus be eliminated from the tasks, entailing further simplification. For example, the *hd*-function becomes trivial since

$$\forall l : list \cdot l \neq [] \Rightarrow hd\ l = i_0$$

holds. However, since the axiom

$$\exists i : item \cdot \exists j : item \cdot i \neq j$$

does not hold, unrolling becomes unsound in this structure and the filter may lose recall. This problem can be be fixed by stipulating that i_0 is a generator but not or not necessarily the only one. To model these assumptions, the surjective unrolling rule must be modified, essentially by a combination of unrolling and diagonalization,

$$\exists i : item \cdot \mathcal{F}[i] \rightsquigarrow \mathcal{F}[i_0] \vee \exists i : item \mathcal{F}[I]$$

or, respectively,

$$\exists i : item \cdot \mathcal{F}[i] \rightsquigarrow \mathcal{F}[i_0] \vee \exists i : item \cdot i \neq i_0 \wedge \mathcal{F}[i]$$

and with similar rules holding for universal quantifiers.

2. The next, larger *item*-domain contains two different elements, i_0 and i_1. It gives rise to two prospective counterexample structures, depending on the interpretation of the $\leq$-predicate over this domain:

   (a) $i_0 \leq i_1$

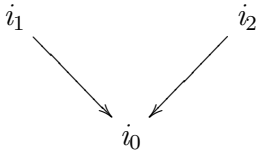
   (b) $\neg i_0 \leq i_1$

   However, although all axioms hold under both interpretations, they may induce "too much" internal structure, thereby possibly degrading the rejective power and the precision of the filter:

   (a) Under this interpretation, the *item*-domain is a *total* order which is not required by the original axioms.

   (b) Under this interpretation, the *item*-domain admits no non-trivial partial order and thus also no strict order. Hence, all sorted lists consist of equal elements only.

3. The final investigated *item*-domain thus comprises the smallest non-trivial partial order:



   I.e., it contains a minimal element i_0 and two other mutually incomparable elements i_1 and i_2.

6.1.2 Experimental Results

The above substructures have been turned into rewrite rules using the techniques described in the previous chapter. These rules were then used to replace the weaker diagonalization rule in the original rewrite system $\mathcal{R}_{DOMAIN}$ which produced the best filtering effects. The resulting rewrite systems were then applied to the tasks, again with a maximal term size for unrolling of 10000.

Unfortunately, some of the *item*-domains do not work very well with such an aggressive quantifier unrolling. The larger number of real and virtual generators and the fact that now *all* occurring quantifiers can be unrolled lead to an explosive term growth which is not always offset by the additional simplifications and thus degrades the rejective power of the filters. This is particularly true for the two modifications of the single-*item*-domain and for the substructure with three *items*. Less aggressive unrolling prevents this term explosion but in turn degrades the overall effectiveness of the filters even more. The above three domains are thus not investigated in more detail.

	$\mathcal{R}_{DOMAIN}$		$\mathcal{R}_1$		$\mathcal{R}_{2,a}$		$\mathcal{R}_{2,b}$		comp.	
T_{max} (sec.)	0.50	5.00	0.50	5.00	0.50	5.00	0.50	5.00	0.50	5.00
$\overline{T}_{task}$ (sec.)	0.24	1.48	0.15	0.53	0.24	1.36	0.23	1.19	0.16	0.74
σ_T	0.21	2.10	0.16	1.28	0.21	2.02	0.20	1.91	0.17	1.60
$\overline{T}_{query}$ (sec.)	28.63	175.59	18.01	63.62	28.85	161.83	27.49	141.55	19.51	88.06
σ_T	10.77	88.50	9.81	62.42	11.30	105.26	11.33	99.41	9.98	91.22
$\underline{r}$ (%)	100.00	100.00	100.00	100.00	100.00	100.00	99.67	99.62	99.67	99.62
$\underline{p}$ (%)	30.06	35.37	35.43	37.49	29.59	40.17	31.45	43.38	43.90	53.65
$\underline{\delta}_p$	2.32	2.73	2.73	2.89	2.28	3.10	2.42	3.34	3.39	4.14
$\overline{p}$ (%)	26.29	28.98	31.74	35.22	25.86	33.41	27.77	36.38	39.89	51.60
σ_p	27.75	28.13	29.77	31.64	27.19	27.82	28.16	28.95	31.24	32.57
$\overline{\delta}_p$	3.79	5.05	7.80	10.19	3.28	6.78	3.64	7.28	10.49	17.71
$\underline{f}$ (%)	34.70	27.25	27.15	24.84	35.44	22.19	32.40	19.39	18.98	12.82
$\overline{f}$ (%)	35.11	28.27	28.45	26.30	35.89	23.51	32.60	20.45	20.08	14.27
σ_f	19.32	18.29	21.93	21.72	20.73	20.44	21.36	20.02	20.71	19.57
$\underline{red}$ (%)	56.83	63.31	63.41	65.42	56.19	67.72	58.87	70.19	70.56	75.93

Table 6.1: Rewrite-Based Rejection: Fixed Domains, Unrolling, $|t|_{max} = 10000$

Table 6.1 shows the results of the remaining filters for the same timeouts as in Section 5.5; for reference, the results of $\mathcal{R}_{DOMAIN}$ are repeated in the two leftmost columns. The main result of this experiment is that the restriction to finite *item*-domains significantly increases the effectiveness of the rewrite-based rejection filters, independent of the particular domain. The reduction factors

increase by up to 17%-points, depending on domain and timeout. The fallout numbers decrease equivalently, dropping even below the 20%-mark for the $\mathcal{R}_{2,b}$-based filter and $T_{max} = 5$ secs.

A second important observation is that all selected finite *item*-domains are well engineered compromises. Both $\mathcal{R}_1$ and $\mathcal{R}_{2,a}$ induce recall-preserving filters, i.e., their relative defect ratio is $\underline{\delta_e} = 0.0$. $\mathcal{R}_{2,b}$ induces an almost recall-preserving filter which looses only seven matches and which thus yields an average error quota of $\underline{e} = 0.07\%$ and a relative defect ratio of $\underline{\delta_e} = 0.01$. Moreover, six of these lost matches are distributed over only two different components, *smallest* and *greatest*. Both component specifications require the input to be a total order but this cannot hold over this particular domain. Hence, their preconditions evaluate (over this domain) always to *false* which causes the failure to retrieve the two components for some queries. As a consequence of these low defect ratios the increased reduction factors immediately turn into an increased precision. Despite the fact that the $\mathcal{R}_{2,b}$-based filter is not completely recall-preserving, it produces again the best results ($\underline{p} = 43.38\%$).

As to be expected, the time-dependent behavior of the the filters is strongly influenced by the size of the respective *item*-domains. In the case of $\mathcal{R}_1$, unrolling of an *item*-quantifier already decreases the size of the proof task. Together with the possible elimination of some function and predicate symbols this slows down the term growth caused by unrolling of the *list*-quantifiers which in turn accelerates the rewriting process. This is further reinforced by the improved effectiveness as more tasks can be rewritten to *true* or *false* and thus do not consume the entire allocated time frame. These effects (and hence the speed-ups) grow with increasing timeouts. $\mathcal{R}_1$ yields a speed-up of $s = 1.6$ over $\mathcal{R}_{DOMAIN}$ for $T_{max} = 0.5$ secs.; this grows to $s = 2.76$ for $T_{max} = 5.0$ secs. The resulting average response times of roughly 20–60 seconds per query (for $T_{max} = 0.5$ secs. and $T_{max} = 5.0$ secs., respectively) are already at the brink of interactiveness and certainly comparable to web-based retrieval methods working with a slow network connection. Even for $T_{max} = 5.0$ secs. the slowest query requires only 305 seconds (instead of $5.0 * 119 = 595$ seconds) and only 20 of the 119 queries in the test set take longer than 120 seconds. Due to the improved efficiency the true speed-ups[2] are even higher, e.g., $s' = 10.51$ based on $T_{max} = 0.5$ secs. for the $\mathcal{R}_1$-based filter. However, even this filter shows saturation effects. A further increase in the timeout to $T_{max} = 90$ secs. yields only a small improvement ($\underline{red} = 65.64\%$). Additionally, the already relatively (compared to $\mathcal{R}_{DOMAIN}$) small improvement from $T_{max} = 0.5$ secs. to $T_{max} = 5.0$ secs. indicates that the saturation now occurs significantly earlier as in the $\mathcal{R}_{DOMAIN}$-based filter.

In the case of the two $\mathcal{R}_2$-based filters, unrolling of an *item*-quantifier essentially doubles the size of the proof task. Since NORA/HAMMR's implementation does not (yet) use a structure-sharing representation for terms, rewriting now

[2]For zero-defect filters the true speed-up is defined at the lowest precision level.

takes longer than in the case of $\mathcal{R}_1$ and requires for $T_{max} = 0.5$ secs. approximately the same time as in the original rewrite system $\mathcal{R}_{DOMAIN}$. With increased timeouts, however, the higher effectiveness of the filter becomes dominant again and the higher number of task rewritten to *true* or *false* within the allocated time frame cancels out the term growth. For $T_{max} = 5.0$ secs. both filters thus achieve small speed-ups over $\mathcal{R}_{DOMAIN}$ ($\mathcal{R}_{2,a}$: $s = 1.09$, $\mathcal{R}_{2,b}$: $s = 1.24$). In contrast to $\mathcal{R}_1$, the saturation effects start later and both filters still profit from an increased timeout of $T_{max} = 90$ secs. ($\mathcal{R}_{2,a}$: $\underline{red} = 71.17\%$, $\mathcal{R}_{2,b}$: $\underline{red} = 73.18\%$)

The three different *item*-domains serve as counterexamples for largely different subsets of the tasks. This becomes apparent (and can be exploited) by a competition between the filters. The combination significantly improves the results of the constituent filters. The reduction factor passes the 70%-mark already after an individual timeout of $T_{max} \approx 0.5$ secs., i.e., the combination achieves a true speed-up of $s' \approx 10$ over the best individual filter and thus a true efficiency of $e' \approx 3.3$. For $T_{max} = 5.0$ secs. the reduction factor even increases to more than 75%. The document-oriented fallout average drops down to 12.82%, i.e., in total the filter detects almost seven out of eight non-matches. However, the query-oriented average is slightly higher ($\overline{f} = 14.27\%$) and the standard deviation is with $\sigma_f = 19.57$ in relation higher than in the individual filters. This indicates that the combination works well for most of the queries but that every individual filter performs relatively poor for a small set of hard queries. In fact, in all three filters the same three queries have with more than 90% the highest fallout.

By construction, the combined filter inherits only the small error quota of $\mathcal{R}_{2,b}$. Hence, the improved effectiveness reflects itself not only in the higher reduction factor but also in a dramatically higher precision—in total, more than the half of the components passing this combination are relevant. The query-oriented average jumps up even more and also passes the 50%-mark. This confirms that only a small set of queries is "resistant" to the chosen counterexamples.

Although finite countermodels are in NORA/HAMMR primarily used to implement rejection filters, it is interesting to see how they perform as further low-cost confirmation filters, similarly to the approach followed in the general case in Section 5.3. Obviously, such confirmation filters can no longer guarantee 100% precision because the validitity of a task in one particular structure does not imply its validity in general.

In practice, however, these low-cost confirmation filters perform rather well (cf. Table 6.2). For $T_{max} = 5.0$ secs. they achieve recall levels of roughly 65%–75% and precision levels of roughly 60%–90%. Moreover, all filters have smaller relative defect ratios than the sound $\mathcal{R}_{DOMAIN}$-based filter. This indicates that the loss of precision is in relation smaller than the gain in recall.

The aggressive abstraction built into $\mathcal{R}_1$ allows the highest number of tasks to be rewritten to *true* and thus the highest recall ($\underline{r} = 77.51\%$) but it also allows the highest number of spurious "proofs" and thus the highest fallout ($\underline{f} = 7.55\%$) and the lowest precision ($\underline{p} = 60.48\%$). Curiously, however, it also yields the

smallest relative defect ratio of all variants. The less aggressive abstractions in $\mathcal{R}_{2,a}$ and $\mathcal{R}_{2,b}$ lead to fewer spurious "proofs" and thus increase the precision but due to the involved larger terms less tasks are rewritten to *true* and *false* so that the filters reach only a significantly lower recall level.

	$\mathcal{R}_{DOMAIN}$		$\mathcal{R}_1$		$\mathcal{R}_{2,a}$		$\mathcal{R}_{2,b}$		comp.	
T_{max} (sec.)	0.50	5.00	0.50	5.00	0.50	5.00	0.50	5.00	0.50	5.00
$\underline{r}$ (%)	58.65	59.19	76.91	77.51	60.78	63.51	65.45	69.75	78.76	80.61
$\overline{r}$ (%)	38.22	38.65	57.49	57.73	44.46	47.89	47.87	53.22	59.20	61.29
σ_r	41.57	41.78	37.91	37.77	38.91	39.14	39.03	38.77	40.91	40.72
$\underline{p}$ (%)	100.00	100.00	60.55	60.48	91.70	91.16	90.25	89.21	78.12	84.57
$\underline{\delta}_p$	7.70	7.70	4.67	4.66	7.07	7.03	6.95	6.87	6.03	6.52
$\overline{p}$ (%)	100.00	100.00	71.93	71.95	90.55	90.53	88.18	87.65	83.09	87.75
σ_p	0.00	0.00	35.10	35.11	17.54	14.98	26.14	25.69	29.04	22.67
$\overline{\delta}_p$	39.74	39.74	25.80	25.80	35.28	35.91	30.87	31.08	29.69	33.52
$\underline{f}$ (%)	0.00	0.00	7.46	7.55	0.82	0.92	1.05	1.26	3.29	2.19
$\overline{f}$ (%)	0.00	0.00	9.27	9.35	2.03	2.15	1.11	1.32	4.66	3.52
σ_f	0.00	0.00	18.36	18.48	7.35	7.37	4.95	5.13	13.41	11.40
$\underline{e}$ (%)	6.64	6.58	3.58	3.50	5.56	5.20	4.95	4.37	3.17	2.87
$\underline{\delta}_e$	0.45	0.44	0.28	0.27	0.43	0.40	0.38	0.34	0.24	0.22
$\overline{e}$ (%)	5.81	5.74	4.26	4.09	6.21	5.90	5.85	5.33	4.13	3.82
σ_e	9.93	9.87	10.31	10.28	10.21	9.87	9.10	8.63	10.13	10.05
$\overline{\delta}_e$	0.45	0.44	0.45	0.45	0.43	0.40	0.55	0.50	0.43	0.41

Table 6.2: Rewrite-Based Confirmation: Fixed Domains, Unrolling, $\mid t \mid_{max} = 10000$

Significantly better results can again be achieved by a competition between the three different domains. Basically, the competition combines the high recall of $\mathcal{R}_1$ with the high precision of $\mathcal{R}_{2,a}$ and $\mathcal{R}_{2,b}$: it achieves $\underline{r} = 80.61\%$ and $\underline{p} = 84.57\%$. Here, a slightly different competition mode than in the other experiments is used: *disjunction with vetoing*. This means that a component is retrieved if the associated proof task is rewritten to *true* in at least one filter but not rewritten to *false* in any of the other filters. The integration of this vetoing mechanism turned out to be crucial. Without vetoing, a slightly better recall is achieved ($\underline{r} = 80.99\%$) but since none of the involved filters is precise, the already relatively high numbers of spurious "proofs" add up even more by the competition and the precision of the combination degrades ($\underline{p} = 60.69\%$).

Overall, the results in Tables 6.1 and 6.2 show that counterexample-based filters and their combinations allow some very interesting recall/precision-tradeoffs.

These tradeoffs are governed by the choice of the counterexample domain, the interpretation of "don't know"-results (i.e., accept or reject) and the mechanism used to implement competition between multiple filters. Moreover, using a rewrite-based implementation technique all these tradeoffs can actually be realized with relatively restricted computational resources. Even with only a single processor the average response times fall into a range between only 20 seconds and 5 minutes.

6.2 Proving over Finite *item*-Domains

The basic idea of the preceding section—fixing a particular *item*-domain and using properties of this domain to achieve a stronger rewrite system and thus more efficient rejection filter—can also be applied in the case of theorem proving. Here, the properties of the domain are encoded as additional axioms. The proof task is no longer checked for validity, but for satisfiability under the extended set of axioms.[3] As in the rewrite-based case, the component can be rejected if a proof for the (negated) task can be found and is passed through if the prover fails.

Of course, this is still an unsound filter but it is recall-preserving as long as the additional axioms are conservative extensions of the original background theory. The additional axioms need not to be very specific; in the extreme case, they may be missing completely such that the prover attempts a pure refutation which is of course recall-preserving. The basic assumption of this approach is that the additional axioms enable the prover to find enough refutations in a very short time to justify the additional proof attempts.

6.2.1 Axiomatizing Finite Domains

For the "proof-of-counterexamples" experiments I used the same substructures as in the rewrite-based case. Since none of the provers has built-in support for finite domains, the various *item*-domains need to be axiomatized explicitly.

A finite datatype T can in principle be axiomatized just by enumerating its elements in a fashion similar to the cases theorem of generated datatypes (cf. page 5.2.2), viz.

$$\forall y \cdot y \in T \Leftrightarrow \bigvee_{i=1,\ldots,n} y = c_i \qquad (*)$$

for some constants c_i. However, there are a number of issues with this schema. First, it does not adequately capture the size of the domain as the constants need

[3]Hence, it must be negated before the proof is attempted. However, if the prover works in a refutational style, it negates the task internally once again, before the actual proof is attempted.

not be distinct. This may weaken the rejective power of the filter. Second, it is not necessary to name the domain elements explicitly. While this makes them visible and thus accessible for use in other lemmas, it might also interfere with prover internals as for example term orderings and clause indexing. Alternatively, the domain elements can also be introduced by an existential quantifier. Finally, the bi-implication does not take full advantage of the availability of sorted quantifiers. The "$\Leftarrow$"-direction can be eliminated by folding-in the sort information for the constants:

$$\begin{aligned}\forall x \cdot x \in T \Rightarrow \bigvee_i x = c_i &\equiv \forall x \bigwedge_i (x = c_i \Rightarrow x \in T)\\ &\equiv \forall x \bigwedge_i (x = c_i \Rightarrow c_i \in T)\\ &\equiv \mathit{true}\end{aligned}$$

For the experiments described in the following section I thus used two different modified versions of (*). The first schema explicitly

$$\bigwedge_i c_i : \mathit{item} \wedge \bigwedge_{i \neq j} c_i \neq c_j \wedge \forall x : \mathit{item} \cdot \bigvee_i x = c_i \tag{6.1}$$

introduces n different constants while the second schema

$$\exists y_1, \ldots, y_n : \mathit{item} \cdot \bigwedge_{i \neq j} c_i \neq c_j \wedge \forall x : \mathit{item} \cdot \bigvee_i x = y_i \tag{6.2}$$

uses existential quantifiers. A third version is applicable only for the case of $|\,\mathit{item}\,| = 1$ where 6.2 simplifies to

$$\exists y : \mathit{item} \cdot \forall x : \mathit{item} \cdot x = y$$

This can be strengthened to

$$\exists y : \mathit{item} \cdot \mathit{true} \wedge \forall x, y : \mathit{item} \cdot x = y \tag{6.3}$$

i.e., the domain inhabitation and domain collapse axioms can be separated from each other. This variant is semantically equivalent but since the domain collapse axiom $\forall x, y{:}\mathit{item} \cdot x = y$ is now fully universally quantified, it is directly applicable more often (i.e., showing that the *item*-domain collapses can be done in a single step and does not entail commutativity and transitivity of equality, as required in 6.2). Hence, the search space may have a completely different structure.

6.2.2 Experimental Results

In the experiments I applied only the SPASS-prover which found the highest number of fast proofs in specification matching (cf. Chapters 7–9). I also restricted

myself to the best variant (i.e., with domain-specific simplifications and lemma selection following the *lemmas*-heuristic) in order to investigate its suitability as additional rejection filter. Without these techniques only far worse results can be achieved.

Pure Refutations

In a first round of experiments I did not use any specific counterexample axiomatization but tried out a pure refutational approach, i.e., the negated tasks where fed into prover. This mimics the general rewrite-based rejection filter described in Section 5.5. The motivation for these experiments is to determine whether the theorem provers applied in the final confirmation filters can also be used to implement a guaranteed recall-preserving rejection filter which could replace or improve the rewrite-based approach.

	pred. encod.		term encod.		comp.	pipe.
T_{max} (sec.)	0.50	5.00	0.50	5.00	5.00	5.00 / 10.00
$\overline{T}_{task}$ (sec.)	0.36	2.97	0.32	2.77	2.68	4.03
σ_T	0.21	2.33	0.22	2.41	2.40	6.21
$\overline{T}_{query}$ (sec.)	42.72	353.91	38.03	329.45	318.95	479.17
σ_T	10.04	106.36	10.44	108.99	107.93	267.62
$\underline{r}$ (%)	100.00	100.00	100.00	100.00	100.00	100.00
$\underline{p}$ (%)	19.64	23.66	22.33	24.60	26.00	38.90
$\underline{\delta}_p$	1.51	1.82	1.72	1.89	2.00	3.00
$\overline{p}$ (%)	19.61	22.21	20.94	22.49	23.38	32.20
σ_p	25.92	26.79	25.84	26.65	26.94	28.00
$\overline{\delta}_p$	1.98	2.55	2.18	2.58	2.80	6.84
$\underline{f}$ (%)	61.04	48.13	51.92	45.75	42.47	23.44
$\overline{f}$ (%)	60.10	47.78	51.46	45.63	42.47	24.62
σ_f	21.60	20.59	21.19	20.42	19.69	17.02
$\underline{red}$ (%)	33.90	45.14	41.84	47.20	50.06	66.62

Table 6.3: Proof-Based Rejection: Proof of Contradiction

Table 6.3 shows a clear—but unfortunately negative—answer to these questions: in general, ATPs are not suitable to replace rewriting using the unrolling strategy as rejection filters. The first five columns contain the results using the term and predicate encoding, respectively, to represent sorts, each with two different timeouts as well as a competition between the two representations.

The results for $T_{max} = 0.5$ secs. clearly show that ATPs are not well suited as *fast* rejection filters. The fallout rates are 15–25 %-points higher as in the rewrite-

based filters (cf. Table 5.2). In fact, using the standard predicate encoding, SPASS is with this timeout barely able to improve the precision of its input. If the non-matches already detected by the preceding simplification phase[4] are discounted, the precision leverage drops down to $\delta_p = 1.12$. The situation slightly improves with increased timeouts, especially if the predicate encoding is used, but even with $T_{max} = 5.0$ secs., the filter does not achieve the efficiency of the rewrite-based filter with $T_{max} = 0.5$ secs.. The fallout rates stay within the 45–50% range and neither a competition between both sort encodings nor a further increased timeout (not shown) bring a decisive improvement.

However, the numbers also show that following the pure refutational approach is still better that having no dedicated rejection filter at all. Even with $T_{max} = 5.0$ secs., the average response time per query remain with roughly 5–6 minutes quite reasonable. Moreover, only for three queries the additional effort spent in the rejection filter is higher than the realized savings in the confirmation filter. Hence, this setup accelerates the entire retrieval process. It is, however, unclear whether this approach can be scaled any farther since it is not known how many of the non-matches have no models at all and, consequently, what fraction of the theoretical maximum the rejected tasks already represent.

The last column in Table 6.3 contains the results for a pipeline comprising the two rewrite-based filters also shown in Table 5.2 and the SPASS-based filter using the term encoding. The two rewrite-based filters where allowed a total of 5 seconds while the prover twice as much time as before. Even with this setup, the pipeline's efficiency is very much dominated by the rewrite-based filters. Its combined reduction factor $\underline{red} = 66.62\%$ is only a small improvement over the 63.96% already achieved after rewriting; similar relations hold for fallout and precision because all filters are recall-preserving.

Proof of Counterexamples

In a second set of experiments I tested the effectiveness of the different counterexample axiomatizations over the same candidate domains as in the rewrite-based case. I only axiomatized the domain size but did not add further domain-specific axioms; doing this yields only slightly better results. Table 6.4 contains more results for these three domains as well as the competition over all variants. Here, I only used the term encoding which in the pure refutational case performed significantly better than the predicate encoding.

Although $\mid item \mid = 1$ shows a significant improvement over the pure refutational approach, the overall results are disappointing. For $\mid item \mid = 2$ and $\mid item \mid = 3$ the average fallout remains in the 45–50% range which means precision levels of approximately 25%. In contrast to the rewrite-based case, the larger domains thus do not induce stronger filters. In fact, for no domain the

[4]Remember that the simplifier used to generate the proof tasks does not apply the quantifier unrolling strategy.

Domain Size	$\|item\| = 1$			$\|item\| = 2$		$\|item\| = 3$		comp.
Axiomatization	sign.	exist.	univ.	sign.	exist.	sign.	exist.	-
T_{max} (sec.)	5.00	5.00	5.00	5.00	5.00	5.00	5.00	5.00
$\overline{T}_{task}$ (sec.)	2.32	2.51	2.53	2.84	2.82	2.83	2.78	2.23
σ_T	2.39	2.41	2.41	2.41	2.40	2.41	2.40	2.38
$\overline{T}_{query}$ (sec.)	275.66	305.86	301.5	338.07	335.46	336.52	331.02	265.22
σ_T	118.70	112.59	113.18	111.14	108.59	108.44	108.29	115.70
$\underline{r}$ (%)	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00
$\underline{p}$ (%)	30.26	26.87	27.48	23.86	24.20	23.89	24.49	31.68
$\underline{\delta}_p$	2.33	2.07	2.12	1.84	1.86	1.84	1.89	2.44
$\overline{p}$ (%)	25.87	23.88	24.34	22.06	22.36	22.01	22.53	26.69
σ_p	27.54	27.15	27.35	26.57	26.62	26.53	26.72	27.69
$\overline{\delta}_p$	3.45	2.93	3.03	2.53	2.54	2.50	2.56	3.61
$\underline{f}$ (%)	34.34	40.55	39.32	47.59	46.73	47.52	46.00	32.13
$\overline{f}$ (%)	35.01	40.82	39.63	47.42	46.85	47.37	45.82	32.83
σ_f	20.43	20.62	20.53	20.94	20.49	20.35	20.35	19.62
$\underline{red}$ (%)	57.15	51.75	52.82	45.61	46.36	45.67	47.00	59.08

Table 6.4: Proof-Based Rejection: Proof of Counterexamples

SPASS-based filters achieve even approximately the rejection rate of the respective rewrite-based implementations. Even for $|item| = 1$ the additional precision leverage gained from the finite domain is with $\Delta_p = 2.33{:}1.89 = 1.23$ rather small. Moreover, the larger domains detect to a large extent the same non-matches as the single-element domains. The competition between all variants thus yields only a small improvement over the best individual filter; the additional precision leverage of $\Delta_p = 2.44 : 2.33 = 1.05$ is neglectable compared to the 1.24-fold improvement achieved in the rewrite-based case.

The effect of the different axiomatizations is relatively small but reveals a surprising pattern. For $|item| = 1$, the signature extension is clearly superior (i.e., faster and more effective) while the existential and universal domain axiomatizations are almost equivalent. For larger domain sizes, however, this relation flips and the existential domain axiomatization yields slightly better results than the explicit signature extension. This effect is even more pronounced for $|item| = 3$ than for $|item| = 2$.

Chapter 7

The Retrieval Base Case

A setup analogue to the naive implementations of specification matching published previously [MW95b, MW97b, MMM94, MMM97, CJ92, JC94, MM91] serves as base case for NORA/HAMMR. Its essential purpose is to provide a reference or, to put it more sloppily, a "bottom line" for the more elaborated variants provided by NORA/HAMMR. It should help to identify the most effective or at least some worthwhile improvements and thus help to make deduction-based retrieval practical.

For the base case I first used the full set (i.e., without rejection filters) of unsimplified (except for the elimination of the propositional constants required by some provers) proof tasks together with the full set of axioms and lemmas and applied several provers in fully automatic mode to them. Since all provers are sound but very weak in detecting non-theorems, I actually restricted the experiment to theorems only and completed the statistics with "reject after timeout" entries for the non-theorems.

Table 7.1 summarizes the complete results for two different timeouts. Here, $\overline{T}_{task}$ and $\overline{T}_{valid}$ both refer to the average response times per task, but while $\overline{T}_{task}$ includes non-theorems, $\overline{T}_{valid}$ is restricted to the valid tasks actually attempted by the provers; hence, Σ_T is the total time spent by the provers. $\overline{T}_{proof}$ is further restricted to successful, non-trivial proofs, i.e., valid tasks which have been proven be the ATP but—in later chapters—not by any filter further up in the pipeline. In the base case, $\overline{T}_{proof}$ thus excludes the 390 tasks already reduced to *true* by elimination of the propositional constants. $\overline{T}_{query}$ is again the average response time for a query comprising 119 single tasks; $q_{0.25}$ and $q_{0.75}$ are the first and third quantile, respectively. These values indicate that for example using OTTER (in `auto2`) with a 90 seconds timeout 25% of the queries required less then 9725 seconds and 75% less than 10624 seconds, or to put it the other way round, that the "inner half" of the queries was solved between 9725 and 10624 seconds. The lower part of the table lists recall and precision as defined in Section 2.2; however, since all provers are sound, the precision is consistently 100%. I have thus also dropped the derived measures as precision leverage or error quota.

	OTTER				GANDALF		SPASS		SETHEO	
	auto 1		auto 2							
T_{max} (sec.)	1.00	90.00	1.00	90.00	1.00	90.00	1.00	90.00	1.00	90.00
$\overline{T}_{task}$ (sec.)	0.97	84.98	0.97	82.26	0.97	83.92	0.99	83.88	0.97	84.93
σ_T	0.14	19.67	0.14	24.74	0.17	21.94	0.05	22.06	0.16	20.32
$\overline{T}_{valid}$ (sec.)	0.79	51.31	0.76	30.46	0.76	43.14	0.93	42.83	0.78	50.98
σ_T	0.33	40.96	0.32	40.27	0.42	42.42	0.13	42.59	0.41	43.07
Σ_T	1443	94299	1391	55793	1404	79300	1703	78726	1441	93694
$\overline{T}_{proof}$ (sec.)	0.37	23.96	0.56	5.58	0.43	18.54	0.66	11.58	0.97	10.72
σ_T	0.04	23.80	0.23	9.38	0.29	27.45	0.17	19.45	0.02	17.74
$\overline{T}_{query}$ (sec.)	115.7	10112	115.3	9788	115.4	10281	117.9	9981	115.7	10107
σ_T	13.5	1798	13.6	1886	17.0	1858	3.7	1839	15.8	1824
$q_{0.75}$	119.0	10710	119.0	10624	119.0	10646	119.0	10625	119.0	10710
$q_{0.25}$	118.4	10399	118.3	9729	119.0	10290	118.5	10203	119.0	10393
# proofs	558	937	749	1274	448	1104	574	1052	612	852
$\underline{r}$ (%)	30.36	50.98	40.75	69.31	24.37	60.07	31.23	57.24	33.30	46.35
$\overline{r}$ (%)	11.49	34.65	22.52	57.21	5.14	50.51	11.85	54.33	13.44	29.38
σ_r	26.67	37.92	33.16	38.98	19.93	38.08	26.34	38.58	28.10	35.25
$\overline{p}$ (%)	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00

Table 7.1: Base Case

The main result of the base case is very clear from the table:

- With a very short timeout of for example a single second, the naive implementation produces "results-while-u-wait" but only at the expense of insufficient recall rates.
- With a relatively long timeout of for example 90 seconds, all provers are already able to solve a significant number of tasks but only at the expense of extremely long overall response times.
- Proof times are short (relative to the timeout) on average but a significant number of tasks is solved with long proofs only.

Effectively, the responsiveness of such a naive retrieval tool is spoiled by the large number of non-matches (i.e., non-theorems) which drown the prover. The effect is still tolerable for short timeout but becomes prohibitive before sufficient recall rates are achieved. Moreover, the response times per query are skewed to the right and the arithmetic mean is closer to the first quantile and usually below the median—that is, most queries take longer than the average. Similarly, even

significant differences between the provers have only barely visible effects on the overall response times (e.g., the two different autonomous modes of OTTER).

In some more detail, the provers achieve overall recall rates between 25% and 40% within a single second and between 45% and 70% for 90 seconds. However, these numbers become less impressive if the 390 (or approximately 20%) trivial tasks are subtracted which are already reduced to *true* by the elimination of the propositional constants. For both timeouts, OTTER's `auto2`-mode comes off first.

Table 7.1 also shows that none of the provers performs consistently well for all the queries. The standard deviation of the query-oriented recall average is generally very high and the document-oriented average is consistently higher than the query-oriented average. This counter-intuitive situation is caused by a small number of queries with large match sets but only simple tasks which in turn are almost all solved by the provers. However, this effect becomes less dominant with increasing timeouts and the query oriented-recall average grows much faster than the document-oriented average although they never really converge.

A closer inspection of Table 7.1 also reveals some unexpected results. The first surprise is the bad performance of the CASC-winner of the last two years, GANDALF, for very short timeouts. This can be explained by the fact that the time slices become too small to find anything else than the most trivial proofs. For longer timeouts, the situation improves but it does not run up to OTTER's `auto2`-mode. Even if OTTER is trimmed down to the average time allocated for each of GANDALF's strategies (i.e., roughly 13 seconds because the version used in the experiments implements 7 strategies),[1] GANDALF comes off no better: OTTER still solves 1194 tasks.

A second surprise is that OTTER's in general highly incomplete second autonomous mode is so much better than the standard autonomous mode. It finds 40% more proofs than the standard mode and simultaneously requires 40% less time. Moreover, it "loses" almost no proofs. The standard mode only finds 6 proofs exclusively and is faster only for 8 other tasks. Hence, the restriction of the paramodulation inferences which is the main difference between the two modes does not harm, at least not for the given domain.

Also surprising is that SPASS, the only prover with built-in support for sorts does not perform better—after all, the problems are formulated in a sorted calculus. However, this is a slightly unfair comparison because SPASS solves "different" tasks than the other provers. The term encodings used to represent the sorts for these provers are optimized towards the specific situation but are less general than the standard relativation technique employed by SPASS. On the encoded variant, SPASS performs much better and solves 1293 tasks within 90 seconds. Section 9.5 contains a more detailed investigation of the effects of the two different

[1] This hypothetical setup is still in favor of GANDALF because usually only a subset of the implemented strategies is actually attempted.

sort handling techniques in combination with lemma selection.

The final surprise is that the only top-down prover used in the experiments, SETHEO, does not perform better. Ideally, such provers should be able to handle the large axiom sets much better than bottom-up provers, due to the goal-orientation built into the calculus. In practice, however, this advantage is usually offset by their weak equality handling. Since equality is the predominant predicate in the tasks and even worse, not a single proof task is equality-free due to the specification style, this disadvantage becomes decisive, resulting in SETHEO's significant lower recall rates.

Similar to the setup in the rejection filters, the different provers can be run in parallel, using competition to join their strengths and cancel out their weaknesses. Table 7.2 summarizes the results for two different competitions and three different timeouts.

	OTTER & SPASS			full competition		
T_{max} (sec.)	1.00	20.00	90.00	1.00	20.00	90.00
$\overline{T}_{valid}$ (sec.)	0.76	7.89	27.40	0.67	7.66	26.62
σ_T	0.32	8.61	38.75	0.43	8.71	38.38
Σ_T	1389	14497	50356	1229	14082	48920
$\overline{T}_{proof}$ (sec.)	0.56	3.52	6.42	0.49	3.27	7.38
σ_T	0.23	0.56	11.66	0.28	4.02	14.54
$\overline{T}_{query}$ (sec.)	115.2	2193	9743	113.9	2189	9731
σ_T	13.6	407	1897	18.5	413	1904
# proofs	757	1274	1348	817	1279	1375
$\underline{r}$ (%)	41.19	69.31	73.34	44.45	69.59	74.81
$\overline{r}$ (%)	23.06	61.90	65.37	27.54	62.51	68.81
σ_r	33.31	37.14	36.22	34.62	36.97	34.06
$\overline{p}$ (%)	100.00	100.00	100.00	100.00	100.00	100.00

Table 7.2: Base Case: Prover Competition

The rightmost columns of Table 7.2 show the results for a full competition between all systems. It results in a *speed-up* factor of $s = 1.14$ on the valid tasks (i.e., the total proof time of the *fastest* individual prover divided by the real time required by the competition).[2] However, this degrades to a neglectable factor of 1.0048 if it is based on all tasks. This in turn confirms the already mentioned effect that even significant differences on the valid tasks have only barely visible effects on the average overall response times. The *efficiency* (i.e., speed-up divided by

[2]Speed-ups cannot be calculated on the basis of solved tasks (i.e., proof times) because that would "punish" a prover for finding a long proof instead of failing.

the number of applied processors) of this competition is $e = 0.23$, again restricted to the valid tasks only. It can be increased at the expense of the speed-up if the competition is restricted to more complementary variants. In the experiments, it becomes maximal for a competition between the two fastest provers, OTTER (using `auto2`) and SPASS. This variant yields $s = 1.11$ and $e = 0.55$ (cf. also the leftmost columns of Table 7.2).

However, the above standard definitions of speed-up and efficiency do not take into account that the competition not only decreases the response times but also increases the total number of proofs found. The *true speed-up* must thus be calculated at the recall level of the *best* individual prover and with respect to its response time. This recall level—1274 proofs—is achieved by full competition with an individual timeout of slightly below 20 seconds which yields a total proof time $\Sigma'_T = 13914$ seconds and thus the true speed-up $s' = 4.01$ and the true efficiency $e' = 0.80$. Again, a higher (true) efficiency can be obtained with a restricted competition. The above OTTER-SPASS competition requires an individual time limit of 20 seconds to reach the same recall level, yielding a true speed-up of $s' = 3.85$. This is a *superlinear* speed-up because the true efficiency $e' = 1.92$ is greater than 1 which means that the combination is faster than the fastest individual prover even if the parallelism is simulated on a single processor, even for the worst possible order of execution. This observation shows that the test set is in general perceptive to such a strategy parallelism.

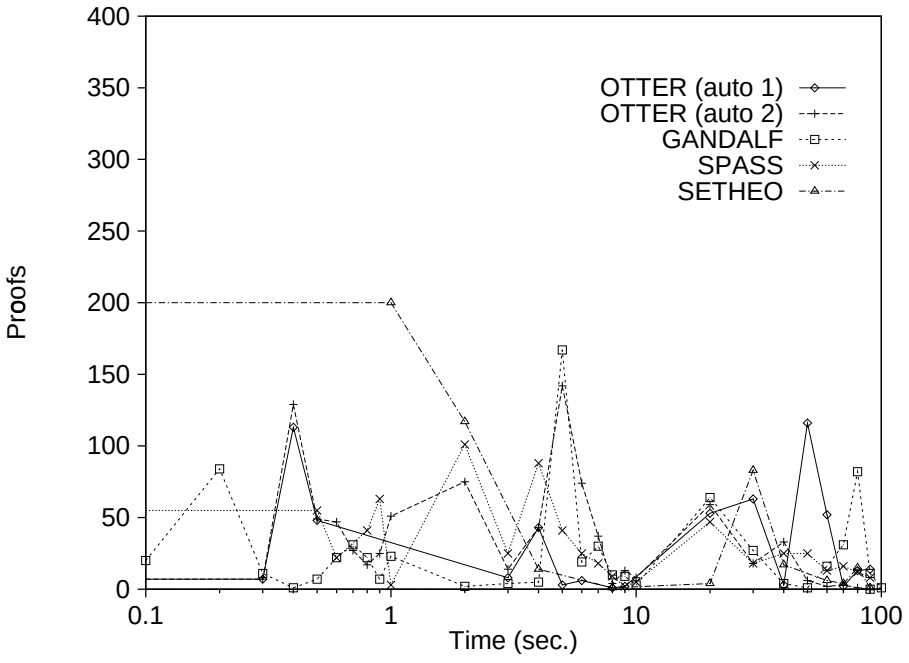


Figure 7.1: Base Case: Proof times

Together with Table 7.2, Figure 7.1 gives a more detailed overview of the

actually required proof times. Here, trivial proof tasks have been excluded and times have been rounded up and arranged for a better legibility. Table 7.2 and Figure 7.1 show that for any ATP the majority of the tasks which are provable at all have relatively short proofs: except for OTTER's `auto1`-mode, 50% of the respective non-trivial proofs are found in less than 5 seconds. However, a significant number of tasks is solved with long proofs only: the standard deviations on T_{proof} are usually substantially larger (up to 60%) than the means $\overline{T}_{proof}$. Moreover, the ATP-performances (i.e., the number of additional proofs found) generally fade out with increasing timeouts but repeated "performance bursts" disturb this trend. These bursts are caused by repeated internal reorganizations of the search space by the ATPs, e.g., pruning the set of kept clauses or iterative deepening steps. The performance bursts can vary with the values of various control parameters and are thus hard to predict from the outside. Hence, the choice of an optimal timeout which balances additional recall against increased response times is prover-dependent and requires careful performance monitoring by the reuse administrator.

From the results of the base case, three conclusions can be drawn which are very encouraging because they support the basic assumptions and design decisions of the NORA/HAMMR-system:

1. Current automatic theorem provers are mature enough to be employed in deduction-based retrieval. However, a non-trivial amount of preprocessing is necessary to achieve good recall rates.

2. Competition between different systems significantly increases the recall and decreases the response times.

3. Dedicated rejection techniques as those developed in Chapters 5 and 6 are required which filter out non-matches as fast as possible to prevent the provers from "drowning".

Table 7.3 further elaborates on the third conclusion. Here, only the tasks which survived a pipeline consisting of the two rewrite-based filters also used in Table 5.3 were fed into the provers. All results are based on $T_{max} = 5$ secs. for the rejection filter and $T_{max} = 90$ secs. for the provers.

The effects are already quite dramatic. The average response times per query drop by about 70%, despite the additional runtime occurred in the rejection filtering. Hence, almost the total reduction factor of the rejection filters carries through to the end of the pipeline. Even the worst-case response times are now better than the former averages. Conversely, the speed-up (i.e., in this case the ratio between the total response times) is consistently around 3.3. Nevertheless, the computational effort is still very high and the average response times are still too long by at least one order of magnitude. Hence, even if future hardware improvements are discounted, parallelization is essential to make deduction-based

	OTTER		GANDALF	SPASS	SETHEO	comp.
	auto 1	auto 2				
T_{max} (sec.)	5.00 / 90.00					
$\overline{T}_{task}$ (sec.)	26.26	24.56	25.72	25.96	26.08	24.56
σ_T	41.91	40.93	41.59	41.80	41.88	41.06
$\overline{T}_{query}$ (sec.)	3125	2923	3061	3090	3103	2923
σ_T	1840	1727	1810	1829	1834	1739
T_{max}	10076	9720	9827	9921	10013	9799
$q_{0.75}$	4125	3997	4102	4125	4117	4018
$q_{0.25}$	1896	1825	1842	1842	1842	1825
# proofs	1250	1492	1375	1280	1250	1528
$\underline{r}$ (%)	68.01	81.18	74.81	69.64	68.01	83.13
$\overline{r}$ (%)	51.76	69.01	63.52	65.42	49.83	75.97
σ_r	37.95	35.79	36.92	33.45	39.03	31.01
$\overline{p}$ (%)	100.00	100.00	100.00	100.00	100.00	100.00

Table 7.3: Base Case + Rewrite-Based Rejection

retrieval practical and to scale it up to larger libraries. However, this is rather easy because only complete tasks must be dispatched to different processors—there is no need to break single tasks apart.

As a welcome side-effect of the pre-filtering, substantial improvements are also achieved for the recall. All provers are able to increase the number of proofs found by simplification (1131, cf. Table 5.3) significantly. The leverage factors are, however, quite different, ranging from an additional recall of 6.5% for OTTER (`auto1`) and SETHEO to 19.7% for OTTER (`auto2`). These improvements represent 119 to 361 additional proofs. If the leverage factors are based on the original results of the provers, it becomes clear that the weaker provers profit much more: SETHEO gains more than 46%, OTTER (`auto2`) still 17%, and the competition only 11%. Hence, the simplification pre-filter has a leveling effect. The difference between the best and the worst individual prover, respectively, drops (from 49.5% to 19.4%) but remains still significant.

Chapter 8

The Effect of Simplification

The most obvious improvement over the base case is to check *simplified* variants of the proof tasks instead of the original, "raw" versions. These simplified variants can for example be produced as a side effect of the rewrite-based rejection filters described in Chapter 5.

However, some precautions must be taken. The naïve assumption that a more simplified proof task should be easier to prove is not necessarily true. Several effects can be counterproductive:

- The simplification ordering may be at least sub-optimal and ultimately produce larger tasks, e.g., more clauses.
- The effect of the simplification may become visible only in conjunction with other preprocessing techniques, e.g., lemma selection.
- The simplification may interfere with prover strategies and heuristics (e.g., orderings and weights) and may thus rearrange the search space arbitrarily.

Unfortunately, the effects can hardly be quantified in isolation because they usually appear in combination. Consider for example the simplification "expansion of defined symbols" which rewrites (besides others) all $<$-literals into equalities and $\leq$-literals.[1] Hence, instead of a single clause $\{x < y, p(x, y)\}$ the simplified variant produces the two clauses $\{x \leq y, p(x, y)\}$ and $\{\neg\, x = y, p(x, y)\}$. Of course, completely different proofs may be derived from these clauses, depending, e.g., on the given lemmas or the automatically chosen ordering on the symbols. As usual, this uncertainty can be turned into an advantage using competition between the different variants.

A similar effect is caused by NORA/HAMMR's pipeline architecture. All advantages of simplification may already be preempted by the filters which precede the final ATP, e.g., by the simplifier/rewrite-based rejection filter itself. That is,

[1] Similar observations also apply to many other important rules, e.g., injectivity of constructor functions.

all or almost all tasks which would additionally become provable after simplification can already be simplified to *true*. However, this is not really counterproductive and thus not an issue of concern.

8.1 Applied Simplifications

For the simplification experiments I used the "raw" versions as base case and I refer to the preceding chapter for its discussion and especially to Table 7.1 for the precise numbers. On top of the base case I used four different levels of simplification. The first two, denoted as *first-order* and *equational*, respectively, in figures 8.1 to 8.4 are also used in the simplification-based rejection filter; for these the rewrite systems $\mathcal{R}_{FOL}$ and $\mathcal{R}_{EQ}$ (cf. p. 105) are used.

The next level, denoted as *unfolding*, expands defined symbols into their definitions, e.g., $<$ into $\leq$ and $\neq$, or $\neq$ into $=$. The justification for this simplification which can actually increase the task size significantly is obviously that

- fewer symbols induce smaller search spaces, and
- upfront unfolding saves proof steps.

The risk is obviously that the tasks become too large, upsetting the smaller search spaces due to the smaller signature, and—less obvious—that the provers may "undo" the unfolding step.

In any case, some care must be taken to make the *unfolding*-simplification work. Consider for example the two symbols $\neq$ and *member*, defined by the axioms

$$\forall x, y : item \cdot x \neq y \Leftrightarrow \neg x = y$$

and

$$\forall l : list, i : item \cdot member(l, i) \Leftrightarrow \exists l', l'' : list \cdot l = l' \frown [i] \frown l''$$

Both symbols are in fact definitional extensions of a suitable base logic (i.e., theory of lists) and the definitions can be considered as rewrite rules and can be used to eliminate the symbols if they are oriented from left to right. However, experience tells us (and experiments confirm it) that unfolding $\neq$ is generally a "good idea" while unfolding *member* is a "bad idea". Obviously, the reason is that each occurrence of *member* introduces two new bound variables. To prevent this proliferation of bound variables, NORA/HAMMR simply restricts unfolding to symbols which have quantifier-free definitions. More elaborate unfolding schemes

which take for example the number of symbol occurrences or the presence of additional lemmas into account are of course possible.[2]

The final level of simplifications, denoted as *domain*, performs unfolding and then applies the rewrite system $\mathcal{R}_{DOMAIN}$ which is constructed from the lemma library (cf. Section 5.2.1 and 5.2.2).

For simplicity, the simplifications are applied only to the proof task and not to the axioms and lemmas in the lemma library. However, since these can reasonably be expected to be given by the reuse administrator in an adequately simplified form, this is in general no severe omission.

8.2 Experimental Results

The different provers respond quite different to the simplifications. I thus first discuss the results of each of the provers investigated separately and in some more detail before I draw some general conclusions.

OTTER

OTTER's (more precisely, its `auto2`-mode) results in the simplification experiment exhibit some noteworthy peculiarities.

First of all, the different variants almost converge and the total improvement is rather small in the long run. For a timeout of 90 seconds, the domain-specific simplifications (which are the best variant) lead to a total of 1293 proofs which represents a meager 1.5% improvement over the base case. The other variants even result in a small net loss of proofs, ranging from 0.6% to 3.0%. Here, the equational simplifications yield the worst result. This is most likely a consequence of substituting the variables throughout the task which obviously increases the number of non-variable term positions to be explored by the restricted paramodulation of the `auto2`-mode.

Second, unfolding predicate definitions does not pay for OTTER. Except for a small gain in the sub-second range which is due to the higher number of tasks rewritten to *true*, unfolding does not improve much on the pure equational simplification. That is, OTTER's built-in ordering already handles the definitional extensions appropriately.

Nevertheless, in general simplification pays even for OTTER and it pays especially for the shorter timeouts between 1 and 5 seconds. Here, the domain-specific simplifications represent an 27.6% and 14.8%, respectively, improvement over the

[2]For example, if the task contains only a single occurrence of *member* and the lemma database contains no additional (beyond the definition) lemmas about *member*, it can easily be unfolded. Additional lemmas change the situation because then unfolding destroys the applicability of a lemma. To maintain applicability, all lemmas need to be unfolded, too, which leads to the proliferation of bound variables and thus to a further increase in proof task size.

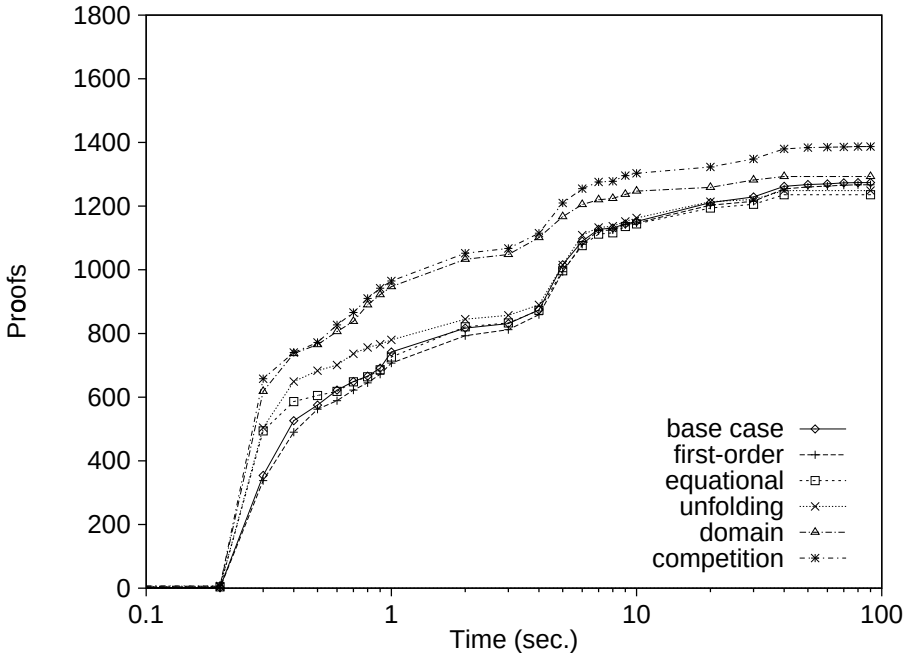


Figure 8.1: Simplification: Proofs over time—OTTER (`auto2`)

base case. But again, the purely syntactic simplifications fall short and run within a narrow margin almost parallel to the base case.

However, and this is the final peculiarity which sets OTTER apart from the other provers, a substantial improvement even in the long run can be achieved by competition. Running the base case and the domain-specific simplified variant in parallel yields 1387 proofs or an 8.9% and 7.3% improvement over the single variants, respectively. The speed-up of this combination is $s = 1.25$ ($e = 0.62$) but again (cf. p. 127) this does not properly reflect the full improvement. The true speed-up becomes apparent at a timeout of $T_{max} = 8.8$ seconds with $s' = 8.06$ and $e' = 4.03$, i.e., it is again superlinear. For shorter timeouts, however, the combination is almost entirely dominated by the domain-specific variant.

GANDALF

Although GANDALF is quite similar to OTTER as far as the basic calculus is concerned, its results in the simplification experiment are different in almost any aspect.

The most obvious difference is the much larger spread between the different variants. For the 90 seconds timeout, the domain-specific simplifications result in a total of 1436 proofs or an 30.1% improvement over the base case, compared to the 1.5% of OTTER. But unlike OTTER, GANDALF also profits from the weaker syntactic simplifications. Moreover, the profits reflect the perceived strength of

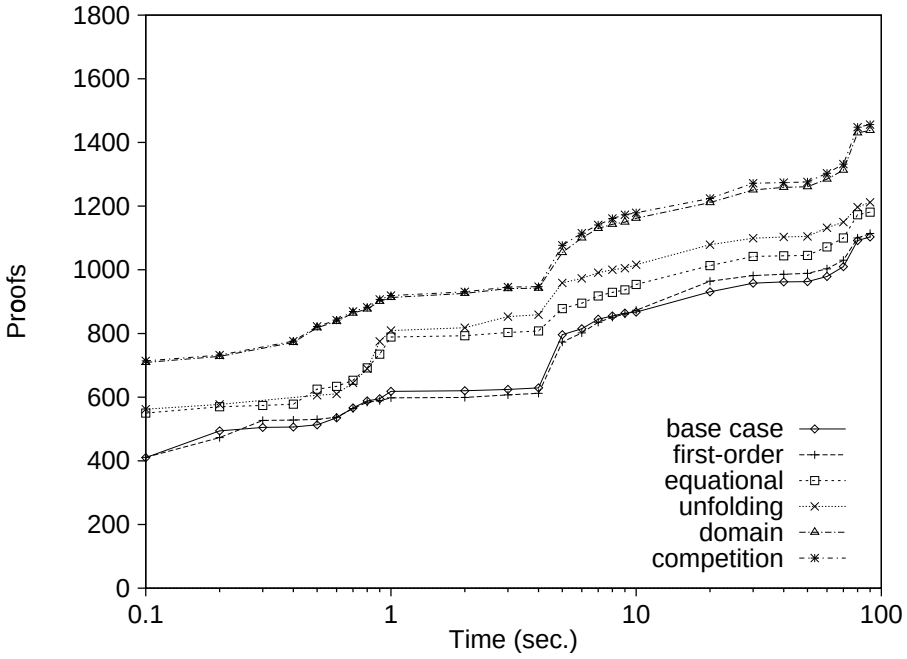


Figure 8.2: Simplification: Proofs over time—GANDALF

the respective simplifications, ranging from 1.0% for the first-order simplifications to 9.9% for unfolding the definitions. Also unlike OTTER, the improvements do not fade out over time—the curves for the different variants run almost in parallel.[3] The large difference between the equational simplifications and the base case, however, is counterintuitive at a first glance. In theory, a complete proof procedure should be insensitive to calculus-level permutations of the proof task; in practice, this is not always the case but SPASS shows that the sensitivity can be kept within reasonable bounds (cf p. 136). In GANDALF's case, equality *is* a calculus symbol and at least the last applied strategy is complete such that it *should* be less sensitive that it actually is. However, as OTTER's results show, incomplete proof procedures can be very sensitive to permutations and GANDALF usually starts with severely restricted and thus incomplete strategies and spends large parts of its total time with these strategies.

Finally, for GANDALF competition between different simplifications does not pay. The competition is dominated almost entirely by the domain-specific variant, especially for short and medium proof times, and yields only 20 or 1.4% occasional additional proofs compared to the domain-specific variant. This difference is so small that it can even be a consequence of timing inaccuracies and load differences between the different test runs (cf. p. 89).

[3]However, keep in mind that the proof times are obtained for a 90 seconds timeout and that—due to GANDALF's scheduling policy—hypothetical values for shorter timeouts cannot be "just read off" the graph.

SPASS

As Figure 8.3 shows, the experimental results for SPASS bear some resemblance to both OTTER's and GANDALF's cases.

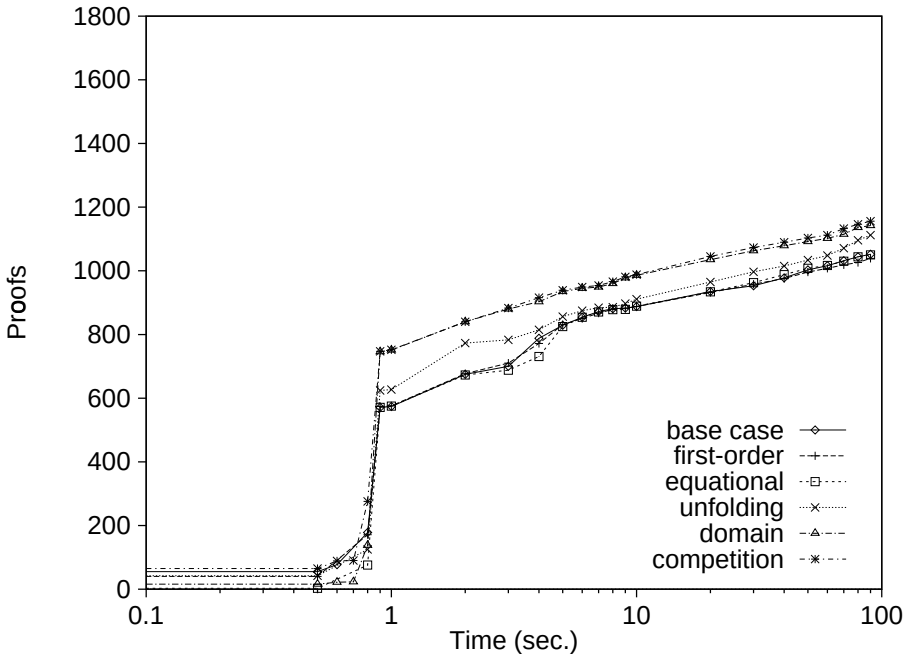


Figure 8.3: Simplification: Proofs over time—SPASS

As OTTER, SPASS shows only a relatively small spread between the variants. The domain-specific simplifications result only in a 8.6% improvement over the base case. Again, the purely syntactic simplifications yield virtually no improvement, independent of the chosen timeout. Moreover, a closer inspection of the solved tasks reveals that there are—in contrast to GANDALF—almost no differences between the "simple" variants (i.e., base case, first-order and equational simplifications, respectively): no variant solves more than 18 tasks (or less than 1%) exclusively.

However, unlike OTTER and similar to GANDALF, SPASS clearly profits from predicate unfolding. It already yields an 5.7% improvement over the base case and thus accounts for most of the domain-specific improvements. Similarly, SPASS does not profit from competition—here the result is even more dominated by the domain-specific variant than in GANDALF's case. This suggests that the small 1.2% gain is in fact a consequence of the biased competition timing.

All variants also show the typical "pre-saturation peak" at approximately 0.9 seconds, i.e., find an unusual number of proofs at that time. However, this is hardly surprising because the duration of the pre-saturation phase is determined only by the axiom set which has for this experiment not been changed between

the different variants.

SETHEO

Although SETHEO solves significantly fewer tasks, its results in the simplification experiment follow essentially GANDALF's pattern.

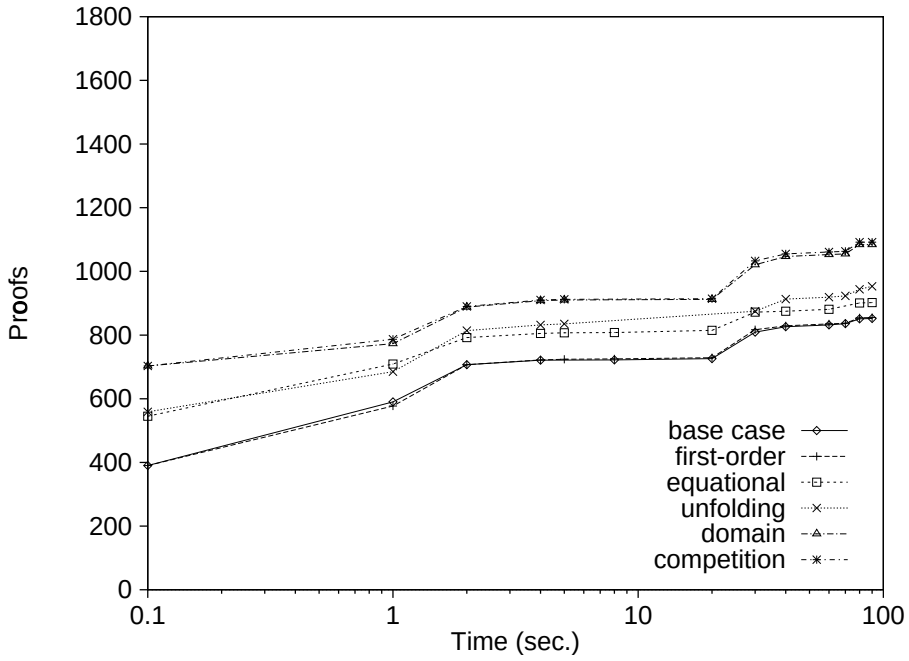


Figure 8.4: Simplification: Proofs over time—SETHEO

The domain-specific simplifications yield a 27.2% improvement (w.r.t. the 90 seconds timeout) over the base case which is roughly of the same order as in GANDALF's case. Similarly, the weaker simplifications result in smaller but significant improvements (unfolding: 11.9%, equational: 5.9%) which are again close to GANDALF's values. And again, the domain-specific variant subsumes the weaker variants such that competition yields no further improvements.

The distribution of proofs over time is remarkably regular—the plots of the different variants run almost in parallel. While this "parallelism" is a coincidence, most simplifications and in particular the equational and unfolding variant, do in fact work complementary to SETHEO's calculus. In both cases, the simplifications directly correspond to actual proof steps (i.e., extension steps) such that any proof which (in the base case) involves such a step is (in the simplified case) found one iterative deepening level earlier.

However, the almost perfect correlation of the base case and the first-order simplifications is not accidental. It is rather a consequence of the fact that

SETHEO uses exactly the same rewrite rules within its clausal normal form translator [Sch98].

General Results

The most general conclusion from the more detailed, prover-specific results discussed above is that *simplification pays*, i.e., the additional effort put into the extra simplification-phase is more than offset by a higher number of proofs and shorter overall response times. The time improvements over the base case (cf. tables 8.1 and 8.2) range in the best case from 1.5% for OTTER to 30.3% for GANDALF; the speed-ups (i.e., Σ_T for the base case divided by Σ_T for the best variant plus simplification time) vary between 1.08 for OTTER and 1.52 for GANDALF, again with significantly higher true speed-ups. The average proof times increase for most provers, with OTTER as the only exception. This increase is caused by the higher number of "complicated" tasks solved but it is not directly correlated to the gain in recall. The response times per query, however, remain essentially unchanged, i.e., they are still dominated by the effort spent on trying to prove mismatches.

Unfortunately, there are two catches.

1. Purely syntactic simplifications (i.e., constrained to the calculus adequately handled by the respective prover) do not entail much improvement.

2. Incomplete proof procedures are very sensitive to any kind of proof task manipulation; in particular, OTTER's `auto2`-mode fails to find proofs for simplified task variants quite often, even if it solves the respective base case.

Fortunately, both catches can be overcome, although the solutions incur substantial infrastructural costs.

The solution for the first catch is obviously to employ domain-specific simplifications, too. These can be implemented efficiently by compiling the lemma library into a domain-specific term rewrite system as described in Chapter 5; this approach takes also care of the problem that the simplification phase has to be updated periodically to keep track with evolving libraries. The effectiveness of such a pre-compiled, domain-specific simplifier can also be seen from the fact that the different provers need in the base case already between 1.5 (OTTER) and almost 5 seconds (GANDALF, SPASS) to achieve the same number of proofs (792) than the simplifier (without the unrolling strategy) within 0.25 seconds.[4] Nevertheless, for larger timeouts, every prover can improve on the results of the unrolling strategy: on the domain-specifically simplified variants, the provers solve between 11 (SPASS) and 307 (GANDALF) or 1.0%–27.1% more tasks than the simplifier using unrolling (cf. Table 5.3)

[4]SETHEO does not solve that number of tasks at all.

	OTTER			GANDALF		
mode	best	comp.	pipe.	best	comp.	pipe.
T_{max} (sec.)	90.00	90.00	5.00 / 90.00	90.00	90.00	5.00 / 90.00
$\overline{T}_{valid}$ (sec.)	28.13	24.32	-	28.27	27.47	-
σ_T	40.38	37.99	-	38.53	38.13	-
Σ_T	51698	44700	-	51953	50488	-
$\overline{T}_{proof}$ (sec.)	4.18	5.75	-	21.70	21.39	-
σ_T	6.62	9.70	-	28.84	28.70	-
$\overline{T}_{query}$ (sec.)	-	-	2865	-	-	2905
σ_T	-	-	1683	-	-	1674
# proofs	1293	1387	1563	1438	1456	1612
$\underline{r}$ (%)	70.35	75.46	85.04	78.24	79.22	87.70
$\overline{r}$ (%)	61.80	66.48	74.76	74.28	75.27	82.36
σ_r	36.11	34.41	32.37	29.75	28.98	26.47
$\overline{p}$ (%)	100.00	100.00	100.00	100.00	100.00	100.00
	SPASS			SETHEO		
mode	best	comp.	pipe.	best	comp.	pipe.
T_{max} (sec.)	90.00	90.00	5.00 / 90.00	90.00	90.00	5.00 / 90.00
$\overline{T}_{valid}$ (sec.)	38.13	37.63	-	40.44	40.11	-
σ_T	42.28	42.12	-	42.90	42.81	-
Σ_T	70090	69169	-	74332	73721	-
$\overline{T}_{proof}$ (sec.)	15.67	16.00	-	16.99	16.91	-
σ_T	21.98	22.22	-	21.25	21.01	-
$\overline{T}_{query}$ (sec.)	-	-	3050	-	-	3006
σ_T	-	-	1801	-	-	1712
# proofs	1142	1156	1342	1084	1092	1405
$\underline{r}$ (%)	62.13	62.89	73.01	58.98	59.41	76.44
$\overline{r}$ (%)	61.33	63.88	70.53	40.65	41.66	58.11
σ_r	37.71	37.27	32.64	37.72	37.82	38.81
$\overline{p}$ (%)	100.00	100.00	100.00	100.00	100.00	100.00

Table 8.1: Simplification: General Results

Moreover, unrolling does not preempt proving the simplified task variants. Hence, the full benefit of simplification becomes apparent only in combination with a rejection filter, as already described for the base case on page 128; in Table 8.1, the rightmost column under each prover heading contains the data

corresponding to Table 7.3. This setup leads to an additional 62 (SPASS) to 237 (GANDALF) proofs, or improvements between approximately 4.8%-points (SPASS, OTTER) and 17.3%-points (GANDALF); these improvements roughly correlate with the effect of the domain-specific simplifications without rejection filter. The additional proofs represent increased domain-oriented recall levels of approximately 4% (SPASS, OTTER) to 12.9% (GANDALF); the recall level peaks at 87.7% for GANDALF. Similarly increased recall levels and at the same time slightly smaller standard deviations are also observed for the query-oriented averages; again, GANDALF delivers the best results with $\overline{r} = 82.4\%$ ($\sigma_r = 26.5\%$).

The solution for the second catch is obviously to employ competition between the different simplification variants. However, the discussion of the individual provers has shown that this pays only for OTTER. More uniform improvements can be achieved if the competition is extended to the different provers. Table 8.2 summarizes these results; similar to Table 7.2 it shows a restricted competition between OTTER and SPASS as well as the full competition between all systems. For each prover, only the best individual variant (cf. Table 8.1) has been used.

	OTTER & SPASS			full competition				
T_{max} (sec.)	1.00	20.00	90.00	1.00	20.00	90.00	5.00 / 20.00	5.00 / 90.00
$\overline{T}_{valid}$ (sec.)	0.67	6.80	24.71	0.53	5.76	20.47	-	-
σ_T	0.35	8.64	38.16	0.46	8.23	35.58	-	-
$\overline{T}_{proof}$ (sec.)	0.65	3.11	7.51	0.55	2.88	8.83	-	-
σ_T	0.20	3.66	15.00	0.23	3.54	18.78	-	-
$\overline{T}_{query}$ (sec.)	113.9	2176	9743	111.8	2160	9636	755.2	2758
σ_T	15.3	415	1897	20.8	423	1927	438.9	1660
# proofs	880	1318	1393	1052	1411	1498	1564	1613
$\underline{r}$ (%)	41.19	71.71	75.79	57.24	76.77	81.50	85.09	87.76
$\overline{r}$ (%)	39.78	66.26	71.50	45.92	73.44	78.27	80.09	82.92
σ_r	35.42	35.24	32.98	37.53	31.28	27.76	28.18	24.94
$\overline{p}$ (%)	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00

Table 8.2: Simplification: Prover Competition

At $T_{max} = 90$ seconds, the speed-up factor of the full competition over the fastest individual prover (OTTER) on the valid proof tasks is $s = 1.37$; the efficiency of this competition is $e = 0.34$. The restricted competition between OTTER and SPASS yields $s = 1.14$ and $e = 0.57$ under these conditions.

In both cases, the simplified proof tasks induce larger speed-ups than the original tasks; moreover, for both competition variants the relative speed-ups of the simplified task over the original tasks (i.e., the respective total proof times of the competition divided by each other) even exceed the plain speed-up ratios.

Hence, simplification significantly amplifies the effects of competition.

As in the base case competition does not only speed up the retrieval process but also improves the recall. The full competition, however, is already almost dominated by the best prover (GANDALF) and yields only 60 or 4.2% more proofs which represents an additional 3.26%-points in the recall level. These numbers shrink down even more if the prover competition is compared to variant competition with GANDALF; here, the bottom line is a 2.28%-points increase in the recall level.

Nevertheless, the combination of faster response times and more proofs adds up to significant higher true speed-ups. The competition achieves GANDALF's recall level—1438 proofs—with a timeout of 40 seconds which yields a true speed-up of $s' = 2.27$ but a still sublinear true efficiency of $e' = 0.69$. In this case, a true superlinear speed-up cannot be achieved by the restricted competition between OTTER and SPASS because it solves less tasks than GANDALF alone.

Somewhat surprisingly, however, the prover competition looses most of its advantages over single provers in terms of recall when it is combined with the rewrite-based rejection filter. This becomes particularly apparent in GANDALF's case (which dominates the competition) where the difference almost completely disappears. In terms of response times, however, the competition maintains a substantial advantage of at least 3.7% or approximately 2 minutes per query on average.

Chapter 9

The Effect of Lemma Selection

The supply of axioms and lemmas is crucial to automated theorem proving—a single missed key lemma can make a proof much harder or even impossible. Unfortunately, the naive solution to include *all* available lemmas does not work because it induces too large search spaces. Hence, a pre-selection is necessary which selects only such axioms 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. Of course, this selection can be heuristic only: deciding whether a lemma shortens a proof is at least as hard as finding the proof in the first place.

In this chapter I describe the lemma selection process in NORA/HAMMR. Section 9.1 discusses the general approach of signature-based selection heuristics. The applied heuristics require meta-level information (e.g., the theory hierarchy) which expresses the human insights into the custom logic. Section 9.2 thus briefly describes the specification language used to provide this meta-level information. Section 9.3 discusses automatic generation of axioms from the meta-level information as additional approach to cut the number of required lemmas further down. Section 9.4 describes the basic selection algorithm and its various refinements implemented in NORA/HAMMR. Finally, Section 9.5 contains the experimental results for the different selection heuristics and provers.

9.1 Signature-Based Heuristics

Heuristics for axiom and lemma selection (and the similar problem of parameter setting) are usually based on syntactic properties of the proof task, e.g., its signature [RS98], the number and structure of literals or clauses, or even complex feature vectors [FF98]. The heuristics either work fully functionally (i.e., the selection depends *only* on the proof task) or maintain a "hotlist" of successfully applied axioms and lemmas which is updated whenever a proof has been found and analyzed. NORA/HAMMR only uses fully functional, signature-based heuristics because these have two major advantages: they are independent of the

applied calculus and they can be run as pure preprocessing steps. Both aspects make it easier to exchange the applied ATP independent of the lemma selection mechanism.

The basic assumption of all signature-based heuristics is that, as a first approximation, only such axioms and lemmas are useful which describe properties of the symbols which actually occur in the proof task. However, this is only an approximation and requires some adjustments which will be described subsequently. Furthermore, it is in fact only a heuristic (even if it allows to determine a minimal set of axioms which makes a proof possible in theory) because one can always construct examples in which the introduction of a new symbol together with some lemmas allows an easier proof. Nevertheless, in practice, it has proven to be useful.

A closer look at the selection problem reveals that it actually consists of two similar and related subproblems:

1. Selection of the most appropriate variant of the *core axioms* which facilitate the proof at all.

2. Selection of promising additional *lemmas* which may shorten the proof.

The first selection subproblem is closely coupled with the translation step from the custom logic to the core logic (cf. Sections 4.1.2 and 4.3.3). It occurs whenever a type admits different but semantically equivalent core axiomatizations or "views". Lists for example admit two alternative views, the free datatype view with the base functions *nil* and *cons*, and the monoid view with the base functions *nil, singleton* and *append*. The choice of one view over another may influence the performance of the applied ATP: a prover with a built-in decision procedure for the free datatype view [NO80] may perform worse if the monoid view is chosen. In practice, however, there is often not much of a choice. Usually, one view is hardcoded into the translation step. NORA/HAMMR for example translates VDM-SL's sequences into the *nil-cons*-fragment; using the monoid view (in which *cons* is a defined function) would lead to unnecessarily large rule sets. Hence, this selection subproblem is not treated in NORA/HAMMR and only the second subproblem remains as the proper lemma selection problem.

9.2 Hierarchic Specifications

For a signature-based selection heuristic, a flat, unstructured list of rules is not very suitable because it cannot convey any information about the (implicit) semantic relations between the symbols and rules. Generally, the more meta-level information is expressed explicitly, the better selection strategies are possible. NORA/HAMMR thus uses hierarchically ordered specifications to organize a rule library, similar to the approach of W. Reif and G. Schellhorn [RS98]. The applied

theory specification language is derived from the notation used in the ISABELLE-system [Pau94] but NORA/HAMMR should not be considered as a full-fledged logical framework. The main conceptual difference is that it does not support the specification of new logics but only conservative and inductive extensions of order-sorted FOL. A more detailed description of the theory specification language can be found in [Wei98].

The theory specification language distinguishes between *signatures* and *theories*. A signature is a named collection of classes, types, and predicate and function symbols, similar to signatures in algebra or definition modules in programming languages. The simple example signature `ItemVDM`[1]

```
signature ItemVDM = FOLneq +
  types  "Item" : equal
end
```

introduces a type `Item` as member of an already defined type class `equal`. Both types and type classes (ultimately) represent object collections. However, types are object-level constructs; they are reflected in the proof tasks as sorts. Type classes are pure meta-level constructs; they are provided only for a convenient formulation of polymorphism.[2] Type classes may thus be reflected only indirectly in the (first-order) proof tasks in the form of their ground instances, i.e., monomorphic types. The hierarchical structure is introduced by imports: `ItemVDM` imports and thus depends on `FOLneq`. For signatures, import just amounts to textual inclusion. Hence, the signature specification

```
signature ListVDM = ItemVDM +
  types  "List" : equal
  consts "[]" : "List"
                  (0);
         "::" : "[Item; List] => List"
                  (infixr 2 45)
end
```

is equivalent to

```
signature ListVDM = FOLneq +
  types  "Item" : equal;
         "List" : equal
  consts "[]" : "List"
                  (0);
         "::" : "[Item; List] => List"
                  (infixr 2 45)
end
```

[1]The examples are taken in slightly adapted form from the rule library used in the experiments.

[2]The current implementation of NORA/HAMMR does not fully support type classes and overloading. The examples do thus not use polymorphic predicates or functions.

where the import of `FOLneq` must be resolved recursively.

For predicate and function symbols, a signature must specify types and some additional parsing information. The example signature `ListVDM` declares the two constructors *nil* (with external representation `[]`) and *cons* (with external representation `::`) of the type `List`. Their respective types `List` and `[Item; List] => List` are built up from the types declared earlier and the type constructors for product (`[_;_]`) and function type (`=>`), respectively, which are pre-defined in the meta-theory. The parsing information (mandatory arity, optional fixity and priority) is used to instantiate a generic term parser.

A theory is simply a collection of named axioms and lemmas, i.e., FOL-formulae. It is thus similar to an implementation module. The example theory `ListAxioms`

```
theory ListAxioms : ListVDM =
  axioms
    Nil     "all val L : List . all val I : Item . ~ [] = I :: L";
    Surj    "all val L : List .
               L = []
             | (ex val K : List . ex val I : Item . L = I :: K)";
    ConsInj "all val L : List . all val K : List .
              all val I : Item . all val J : Item .
               I :: L = J :: K --> I = J & L = K";
    Finite  "all val L : List . all val I : Item . ~ L = I :: L"
end
```

contains the four axioms `Nil`, `Surj`, `ConsInj` and `Finite` which sufficiently characterize finite lists over the `Item`-type. Lemmas (of which `ListAxioms` contains none) are assumed to be logical consequences of the axioms. Although this is currently not checked by the NORA/HAMMR-system, it is exploited by the selection mechanisms (cf. Section 9.4). A theory can be constrained or "typed" by a signature; in the example, `ListAxioms` is declared as an instance of of `ListVDM`. The constraint automatically imports the constraining signature into the theory.

Both types and type classes can be ordered hierarchically. Again, a type-subtype relation is directly reflected in the proof tasks in the form of a respective sort-subsort relation (or axiom) while the class ordering is only used of a fine-grained control of polymorphism, as in ISABELLE [Pau94] or Haskell [NS91, NP95]. In the example, the type `List` could alternatively be axiomatized using the two subtypes `Nil` and `Cons`:

```
signature ListSubsortedVDM = ItemVDM +
  types  "List" : equal;
         "Nil"  < "List";
         "Cons" < "List"
  consts "[]" : "Nil"
                  (0);
         "::" : "[Item; List] => Cons"
```

```
                    (infixr 2 45)
end
```

The new subtypes can then be used for an optimized axiomatization:

```
theory ListSubsortedAxioms : ListVDM = ListSubsortedVDM +
  axioms
    Disjunct "all val N : Nil . all val C : Cons . ~ N = C";
    SurjNil  "all val N : Nil . N = []";
    SurjCons "all val C : Cons . ex val L : List . ex val I : Item .
                C = I :: L";
    ConsInj  "all val L : List . all val K : List .
                all val I : Item . all val J : Item .
                  I :: L = J :: K --> I = J & L = K";
    Finite   "all val L : List . all val I : Item . ~ L = I :: L"
end
```

Here, the `Nil`-axiom has bin replaced by a disjointness axiom which can be handled more efficiently in sorted calculi. More importantly, the troublesome non-horn `Surj`-axiom has been replaced by two separate versions for the two subsorts.

This core language can then be extended by meta-level constructs to specify for example operator axioms for associativity and commutativity or generator functions for freely generated datatypes.

9.3 Generating Axioms

The most effective way to reduce the number of rules in the proof tasks is to build the rules directly into the calculus. This idea dates back to the very beginnings of automated theorem proving [Plo72] and has been applied in many variants, e.g., paramodulation, theory unification, or integrating decision procedures.

Building-in a theory, however, also requires marking function or predicate symbols with the built-in property. The simplest solutions rely on purely syntactic conventions, e.g., naming schemas or the syntactic structure of the axioms. OTTER for example interprets every binary predicate symbol whose name starts with `eq` as a instance of the equality predicate and the SCAN-IT tool which is a preprocessor for the theorem prover PROTEIN [BF94, Bau96] uses pattern matching to identify instances of particular axiom schemas. More elaborate solutions use the FOL-semantics of the rule set. The TOPS tool [Roa97] which is part of the Meta-Amphion system [LV95, RVL97, LV97] uses a theorem prover to show that a set of rules is a logical consequence of a theory and can thus be replaced by a decision procedure for that theory.

Tools as TOPS are very general but they have the disadvantage that they first force the reuse administrator to formulate the axioms in FOL before they then spend considerable effort on identifying the concepts behind these FOL-rules.

NORA/HAMMR uses a different approach. The specification language for the rule library provides syntactic means to uniformly mark symbols with commonly built-in properties. These are then interpreted in a prover-specific way which means that adequate FOL-axioms are generated for ATPs which do not support that built-in. While this requires a modification of the system kernel whenever the set of supported built-in properties changes, it has some decisive advantages. Obviously, it makes reuse administration easier and more reliable: if rules are generated automatically, one source of errors is eliminated.[3] Moreover, it also separates the specification of a generally applicable rule library from specific requirements of a particular calculus (which can be generated). This also makes the integration of a new prover into the system easier. The biggest advantage, however, is that the rule selection can be more efficient because the library contains less rules to select from. For example, without generating axioms, signature axioms would be selected even for provers which rely on term encodings.

Equality Axioms

Theorem provers which are based on an equality-free calculus (e.g., SETHEO) usually require an explicit axiomatization of the equality predicate.[4] In principle, Birkhoff's rules [Bir35] provide this axiomatization but they contain two axiom schemas which need to be instantiated over the signature of the proof task. Unfortunately, in this is also necessary for any skolem functions introduced by clausification. Generating the equality axioms is thus usually integrated into the prover-internal preprocessing phase and need not be supported by NORA/HAMMR.

Operator Axioms

Building-in operator properties as associativity or commutativity by means of theory unification [Plo72] was one of the first attempts to reduce the number of axioms and, consequently, replace search by more goal-directed computation. However, none of the provers currently integrated into NORA/HAMMR implements theory unification. The extra operator axioms *associativity* and *commutativity* are nevertheless still useful because the rewrite machine used for task rejection (cf. chapters 5 and 6) employs a version of rewriting modulo AC.

[3] In fact, the initial rule library contained a number of inconsistencies which showed up only during the experimental evaluation. Some of these were eliminated by generating the sort axioms schematically (see below).

[4] Alternatively, Brand's modification method [Bra75] can be used but this requires extensive preprocessing of the proof task.

Sort Axioms

If a predicate encoding is used to relativize the sorted proof tasks, additional *sort axioms* are required or useful to represent all signature information adequately. These sort axioms may be divided into three categories.

1. *subsort axioms*: for each subsort relation $S_1 \prec S_2$ the subsort axiom $\forall x \cdot s_1(x) \Rightarrow s_2(x)$ is required [Obe62, Wei96]; for each pair of incomparable sorts S_1, S_2 a disjointness axiom $\forall x \cdot s_1(x) \Rightarrow \neg s_2(x)$ is required.

2. *signature axioms*: for each total operator $f : S_1 \times \ldots S_n \rightarrow S$ declared in the signature an axiom $\forall x_1, \ldots, x_n \cdot s_1(x) \wedge s_n(x) \Rightarrow s(f(x_1, \ldots, x_n))$ is required; for each partial operator an axiom $\forall x_1, \ldots, x_n \cdot pre_f(x_1, \ldots, x_n) \wedge s_1(x) \wedge s_n(x) \Rightarrow s(f(x_1, \ldots, x_n))$ is required [Wei96].

3. *sort inhabitation axioms*: for each sort S to which is *a priory* known to be inhabited (i.e., which has at least one element), a sort inhabitation axiom $\exists x \cdot s(x)$ can be generated.

The subsort axioms simply reflect the subset-relation between the two sort domains. The signature axioms allow the ATP to infer the sorts of composite terms. Both axiom categories are unnecessary for term encodings because the encodings build the respective properties directly into the representation. Extra sort inhabitation axioms are not strictly necessary. In fact, they are subsumed by the signature axioms if the signature contains at least one constant of the sort. However, they are useful for efficiency reasons, for example, if the constant is "buried" into a subsort, or if the sort only has a dynamic definition (e.g., prime numbers). Moreover, term encodings usually assume sort inhabitation implicitly (by virtue of skolem constants) so that extra sort inhabitation axioms help to keep the semantics of the different encodings consistent.

The sort axioms can easily be generated automatically for static sorts and total functions. The signature axioms for partial functions can also be generated automatically as described above. An optimized axiom version can be generated in certain cases if the precondition can be separated into conjunctively connected subexpressions, each constraining only a single component parameter. It is then possible to generate new (generally dynamic) subsorts with the negated separated subexpressions as membership axioms and to remove the precondition by tweaking the signature appropriately. For example, the *head*-function of the type $list \rightarrow item$ can be represented by the axiom

$$\forall l \cdot \mathrm{dom}_{head}(l) \Rightarrow item(head(l))$$

and the additional two sort definitions

$$\forall l \cdot \mathrm{dom}_{head}(l) \Rightarrow list(l)$$
$$\forall l \cdot \mathrm{dom}_{head}(l) \Leftrightarrow \neg l = [\,]$$

This is essentially one of the techniques mentioned in [Jon95] to handle partial functions in different logics; that paper also discusses the limits of this technique. NORA/HAMMR currently generates subsort axioms and the simple version of signature axioms.

Datatype Axioms

Generated and freely generated datatypes come with much internal structure which results in a potentially large number of axioms. (Cf. Definitions 5.2.2 and 5.2.5.) Fortunately, their internal structure is very regular such that the axioms can easily be generated automatically. The constructors of a generated datatype can be considered to span subsorts of the datatype such that the sort handling described above can be reused. For freely generated datatypes, these subsorts are mutually disjoint. This leaves generating the *injectivity* and *acyclicity axioms* as the only steps particular to freely generated datatypes. Both are instantiations of simple axiom schemes; all information required for the instantiation (i.e., generator functions and signatures) can be extracted easily from the lemma library.

9.4 Selection Mechanisms

Based on the hierarchically ordered rule library and the distinction between axioms and lemmas, several selection mechanisms can be implemented. They all follow the same general schema:

1. Determine the signature $\Sigma_{\mathcal{T}[q,c]}$ of the proof task, i.e., all occurring extralogical symbols.

2. Determine the defining theory T_f for each symbol f.

3. Select all axioms from all defining theories T_f.

4. Select all axioms from the theories recursively referred to by the T_f.

5. Optionally, select lemmas which do not introduce additional symbols.

This general schema selects axioms (but not lemmas) only if they are defined in non-redundant theories where a theory is considered to be redundant if it introduces only symbols not occurring in the problem and is not referred (directly or indirectly) by other non-redundant theories.

This general schema admits a variety of variants which I describe briefly and justify informally. These variants differ in the axioms and lemmas which are selected in steps 3–5. The choice of a specific variant is currently left to the NORA/HAMMR-user or to the reuse administrator.

- *pure calculus*: This is in fact a non-selection mechanism; it is the diametrical opposite of the naive non-selection mechanism *full library* because it selects no rule at all. Its justification is that it forces the prover to work with its calculus only which is usually very efficient and might be sufficient in many cases (e.g., if the query is a pure propositional or equational variant of the candidate component).

- *pure axioms*: This variant selects only the axioms of the symbols which actually occur in the proof task but does no completion along the theory hierarchy (i.e., omits steps 4 and 5 of the general schema). It is thus deliberately incomplete but its justification is again that it forces the prover to work with the actual proof problem and not with the background theory.

- *complete axioms*: This variant is the general schema without the optional lemma addition in step 5. It selects (by definition) the smallest set of rules which make a proof possible at all[5] but since this selection does not necessarily also guarantee the shortest proof, this property (and thus also this selection) is of limited interest in practice.

- *selector add-on*: The constructor and selector functions of a generated datatype are closely coupled by the semantics of that datatype. Hence, the selector axioms can be added even if the selector functions do not appear in the proof task.[6]

- *pure lemmas*: This variant builds on the *complete axioms* mechanism but restricts the set of additional lemmas selected in step 5 of the general schema to such lemmas defined in the theories determined in step 2. Its justification is that it keeps a complete axiomatization but "encourages" the prover to work with the actual problem and not with the background theory by providing more inference possibilities with the problem.

- *local lemmas*: This is a slight variant of the preceding *pure lemmas* mechanism; it uses all theories recursively referred to by the T_f (instead of only the T_f itself) to select the lemmas in step 5 of the general schema. Its justification is that it selects the smallest set of "necessary" rules as defined by the reuse administrator but at the expense of a reduced goal orientation.

- *full lemmas*: The final variant simply implements the full lemma addition step.

Obviously, the variants are not inherently independent of each other as their relative merits critically depend on an appropriate theory structure and the *axiom/lemma* distinction to be provided by the reuse administrator. For example,

[5]Provided that the library is consistent and theory persistent [RS98].

[6]The inverse situation that only the destructors appear and not the constructors is impossible because the constructors are automatically selected together with the datatype.

selector add-on can be simulated by *complete axioms* if the constructors and selectors of any datatype are always axiomatized in the same theory. However, this additional burden for the reuse administrator is minimal and justified because it also entails an additional degree of human control.

It is not completely clear from the available literature how these variants are related to the selection mechanism of W. Reif and G. Schellhorn [RS98]. However, the published examples suggest that their technique amounts to the *full lemmas* strategy.

9.5 Experimental Results

The obvious evaluation criterion for the selection mechanisms seems to be their reduction rate but I do not think that this is a sensible criterion—the proof of a useful selection mechanism is still in the proof of the tasks. Under this criterion, however, the selection mechanisms are much harder to evaluate because their effects may depend on or be overshadowed by other preprocessing steps, particularly by the proof task simplification which may alter the original signature of the task. To account for this dependency, I evaluated the selection mechanisms for the unsimplified and the fully simplified task variants, respectively. I only chose the two basic strategies *complete axioms* and *full lemmas* to keep the experimental effort within reasonable bounds; moreover, the relatively small size of the lemma library (28 axioms, 60 lemmas, on average 4.2 rules per symbol) and its relatively flat internal structure (maximal theory nesting level: 4) do not warrant the more elaborate strategies as they would result in different selections only for a very few tasks. In order to evaluate not only their relative merits over each other but also their absolute utility, I complemented these two selection strategies by the two "non-selection" strategies *pure calculus* and *full theory*; the *pure calculus* strategy, however, was applied only to the fully simplified task version in order to achieve useful recall levels.

Notwithstanding the claim that the ultimate utility measure for a selection mechanism is the number of proofs facilitated, a few numbers are useful results on their own. The two strategies reduce the average number of rules per proof task to 11.3 (*axioms*) and 24.6 (*lemmas*) with a maximum of 18 (*axioms*) and 47 (*lemmas*) rules per task. This reflects average reduction rates of approximately 87% and 78%, respectively, and is in accordance with the results by W. Reif and G. Schellhorn [RS98]. They report reduction quotas between 80% and 99%, based on significantly larger libraries containing between 400 and 500 rules.

Similar to the case of the simplification experiments described in the preceding chapter, the different provers respond quite different to the selection mechanisms. I thus again discuss the results of each of the individual provers separately and in some more detail before I draw some general conclusions.

OTTER

Figures 9.1 and 9.2 show the results for the different selection mechanisms for OTTER (`auto2`) for the original and simplified proof tasks, respectively.

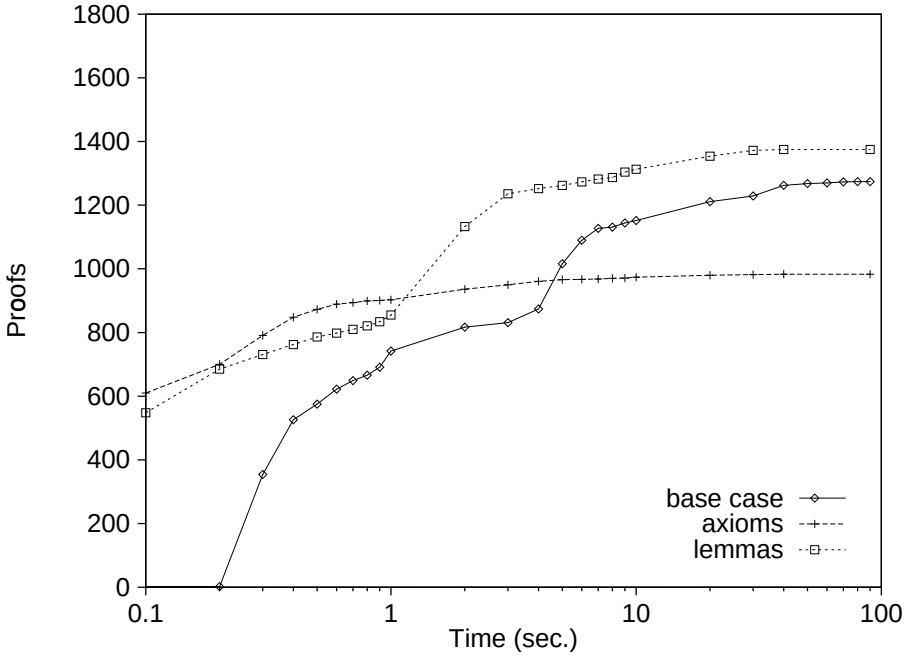


Figure 9.1: Lemma Selection: Proofs over time—OTTER (`auto2`), unsimplified tasks

Both selection mechanisms substantially increase the number of fast proofs for the original, unsimplified proof tasks because they reduce the amount of preprocessing OTTER has to spend on the input (i.e., clausification and back demodulation within the set of support). For short timeouts (less than 0.2 seconds), only selection enables proofs at all (*axioms*: 700 proofs, *lemmas*: 685 proofs) because preprocessing of the full rule base requires even more than the available time. With increasing timeouts, however, the "underselection effect" of the *axioms*-heuristic becomes apparent, i.e., the smaller branching factors are offset by the larger number of required proof steps. The *lemmas*-heuristic becomes superior for timeouts beyond a second and even the full rule library yields better results after 5 seconds. In the end, for $T_{max} = 90$ secs., the *lemmas*-heuristic yields an additional 101 proofs, increasing the recall average by more than 5%-points up to almost 75% while the *axioms*-heuristic looses 291 proofs or more than 20% of the original recall rate.

A comparison between figures 9.1 and 9.2 shows that similar effects can be observed for the simplified tasks but on a generally higher recall level. Again, and obviously for the same reason, both selection strategies substantially increase

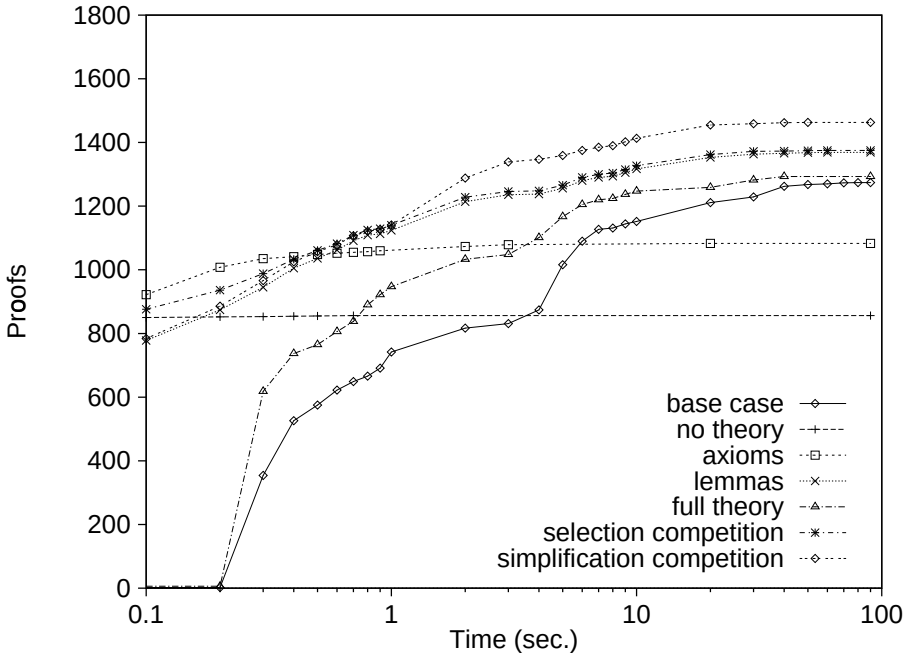


Figure 9.2: Lemma Selection: Proofs over time—OTTER (`auto2`), simplified tasks

the number of fast proofs. Even without any axioms, OTTER solves a significant number of proof tasks on top of those already solved by the simplifier without surjective unrolling of quantifiers: for $T_{max} = 0.2$ secs. 58 additional proofs are found on top of the 794 trivial tasks. However, task simplification amplifies the advantage of having a minimal axiomatization for short timeouts (*axioms*: 1008 proofs, *lemmas*: 874 proofs, again for $T_{max} = 0.2$ secs.) as well as the long-term advantage of additional lemmas. The *lemmas*-heuristic becomes superior for $T_{max} = 0.6$ secs. and thus even faster than in the base case and even the full library yields better results slightly faster. In the end, for $T_{max} = 90$ secs., however, almost the same picture emerges as for the unsimplified tasks. The *lemmas*-heuristic yields an additional 76 proofs or a 4.1%-points increase of the recall average over the entire library while the *axioms*-heuristic looses 210 proofs or still more than 15% of the original recall rate.

A cross-inspection of both figures reveals that the effects of the two selection mechanisms are not completely independent of the proof task simplification. The *axioms*-heuristic yields a recall improvement of approximately 10%-points and a speed-up of $s = 1.14$ for the simplified over the unsimplified task variant (cf. also Tables 9.2 and 9.3) while the *lemmas*-heuristic produces essentially the same results ($s = 1.00$) for both variants. Again, this seems to be a consequence of the high degree of incompleteness of OTTER's `auto2`-mode whose effects may overshadow the effects of lemma selection. This is confirmed by a simulated

competition between the *lemmas*-heuristic applied to both simplification variants. It solves a total 1463 proofs with an average response time per task of $\overline{T}_{valid} = 19.31$ secs. ($T_{max} = 90$ secs.); this represents a larger improvement of 6.4%-points in the recall level and a larger speed-up factor of $s = 1.24$.

In contrast to the case of the simplifications, a competition between the different selection mechanisms does not lead to substantial improvements. Figure 9.2 also shows a competition between the *pure calculus*- and *lemmas*-heuristic, respectively. However, except for short timeouts ($T_{max} \leq 0.2$ secs.) it is dominated by the *lemmas*-heuristic and within 90 seconds the *pure-calculus*-heuristic solves only 6 tasks exclusively and is significantly (i.e., more than 0.5 seconds) faster only for an additional 17 tasks.

GANDALF

As in the simplification experiments, GANDALF's behavior is quite different from OTTER's, despite their similar calculi.

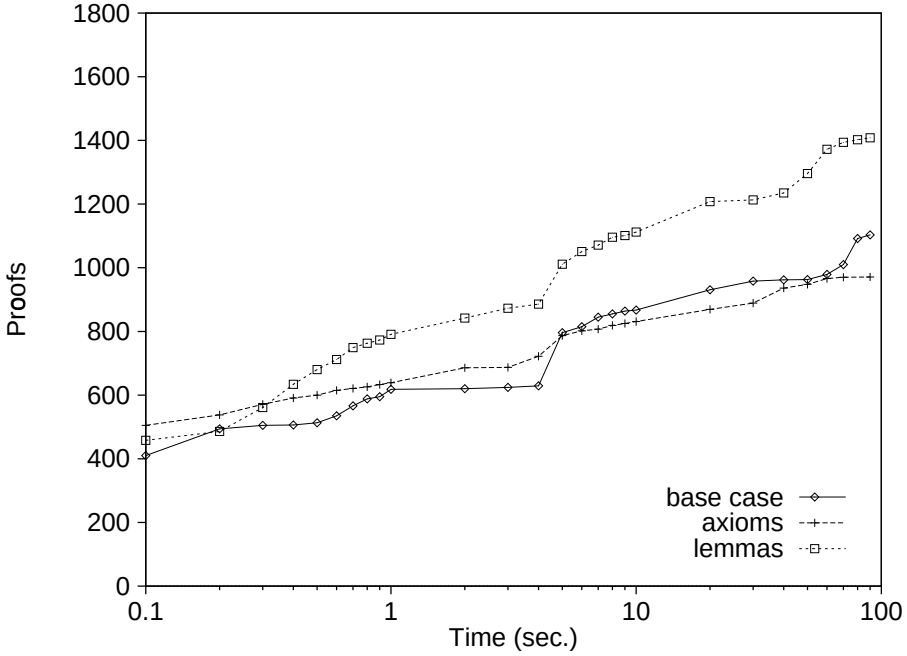


Figure 9.3: Lemma Selection: Proofs over time—GANDALF, unsimplified tasks

Since GANDALF does not incur the large preprocessing costs of OTTER (and SPASS), the difference between the selection mechanisms is much smaller for short timeouts. For the unsimplified tasks and $T_{max} = 0.2$ secs., both variants improve the performance of the base case (500 proofs) only slightly (*axioms*: 564 proofs, *lemmas*: 547 proofs). For the longer timeout of $T_{max} = 90$ secs. two more striking differences become visible at all. First, GANDALF does not suffer from

the underselection effect of the *axioms*-heuristic as much as OTTER. In fact, the plots for the base case and the *axioms*-heuristic run within narrow margins for approximately the first seventy seconds before a significant underselection effect becomes visible; in the end, it results in a loss of 133 proofs or a 7.4%-points decline of the recall average. This peculiar behavior (which is repeated for the simplified task version as well, cf. Figure 9.4) is most likely a consequence of GANDALF's time-slicing mechanism and, in particular, its "end-game mode" which favors (for this application profile) back subsumption into the *usable*-list (i.e., axioms and lemmas) [Tam97b] and thus rarely succeeds for the smaller *usable*-lists generated by the *axioms*-heuristic.

Second, GANDALF profits significantly more from the *lemmas*-heuristic than OTTER. This shows in two different aspects. In the short term, the break-even time of the *lemmas*-heuristic over the *axioms*-heuristic is much smaller than in OTTER's case and is only approximately 0.3 seconds. In the long term, the number of additional proofs found after $T_{max} = 90$ secs. is much higher, even allowing GANDALF to surpass OTTER in the total number of proofs: for GANDALF, the *lemmas*-heuristic yields an additional 304 proofs or 16.5%-points increase of the recall average.

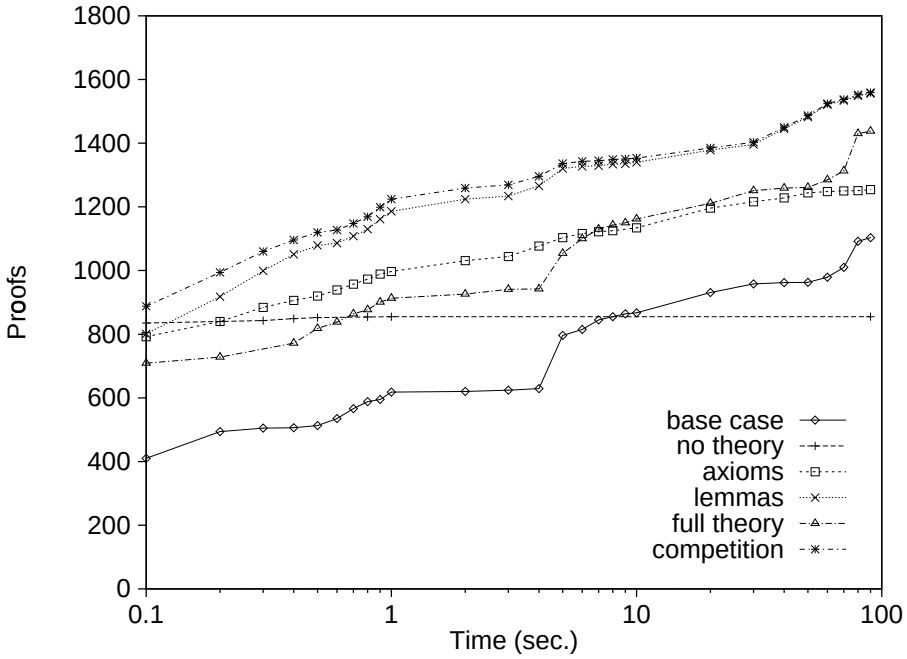


Figure 9.4: Lemma Selection: Proofs over time—GANDALF, simplified tasks

The simplified task variant (cf. Figure 9.4) essentially confirms the above observations but again amplifies the effects. For $T_{max} = 0.2$ secs., both variants improve the results achieved with the full theory (728 proofs) significantly more than in the case of the unsimplified case (*axioms*: 855 proofs, *lemmas*: 963

proofs); without any axioms, GANDALF still solves 46 non-trivial tasks, i.e., slightly fewer than OTTER. For the larger timeout of $T_{max} = 90$ secs., the *axioms*-heuristic follows the same pattern but the underselection effect becomes stronger: it results in a loss of 184 proofs or a 13.8%-points decline of the recall average. For the *lemmas*-heuristic, however, a saturation effect becomes visible which caps the achieved improvements at 118 additional proofs or a 2.8%-points increase of the recall average.

Similar to OTTER's case a competition between the different selection mechanisms does not result in any substantial further improvement. Due to the faster start-up of GANDALF, the competition between the *pure calculus* and *lemmas*-heuristic, respectively, is even more dominated by the *lemmas*-heuristic than in OTTER's case. Within 90 seconds, the *pure calculus* heuristic solves only 3 tasks exclusively; however, it is significantly faster for 40 more tasks or 2.5% of the tasks solved in total.

In contrast to OTTER's case the selection mechanisms work largely independent of the preceding simplification step. Both mechanisms yield the same speed-up of $s = 1.48$ for the simplified over the original task variant which is relatively close to the speed-up achieved by simplification alone ($s = 1.52$). The significantly higher recall improvement of the *axioms*-heuristic (+15.5%-points, *lemmas*: +8.6%-points) again indicates the beginning saturation effects of the combined preprocessing efforts.

SPASS

As in OTTER's case, both selection mechanisms substantially increase the number of fast proofs SPASS is able to find because they almost eliminate the preprocessing overhead associated with the large axiom sets.

For $T_{max} = 0.2$ secs., proofs can again only be found using the smaller axiom sets; again, the applied selection strategy is largely irrelevant, both for the original (*axioms*: 611 proofs, *lemmas*: 599 proofs) and the simplified variant (*axioms*: 826 proofs, *lemmas*: 792 proofs). As the timeout is increased, the underselection effect of the *axioms*-heuristic begins to show as usual and in the unsimplified case the *lemmas*-heuristic becomes superior for $T_{max} = 0.4$ secs. (cf. Figure 9.5). In contrast to all other provers, however, the *axioms*-heuristic still gives slightly better results than the base case; for $T_{max} = 90$ secs., it additionally yields 35 proofs or an 1.9%-points recall improvement compared to the base case. On the other hand, of all provers SPASS also receives the highest benefits from the *lemmas*-heuristic. Compared to the base case, the additional 392 proofs represent a 21.3%-points improvement of the recall average. This is significantly more than the 16.5%-points improvement achieved by the second-ranking prover GANDALF and allows SPASS to pass both OTTER and GANDALF.

Both observations together justify the conclusion that SPASS cannot handle large, redundant rule sets as well as other provers. This also corroborated

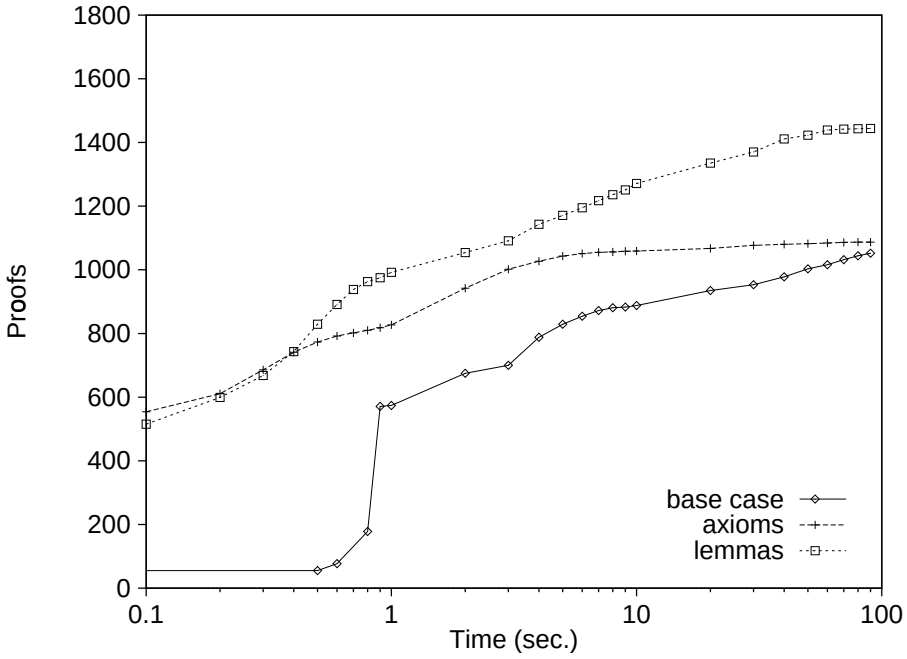


Figure 9.5: Lemma Selection: Proofs over time—SPASS, unsimplified tasks

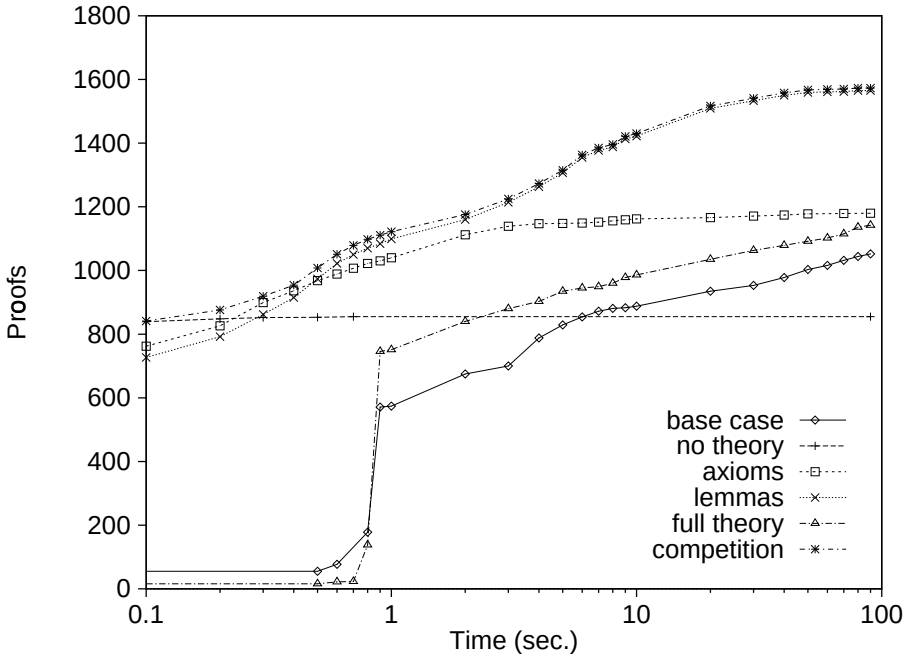


Figure 9.6: Lemma Selection: Proofs over time—SPASS, simplified tasks

by the experiments reported by W. Reif and G. Schellhorn [RS98] and further

confirmed by the quite similar results over the simplified tasks (cf. Figure 9.6).[7] Again, the minimal axiomatization of the *axioms*-heuristic allows SPASS to solve a small number of tasks more (+38 proofs) more than with the full library. The achieved recall improvement of 2.8%-points is in the same order or magnitude as for the original tasks. Similarly, the *lemmas*-heuristic yields 423 additional proofs (+23.0%-points), which is even slightly more than for the unsimplified tasks; again, this improvement makes SPASS the top-ranking prover.

In contrast to OTTER and GANDALF, proof task simplification here also amplifies the effects of lemma selection. This can be seen from the speed-ups the different mechanisms achieve if the simplified tasks are compared the original version. The *axioms*-heuristic yields a speed-up of $s = 1.15$ and a 6.4%-points improvement in the average recall level which is close to the effects of simplification alone ($s = 1.12$ / +4.9%-points) whereas the *lemmas*-heuristic results in a significantly higher speed-up ($s = 1.43$) and overall recall gain (+10.3%).

As already shown in OTTER's and GANDALF's case, a competition between the different selection mechanisms yield no noteworthy further improvements: while 8 tasks are solved exclusively under the *pure calculus* variant, only 26 proofs are found significantly faster than using the *lemmas*-heuristic.

The reasons for SPASS's different and exceptionally strong reaction to the lemma selection mechanisms are not completely clear; however, it is very likely to be related to the representation of sort information:

- SPASS uses very elaborate term indexing methods [Gra96] to organize the clause sets. The term encoding of the sorts leads to deeper term structures which then allow a finer indexing. This can in turn significantly speed up the basic retrieval operations (e.g., finding complementary unifiable literals) on which SPASS' performance critically depends.

- The term encoding does not require the signature and inhabitation axioms which induce larger search spaces.

- The sort predicates *list* and *item* may interfere with the automatically generated term orderings in such a way that inferences with relevant lemmas are postponed.

In order to explore this possible correlation further, I repeated the lemma selection experiments but now with the same term encoding as used for the other ATPs. Figure 9.7 shows the results for the simplified task variant in comparison with the standard predicative sort representation. It uncovers some interesting aspects; the most interesting aspect is undoubtedly the significantly higher number of proofs which is now found if the full theory is used (+295 proofs). The

[7] However, keep in mind that these experimental results (as well as those in [RS98] are based on a predicate encoding of the sorts. Cf. Figure 9.7 for results based on a term encoding.

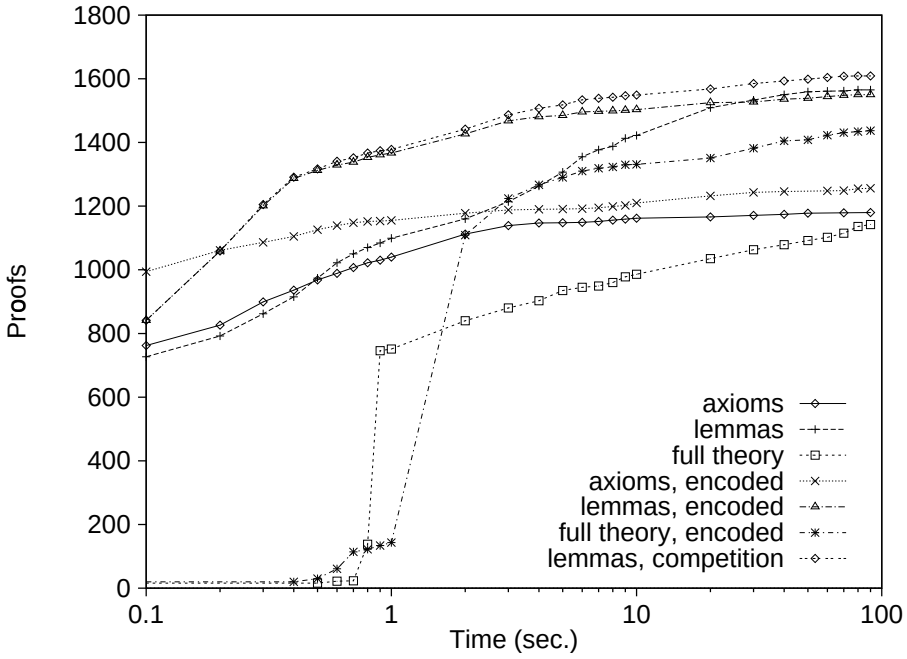


Figure 9.7: Lemma Selection: Proofs over time—SPASS, simplified tasks, different sort representations

total of 1437 proofs represents a recall level of more than 78% which is almost exactly the same level GANDALF achieves with exactly the same proof problems.[8] Moreover, the term encoding also leads to a substantial speed-up ($s = 1.66$), even if the pre-saturation of the problems takes slightly longer. The actual selection heuristics, however, benefit less—if at all—from the term encoding: the *axioms*-heuristic still yields a recall improvement (+76 proofs / +4.1%-points) while the *lemmas*-heuristic becomes slightly less efficient (-15 proofs / -0.8%-points). The achieved speed-ups drop appropriately (*axioms*: $s = 1.12$, *lemmas*: $s = 1.06$).

In total, the term encoding makes SPASS behave much more similar to the other provers. In particular, it now also exhibits the typical underselection effect when the *axioms*-heuristic is used. However, to determine the ultimate cause for SPASS' different behavior in the lemma selection experiments, further experiments and more detailed proof analyses are required which go beyond the scope of this thesis.

Independent of their ultimate cause, however, the different search spaces introduced by the two sort representations can be exploited by a competition (cf. Figure 9.7). This competition solves 1609 tasks in total which represents an average recall of $\underline{r} = 87.54\%$.

[8] Remember that GANDALF is run on the CNF produced by FLOTTER, the SPASS-clausifier.

SETHEO

One of the alleged advantages of top-down provers as SETHEO is their goal orientation, i.e., the fact that they start the proof from the conjecture and not from an arbitrary clause. This is then in turn claimed to allow them to handle larger axiom sets better.

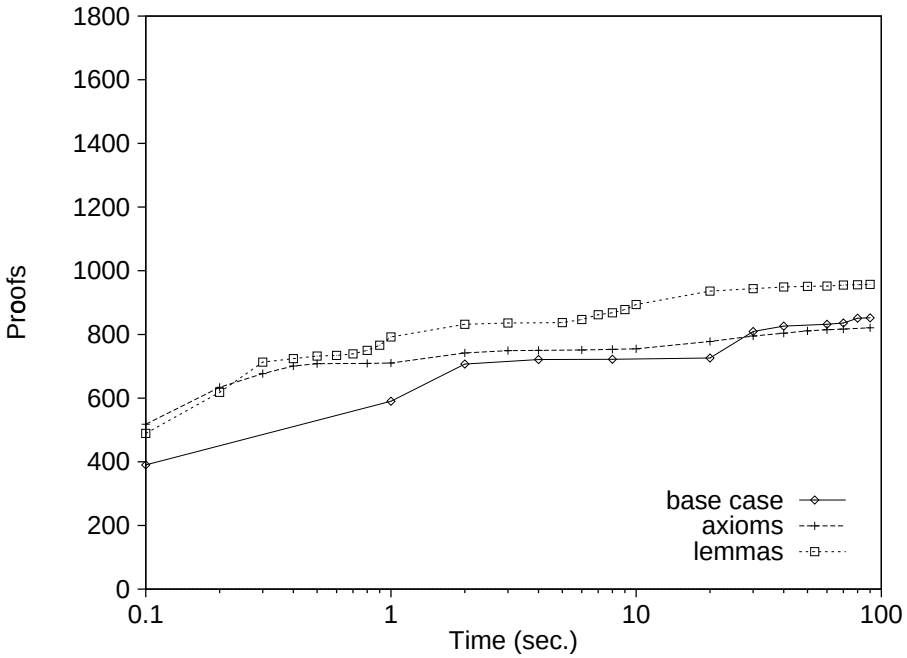


Figure 9.8: Lemma Selection: Proofs over time—SETHEO, unsimplified tasks

The retrieval experiments support this claim in an indirect way, at least to some extent, as SETHEO profits significantly less from the *lemmas*-heuristic than GANDALF and SPASS. For the unsimplified task variant and $T_{max} = 90$ secs., only 105 additional proofs (or +12.3%-points) are found, compared to the 304 and 392 additional proofs for GANDALF and SPASS, respectively, which represent higher gains (+16.5%-points / +21.3%-points) which in turn are even achieved from significantly higher initial recall levels. Conversely, the underselection effect of the *axioms*-heuristic is relatively small and amounts to a loss of only 31 proofs (or -3.6%-points) for $T_{max} = 90$ secs.; moreover, it becomes apparent only for $T_{max} \geq 30.0$ secs.. These numbers justify the conclusion that SETHEO is relatively less sensitive to large, redundant rule sets than the other provers used in NORA/HAMMR.

Similar to the situation of the other provers, both mechanisms significantly improve SETHEO's performance for very short timeouts since they cut down the time spent on preprocessing (i.e., compilation into abstract machine code): for $T_{max} = 0.2$ secs., both heuristics achieve almost the same recall levels ($\underline{r} \approx 35\%$)

and both find a substantial number of non-trivial proofs (*axioms* 254 proofs / *lemmas* 250 proofs) while SETHEO needs more than a second to find the first proof using the full axiomatization.

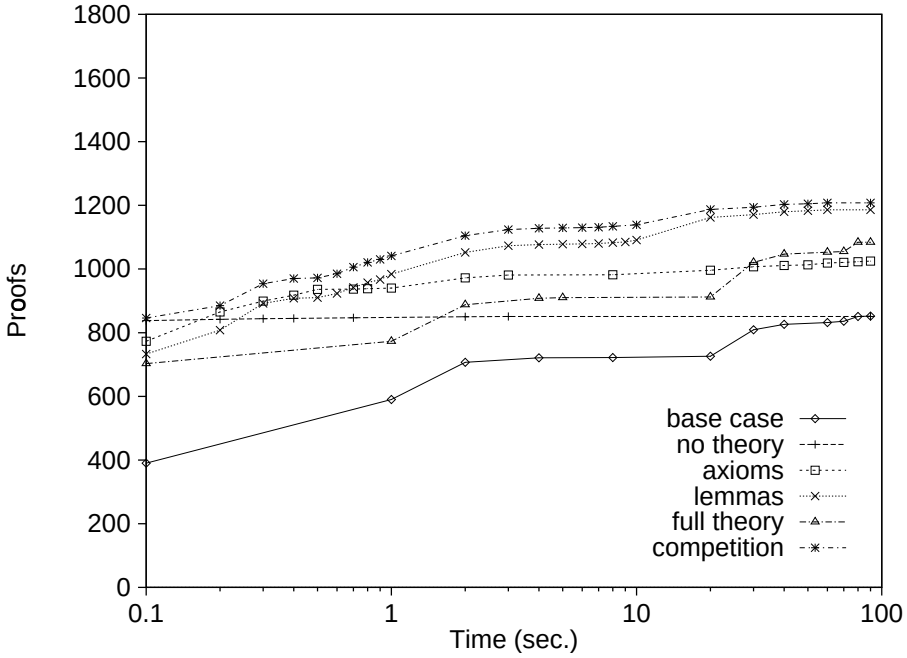


Figure 9.9: Lemma Selection: Proofs over time—SETHEO, simplified tasks

For the simplified task variant, a similar picture emerges (cf. Figure 9.9). Again, the underselection effect of the *axioms*-heuristic becomes apparent relatively late ($T_{max} \geq 30.0$ secs., as in the unsimplified case) and remains relatively small (-59 proofs / -5.4%-points) for $T_{max} = 90$ secs. And again, the benefits of the *lemmas*-heuristic remain relatively small. It yields only 84 proofs (or +7.7%-points) more than the full lemma library, i.e., even less than in the unsimplified case.

For SETHEO, proof task simplification also amplifies the effect of lemma selection but less pronounced than in SPASS' case. The *axioms*-heuristic yields a larger speed-up of $s = 1.25$ for the simplified over the original proof tasks and comes thus much closer to the result of the *lemmas*-heuristic which here only yields $s = 1.34$. However, SETHEO is the only prover for which the *lemmas*-heuristic does not dominate all other mechanisms. A competition between the *axioms*-heuristic and the *pure calculus* mechanism yields over the simplified tasks a total of 1208 proofs ($\underline{\mathrm{r}} = 65.72\%$), or a modest 3.4%-points (+40 proofs) recall improvement over the *lemmas*-heuristic.

General Results

The most general conclusion from the more detailed, prover-specific results discussed above is that *lemma selection pays*, i.e., the additional effort put into the extra selection-phase is more than offset by a higher number of proofs and shorter overall response times; moreover, the *runtime* overhead of the selection phase is virtually zero. Tables 9.1 and 9.2 summarize these results for the original and simplified proof tasks, respectively.

	OTTER		GANDALF		SPASS		SETHEO	
mode	*axioms*	*lemmas*	*axioms*	*lemmas*	*axioms*	*lemmas*	*axioms*	*lemmas*
T_{max} (sec.)	90.00	90.00	90.00	90.00	90.00	90.00	90.00	90.00
$\overline{T}_{valid}$ (sec.)	42.12	23.95	45.39	28.30	37.61	22.75	51.01	44.23
σ_T	44.68	38.49	43.15	37.66	43.76	36.35	43.92	44.24
Σ_T	77420	44013	83427	52006	69134	41810	93757	81288
$\overline{T}_{proof}$ (sec.)	0.79	2.38	9.29	13.07	2.21	6.02	5.17	3.52
σ_T	2.71	4.62	15.31	20.34	6.94	11.92	13.52	9.27
# proofs	983	1375	971	1408	1087	1444	821	957
$\underline{r}$ (%)	53.48	74.81	52.83	76.61	59.14	78.56	44.67	52.07
$\overline{r}$ (%)	60.61	67.80	53.73	64.90	60.73	70.61	38.68	36.76
σ_r	34.46	34.24	35.45	35.88	34.69	31.16	34.22	35.11
$\overline{p}$ (%)	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00

Table 9.1: Lemma Selection: General Results, Unsimplified Tasks

For very short timeouts, *every* selection mechanism pays, no matter how it is semantically justified. This is a consequence of the essentially breadth-first search pattern which all provers ultimately implement to retain completeness. Any selection mechanism cuts down the branching factor of the search space significantly which results in a better depth-coverage and thus more proofs for a given timeout.

The extreme incarnation of this approach is the *pure calculus* (non-) selection mechanism, i.e., a proof attempt without any axioms. It is surprisingly effective, given its simplicity and radicality. It allows the ATPs to solve a large number of tasks (for most provers approximately 100 tasks, for SPASS—due to the expensive preprocessing—even more than 800) significantly faster than with the full background theory; the cut-off timeout for this strategy is between 0.7 secs. and 1.1 secs., depending on the prover. Moreover, for a substantial number of tasks it still enables faster proofs than any other mechanism and a small number of proofs can be found at all only using this strategy, even for $T_{max} = 90$ secs.

The *pure calculus* mechanism is easy to implement within any ATP and

requires—in contrast to the other selection strategies—no additional user intervention. It can be used as a "pre-prover" with very restricted resources, similar to the approach taken in the FLOTTER-clausifier [NRW98], where the SPASS-prover is invoked. Such a setup does probably not help with highly tuned benchmark collections as the TPTP which contain mostly hard problems but in "real life" applications as here it certainly pays. Hence, if no other, "fancier" selection mechanisms are implemented, this approach should given a chance.

With increasing timeouts, however, the restricted *pure calculus* mechanism as well as the theoretically sufficient *axioms*-heuristic suffer from a severe *underselection effect*, i.e., the smaller branching factors are offset by the larger number of proof steps, and the number of proofs found becomes smaller than in the case of the more permissive strategies. This leaves the *lemmas*-heuristic as the selection method of choice. In practice, it almost dominates all other methods; a competition between different methods yields thus only minor improvements for $T_{max} = 90$ secs. (cf. Table 9.2). Consequently, the distinction between proper axioms and derived lemmas seems to be unnecessary in practice, at least for relatively small lemma libraries as the one used here.

Another important result of the lemma selection experiments is that *lemma selection is not preempted by simplification and vice versa*. That is, even though the two preprocessing steps are not completely independent of each other, neither of them subsumes the other and their combination allows most provers to solve significantly more tasks.[9] The exact relation between simplification and lemma selection, however, seems to be very intricate and depends on the particular ATP. For GANDALF, simplification ($s = 1.53$, +18.2%-points) and lemma selection ($s = 1.53$, +16.6%-points) yield more or less the same results. Both preprocessing steps are almost orthogonal to each other and their combination yields thus a speed-up of $s = 2.25$. SPASS profits much more from the lemma selection ($s = 1.88$, +21.3%-points) than from simplification ($s = 1.12$, +4.9%-points). Their combination, however, yields additional synergies, i.e., the combined speed-up ($s = 2.69$) and recall improvement (+27.9%-points) are even larger than the product and sum, respectively, of the individual values. SETHEO, on the other hand, is more receptive to simplification ($s = 1.26$, +12.3%-points) than to lemma selection ($s = 1.12$, +5.7%-points); however, the combination of both steps again yields additional synergies, although to a smaller extent than in SPASS's case ($s = 1.61$, +18.2%-points). The conclusion is that proof task simplification and lemma selection are equally important for applications and must both be supported by ATPs if they are intended to be used in applications.

W. Reif and G. Schellhorn [RS98] report on a similar approach to the lemma selection problem. Their experimental evaluation is done in the context of three different case studies in program verification but is based on significantly less

[9] The only exception here is OTTER which is extremely sensitive to any reorganization of the proof task, due to the highly incomplete `auto2`-mode (cf. Section 8.2).

	OTTER				GANDALF			
mode	*axioms*	*lemmas*	comp.	pipe.	*axioms*	*lemmas*	comp.	pipe.
T_{max} (sec.)	90.00	90.00	90.00	5.00 / 90.00	90.00	90.00	90.00	5.00 / 90.00
$\overline{T}_{valid}$ (sec.)	37.04	23.95	23.57	-	30.67	19.15	18.79	-
σ_T	44.22	38.83	38.69	-	41.23	33.60	33.52	-
Σ_T	68083	44027	43327	-	56374	35189	34531	-
$\overline{T}_{proof}$ (sec.)	0.34	2.72	2.46	-	6.92	11.50	11.00	-
σ_T	1.35	5.81	5.29	-	13.22	20.30	20.22	-
$\overline{T}_{query}$ (sec.)	-	-	-	2873	-	-	-	2821
σ_T	-	-	-	1733	-	-	-	1666
# proofs	1083	1369	1375	1540	1254	1556	1559	1658
$\underline{r}$ (%)	58.92	74.48	74.81	83.79	68.23	84.66	84.82	90.21
$\overline{r}$ (%)	60.33	72.71	74.52	80.23	71.13	81.13	82.39	85.16
σ_r	33.20	29.91	27.55	26.39	30.42	26.31	23.44	24.69
$\overline{p}$ (%)	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00
	SPASS				SETHEO			
mode	*axioms*	*lemmas*	comp.	pipe.	*axioms*	*lemmas*	comp.	pipe.
T_{max} (sec.)	90.00	90.00	90.00	5.00 / 90.00	90.00	90.00	90.00	5.00 / 90.00
$\overline{T}_{valid}$ (sec.)	32.77	15.94	15.51	-	40.69	33.06	31.76	-
σ_T	42.90	31.72	31.37	-	44.33	42.48	42.29	-
Σ_T	60234	29293	28513	-	74785	60769	58373	-
$\overline{T}_{proof}$ (sec.)	2.11	5.47	5.35	-	5.01	4.33	3.31	-
σ_T	7.17	9.58	9.58	-	13.80	8.54	7.94	-
$\overline{T}_{query}$ (sec.)	-	-	-	2826	-	-	-	2941
σ_T	-	-	-	1687	-	-	-	1670
# proofs	1180	1565	1573	1618	1025	1168	1208	1452
$\underline{r}$ (%)	64.20	85.15	85.48	88.03	55.77	64.53	65.72	79.00
$\overline{r}$ (%)	67.65	79.44	83.51	81.68	48.94	51.07	62.49	63.16
σ_r	33.50	29.29	21.88	28.51	36.68	36.89	33.75	35.83
$\overline{p}$ (%)	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00

Table 9.2: Lemma Selection: General Results, Simplified Tasks

(139 in total) proof tasks. In general, both experiments yield similar results. [RS98] also report a generally increased number of proofs found, as well as faster proofs, independent of the particular ATP. The improvements described there are consistent with the improvements observed here. In particular, [RS98] also

note SPASS' strong sensitivity to large lemma sets (if the built-in predicative sort encoding is used; term encodings are not investigated) as well as the relative insensitivity of tableau-style provers, i.e., SETHEO. Together with the experiments described here, they cover a wide spectrum of application characteristics. Over this spectrum, the effectivity of signature-based heuristics is, hardly surprising, determined by the signature characteristics of the proof tasks and lemma library. They are most effective when the domain theory contains only few sorts (or, by extension, is even unsorted) and relatively few operators but many axioms and lemmas. If term encodings are used, their number is the critical variable: the more sorts the domain contains, the better the built-in indexing schemas of the ATPs cope with large lemma sets. Conversely, hierarchical specifications can be considered as an additional indexing schema which complements the pure sort-based indexing.

Finally, Table 9.3 shows again the results of the prover competition and its combination with the rewrite-based rejection filter. For each prover, only the best variant (i.e., simplified tasks and *lemma*-heuristic has been used.

	OTTER & SPASS			full competition				
T_{max} (sec.)	1.00	20.00	90.00	1.00	20.00	90.00	5.00 / 20.00	5.00 / 90.00
$\overline{T}_{valid}$ (sec.)	0.93	17.64	79.97	0.92	17.88	79.88	-	-
σ_T	0.25	5.95	27.99	0.26	6.07	28.17	-	-
Σ_T	784	7518	23399	674	6741	22113	-	-
$\overline{T}_{proof}$ (sec.)	0.15	1.36	2.39	0.13	0.95	2.14	-	-
σ_T	0.22	3.30	6.93	0.20	2.94	7.52	-	-
$\overline{T}_{query}$ (sec.)	110.1	2134	9517	109.0	2128	9506	736.1	2692
σ_T	20.6	429	1971	21.0	433	1977	435.3	1670
# proofs	1233	1569	1621	1365	1576	1631	1648	1673
$\underline{r}$ (%)	67.08	85.36	88.19	74.27	85.75	88.74	89.66	91.02
$\overline{r}$ (%)	66.62	80.66	83.76	71.43	81.19	84.34	84.71	86.23
σ_r	34.71	26.32	23.64	31.86	26.25	23.73	24.45	23.26
$\overline{p}$ (%)	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00

Table 9.3: Lemma Selection: Prover Competition

At $T_{max} = 90$ seconds, the speed-up factor of the full competition over the fastest individual prover (SPASS) on the valid proof tasks is $s = 1.32$; the efficiency of this competition is $e = 0.33$. The restricted competition between OTTER and SPASS yields $s = 1.25$ and $e = 0.63$ under these conditions. These speed-ups are comparable to those observed in the other competitions (cf. Tables 7.2 and 8.2)

The combination of prover combination and the rewrite-based rejection filters again improves the recall noticeable. An overall recall level of approximately 90% is already achieved with a rewriter timeout of $T_{max} = 5.0$ secs. and a prover timeout of $T_{max} = 20.0$ secs. for each individual prover. The average query response time $\overline{T}_{query}$ of approximately 12 mins. means that the expected further hardware improvements make deduction-based software component retrieval a technically feasible option for the near future.

Chapter 10

Summary and Conclusions

Deduction-based software component retrieval is a multi-faceted research topic. In this thesis I have obviously investigated only some of the facets. This final chapter summarizes its main results (Section 10.1) and relates them to other work on this topic (Section 10.2). The very final Section 10.3 briefly sketches some other facets which are left as future work.

10.1 Summary of Results

Deduction-based software component retrieval is an ambitious technology which has an enormous potential to reduce the cost incurred and the time required to develop highly reliable software. In this thesis, I have investigated several aspects of deduction-based software component retrieval and shown ways to make it practical. Its main theoretical contributions are the investigation and characterization of abstract match predicates and retrieval algorithms and the detailed investigation of different match predicates, their reuse effects, and their interrelations. Its main practical contributions are the design and implementation of an advanced retrieval system architecture and the substantial experiments with off-the-shelf ATPs to evaluate and demonstrate the technical feasibility of deduction-based software component retrieval.

In Chapter 2, I have presented a new, abstract view of component retrieval based only on the concept of sets of relevant, matching, and found components, respectively. I have shown how properties of abstract match predicates (e.g., transitivity) are reflected in these sets and vice versa and how this duality can be used to build library indexes. I have introduced the concepts of closure under iterated retrieval and query stability which characterize an important class of retrieval algorithms. For the algorithm evaluation in the context of NORA/HAMMR's architecture, I have introduced the concepts of precision leverage and relative defect ratio which measure the filtering qualities of an retrieval algorithm and I have shown how these concepts relate to each other.

In Chapter 3, I have demonstrated how different match predicates are built up from their constituents, i.e., quantifier prefix, type compatibility predicate, and body. I have analyzed how the choices of different type compatibility predicates and—especially—quantifier prefixes already affect the reusability of retrieved components, even with the same body; this is also the first time other quantifier prefixes than the usual fully universal prefix have been considered for deduction-based retrieval. I have identified relevance conditions and reuse effects for the three different retrieval modes, i.e., exact, proper, and approximate retrieval, and defined various match predicates for these modes. I have also formally shown under which conditions on the component specifications and queries match predicates become equivalent or follow from others.

In Chapter 4, I have identified the two most important user requirements to making deduction-based software component retrieval more practical, namely "Components, not proofs!" and "Results-while-u-wait!". From these, I have developed a new architecture for retrieval systems which is based on a pipeline of filters of increasing deductive strength. This pipeline architecture allows to filter out non-matches quickly and cheaply and thus prevents the applied theorem provers from "drowning". It also supports the any-time inspection of all intermediate results and it can easily be extended to a parallel architecture, thus harnessing the computational power of an entire local or wide-area network for retrieval purposes. This architecture is implemented in the fully automatic retrieval system NORA/HAMMR, which is the first realistic complete implementation of the deduction-based retrieval concept.

In Chapters 5 and 6, I have shown that relatively simple techniques are sufficient to identify a sufficiently large number of non-matches quickly and cheaply. In Chapter 5, I have described the application of term rewriting to simplify the proof tasks and to reduce them eventually to *true* or *false*, thus exposing trivial matches and non-matches, respectively. I have developed a general quantifier elimination schema and instantiated it for equality types (i.e., the diagonalization rule) and generated (or data) types (i.e., the surjective unrolling rule). This approach is surprisingly effective for the test library and reduces the fallout of the answer set to almost 25%. In Chapter 6, I have analyzed the application of specific counterexamples to identify non-matches. I have used both term rewriting and theorem proving to evaluate the proof tasks over prospective counterexamples. Term rewriting turned out to be clearly better, reducing the fallout of the answer set to less than 15%.

In Chapters 7 to 9, I have shown that fully automatic theorem provers for first-order logic have in principle reached such a level of maturity that their performance is no longer the major obstacle to building deduction-based retrieval systems. In fact, this thesis even represents—to the best of my knowledge—the first *serious* attempt to investigate and evaluate the suitability of different

provers for the proof tasks emerging in deduction-based retrieval.[1] However, the experiments revealed that the applied provers were not yet mature enough to work directly on the "raw" automatically generated proof tasks. In this case, recall levels do not exceed 70%, even with timeouts of $T_{max} = 90.0$ secs. per individual proof task (cf. Figure 10.1).

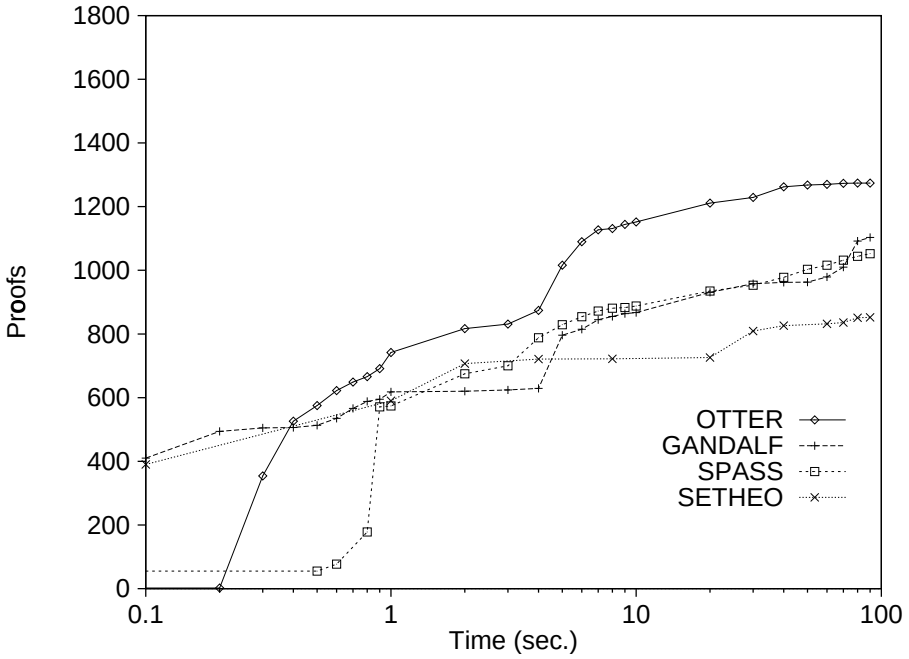


Figure 10.1: Proofs over time—base case

In Chapters 8 and 9, I have thus experimented with two independent proof task preprocessing techniques. In Chapter 8, I have reused the term rewriting techniques from Chapter 5 to simplify the conjectures; in Chapter 9, I have developed a simple signature-based heuristic to simplify the background theory by identifying a suitable subset of axioms and lemmas. Both techniques generally improve the prover performances, and thus lead to increased recall levels. Moreover, they are to a certain extent complementary and their combination yields additional improvements, resulting in overall recall levels of 65%–85% (cf. Figure 10.1).

A welcome side effect of the preprocessing techniques described in Chapters 8 and 9 is that the provers do not only solve more problems but also solve them faster. The average response times per valid proof task (cf. Table 10.1) drop

[1] A. Moorman Zaremski's investigations [Moo96] can hardly be called serious. They are based on a test set of only 33 valid tasks; moreover, she used the interactive LarchProver as theorem prover. J. Penix's investigations [Pen98] are based on a precursor of the library used here; however, since he follows a different approach to retrieval, the simpler tasks are overrepresented in his set-up.

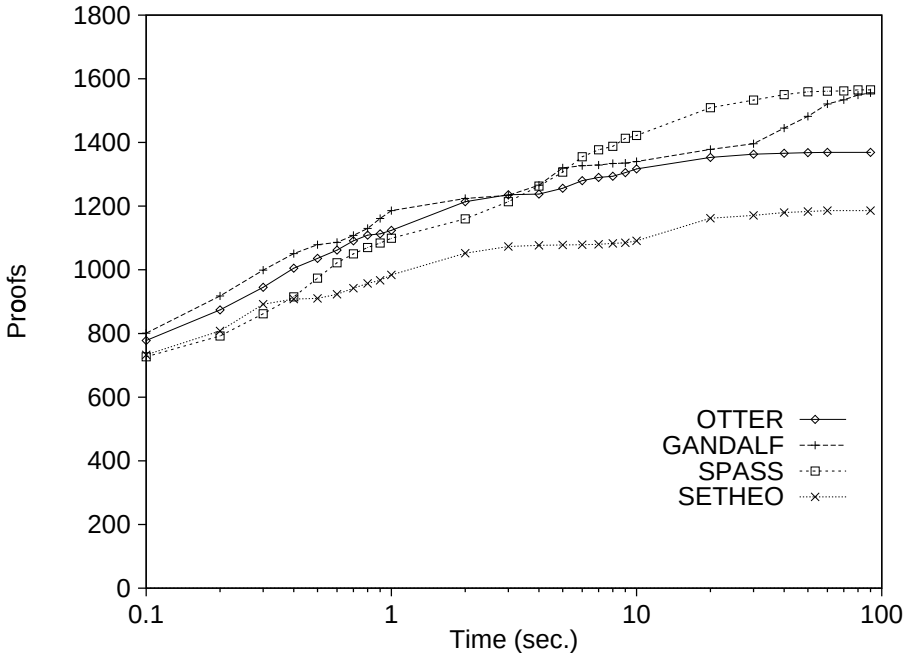


Figure 10.2: Proofs over time—best case

	Base Case		Best Case		Speed-up	
	task	proof	task	proof	task	proof
OTTER	30.46	3.95	23.95	1.33	1.27	2.97
GANDALF	43.14	11.99	33.60	6.30	1.28	1.90
SPASS	42.83	7.59	15.94	3.02	2.69	2.51
SETHEO	50.98	5.81	33.06	0.40	1.54	14.53

Table 10.1: Comparison of response times and proof times

approximately from 30–50 seconds in the base case to 15–30 seconds for the best variant. For most provers, the aggregated speed-up falls into the range from $s = 1.25$ to $s = 1.5$; the exception is SPASS which profits extraordinarily from the lemma selection step (cf. Section 9.5) and thus achieves a significantly higher speed-up factor of $s = 2.69$. Interestingly, the average proof times drop even faster, from approximately 4–12 seconds to approximately 0.4–6 seconds; this is reflected in the generally significantly higher speed-ups achieved. This means that the applied preprocessing steps (i.e., simplification and lemma selection) are working uniformly over all proof tasks and are still effective even for the simpler tasks which the provers can already solve in the original form.

Despite all improvements, the common brute-force, all-out-match approach to deduction-based component retrieval remains naive and unfeasable. In order to

achieve acceptable recall levels within acceptable response times, more elaborate approaches as for example NORA/HAMMR's pipeline architecture are required. However, there is no *one* optimal pipeline; rather, the pipeline structure depends on the targeted response times and the available computational resources. Table 10.2 summarizes the pipelining effect for five different pipelines.

	$\mathcal{P}_1$	$\mathcal{P}_2$	$\mathcal{P}_3$	$\mathcal{P}_4$	$\mathcal{P}_5$
$\overline{T}_{task}$ (sec.)	0.47	1.00	1.00	5.35	12.05
σ_T	0.45	1.56	1.60	10.25	31.53
$\overline{T}_{query}$ (sec.)	55.82	118.9	118.5	636.9	1434.1
σ_T	22.39	76.9	79.1	499.1	1666.7
T_{max}	114.12	399.9	424.4	2781.6	9880.2
# proofs	1355	1473	1581	1621	1675
$\underline{r}$ (%)	73.72	80.23	86.11	88.29	91.23
$\overline{r}$ (%)	67.36	75.95	80.91	83.65	86.40
σ_r	35.70	31.51	28.79	26.02	23.30
$\overline{p}$ (%)	100.00	100.00	100.00	100.00	100.00

Table 10.2: Comparison of different filter pipelines

$\mathcal{P}_1$ is a single-processor pipeline designed to guarantee query response times of less than two minutes (i.e., a second per component) even in the worst case. It thus comprises only two filters, a single rejection filter and a single confirmation filter. $\mathcal{P}_1$ uses the rewrite-based rejection filter $\mathcal{R}_{DOMAIN}$ with quantifier unrolling ($\mid t \mid_{max}= 10000$) and a time limit of $T_{max} = 0.2$ secs., followed by a SPASS-based confirmation filter with term-based sort encoding, lemma selection, and a time limit of $T_{max} = 0.8$ secs. By construction, SPASS works on the simplified task variant, because the intermediate results (i.e., before quantifier unrolling) of $\mathcal{R}_{DOMAIN}$ are fed into the confirmation filter. Compared to the base case (cf. Table 7.1), the pipeline yields dramatically improved results, with a more than 30%-points recall increase for the same timeout.

The second pipeline, $\mathcal{P}_2$, is also a single-processor pipeline designed for short query response times; however, here an *average* response time of less than two minutes is targeted. Since the recall of a pipeline is determined by the final confirmation filter, it is necessary to allocate for each task as much prover time as possible. Hence, an improvement over $\mathcal{P}_1$ can already by achieved by increasing SPASS' time limit such that the targeted average query response time is maintained. This allows a time limit of $T_{max} = 2.3$ secs. and yields a recall of $\underline{r} = 78.24\%$ or 1438 proofs. In this case, the maximal query response time amounts to 261.0 secs. However, a further improvement can be achieved by initially allocating more time for the rejection filters. Hence, $\mathcal{P}_2$ uses two re-

jection filters, $\mathcal{R}_{DOMAIN}$ followed by the counterexample-based filter $\mathcal{R}_1$. Both filters are used with the same setup as in $\mathcal{P}_1$ except for the greater time limits of $T_{max} = 0.5$ secs. each. This allows SPASS to be run with the significantly increased time limit of $T_{max} = 3.8$ secs., which yields the 1473 proofs ($\underline{r} = 80.23\%$) shown in Table 10.2.

The only slight disadvantage of this approach is that for the very few queries for which the rejection filters fail to reduce the fallout substantially, the additional time spent in the rejection filters and the larger prover time limit directly translate into larger response times. For the worst case, the query response time thus increases to 399.9 secs. This increase is, however, still slightly below the theoretical maximum due to the increased prover time which indicates that the rejection filters still recoup some of the initial time investment.

The third pipeline, $\mathcal{P}_3$, is a slight modification of $\mathcal{P}_2$. It is targeted at the same response times and uses the same filters with (almost) the same timeouts but swaps the order of the two rejection filters. Since $\mathcal{R}_{DOMAIN}$ is sound, this set-up allows to bypass SPASS for tasks which are already reduced to *true* by $\mathcal{R}_{DOMAIN}$ and thus to take advantage of the (partial) complementarity of rewriting and proving already demonstrated throughout the experiments. This results in a substantially higher number of proofs (1581) and thus a higher recall ($\underline{r} = 86.11\%$) than in the case of $\mathcal{P}_2$.

The fourth pipeline, $\mathcal{P}_4$, is again a single-processor pipeline. It uses the same filters in the same order as $\mathcal{P}_3$. However, it is designed to optimize the recall within acceptable average response times. Hence the time limits have been increased to $T_{max} = 5.0$ secs. for the rejection filters and $T_{max} = 20.0$ secs. for SPASS. However, in comparison with the results of $\mathcal{P}_3$, it is not clear whether $\mathcal{P}_4$ accomplishes this goal. In total, $\mathcal{P}_4$ retrieves 40 components more, increasing the overall recall by slightly more than 2%-points; $\mathcal{P}_4$ also works well more uniformly, as witnessed by the higher query-oriented recall average ($\overline{r} = 83.65\%$) and lower variance ($\sigma_r = 26.02$). However, these better results are obtained only at the expense of an almost five-fold increase in average response times and a maximal query response time of approximately 45 mins.

The final pipeline $\mathcal{P}_5$ is a multi-processor pipeline which employs up to four processors. Its goal is to maximize the overall recall, within roughly the same average response times as before but now harnessing much more computational power. $\mathcal{P}_5$'s general structure is the same as that of $\mathcal{P}_3$ and $\mathcal{P}_4$ but it uses competition in each step: instead of a single counterexample-based filter it uses all three, instead of only the CNF-version of $\mathcal{R}_{DOMAIN}$ it uses both the CNF-version and the DNF-version in parallel, and of course it uses all provers in a competition which also includes the term encoding and predicate encoding versions for SPASS. Moreover, the prover time limits are increased to $T_{max} = 90.0$ secs. Due to this massive computational effort, $\mathcal{P}_5$ reaches even higher recall levels than $\mathcal{P}_4$ but the saturation effect which already surfaced with $\mathcal{P}_4$ becomes even more apparent.

10.2 Related Work

Deduction-based retrieval has enjoyed quite some popularity as a research topic; consequently, there is an abundance of related work. This section is not intended as an exhaustive literature survey. Instead I focus on four different aspects which are most closely related to NORA/HAMMR.

Match Relations

The definition and investigation of different match relations, their interrelations, and—to a lesser extent—their effects on the reusability of the retrieved components has been a major activity from the beginning of deduction-based retrieval. Its attraction can no doubt to a large extent be explained by the fact that this can (and has) be done as a purely theoretical exercise and does not require any working retrieval system for experimentation.

The original work on specification matching by E. Rollins and J. Wing [RW90b, RW91] uses only strict plugin match (cf. Def. 3.3.4) as match predicate. A. Moorman Zaremski and J. Wing [MW95b, Moo96] extend this work. They define two generic forms of match predicates, *generic pre/post match*

$$(pre_q \,\mathcal{R}_1\, pre_c)\, \mathcal{R}_2\, (post_c \,\mathcal{R}_3\, post_q)$$

and *generic predicate match*

$$\mathcal{F}[pre_q, post_q] \Leftrightarrow \mathcal{F}[pre_c, post_c]$$

and then derive a variety of different match predicates by instantiating the logical operators $\mathcal{R}$ and the specification predicate $\mathcal{F}$. More details can be found in the discussions in Chapter 3. In [MW97b] they modify their definition of generic pre/post match to

$$(pre_q \,\mathcal{R}_1\, pre_c)\, \mathcal{R}_2\, (\widehat{\mathcal{C}} \,\mathcal{R}_3\, post_q)$$

to allow "guarded" versions of some matches by instantiating $\widehat{\mathcal{C}}$ to either $post_c$ or to $pre_c \wedge post_c$. However, their rigid syntactic framework precludes the definition of proper match (cf. Def. 3.3.1), the most natural and common match predicate. Moreover, since Moorman Zaremski and Wing do not consider explicit quantification, they restrict themselves (implicitly) to the case of fully universal prefixes.

J. Penix' dissertation thesis [Pen98] also discusses a variety of match predicates. Penix rightly rejects Moorman Zaremski and Wing's purely syntactic approach and instead "selects matches based on formalizations of intuitive notions of reusability, and their utility in component retrieval" [Pen98, p. 23]. However, his match predicates are still variations of the fully universally quantified *proper* match (called *satisfies* by Penix) and—due to his component adaptation

framework—he does not consider any of the equivalence matches used for exact retrieval.

In the context of behavioral subtyping for object-oriented languages, K. Dhara and G. Leavens [DL96] a match predicate called *guarded generalized match* which has a fully universal prefix and a body of the form

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

However, this can be shown to be equivalent to proper match.

A. Mili, R. Mili, and R. Mittermeir [MMM94, MMM97] use a relational approach to specification matching which relies on the notion of *refinement* of relation. R refines S ($R \sqsupseteq S$) iff

$$(R \circ U) \cap (S \circ U) \cap (R \cup S) = S$$

where U is the universal relation over the domain. However, if this is translated into an explicit representation using pre- and postcondition expressions, it turns out to be equivalent to proper match. In [JD+97], L. L. Jilani et al. define approximate retrieval algorithms over a component library ordered by refinement. However, since the retrieved components depend on the induced topology over the library, a functionally equivalent match predicate cannot be defined.

J.-J. Jeng and B. Cheng [JC93, JC94] use a match predicate

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

which superficially looks the same as strict plug-in match (cf. Def. 3.3.4) but has a subtle difference: the subsumption relation $\sqsubseteq$ used here is not the usual logical implication relation. Instead, it is operationally defined as a modification of Chang and Lee's subsumption test algorithm [CL73] which checks logical implication. In this modified subsumption test algorithm, two complementary literals may be resolved against each other even if their top-level functors are not the same; they are, however required to belong to the same user-defined congruence class. This modification allows the retrieval system to deal easily with different naming conventions, e.g., $\neg is\_empty(l)$ can directly be resolved against $nil?(l)$. However, this is a double-edged sword and its disadvantages outweigh its advantages. It is not applied consistently but only for the top-level functors, the user-defined congruence relation on the literal symbols is non-transparent to and non-portable between different users, and, most importantly, it compromises the defining feature of deduction-based retrieval that only *proven matches* are retrieved.

The relation between different match predicates is traditionally characterized by the so-called "lattice of specification matches", e.g., [Moo96, Figure 3.5], [MW97b, Figure 4], [Pen98, Figure 3.2], or [CC99, Figure 1]. This structure is obtained by ordering the (propositional bodies of the) match predicates by implication. Y. Chen and B. Cheng [CC99] try to consolidate this work. They

introduce the concept of a *reuse-ensuring match* via the relational semantics of specifications and then show that the set of all reuse-ensuring matches is in fact a complete lattice with proper match as top element, i.e., proper match is the most general reuse-ensuring match. However, while this intuitive result is hardly surprising, their technical approach is severely flawed. Again, the flaw is caused by considering only the propositional structure of the bodies of the different match predicates—if different quantifier prefixes are taken into account, the set of all reuse-ensuring matches is no longer finite and thus a crucial assumption of their proof no longer holds.

Library Organization

Match predicates can be used to order components by generality (cf. Def. 2.2.5) and thus to organize a library hierarchically. A. Mili, R. Mili, and R. Mittermeir [MMM94, MMM97] describe such a library organization in which the components are stored in a lattice-like fashion, using relational subsumption (cf. above) as ordering in such a way that the most specific components become maximal elements. Their retrieval algorithms then exploit this hierarchical structure. For proper retrieval (which is exact retrieval in their terminology) they start with these maximal elements and proceed breadth-first, checking the nodes against the query until more general nodes are no longer subsumed by the query. The query result is then obtained as the (upward) transitive closure of the found minimal nodes. Hence, their algorithm is closed under iterated retrieval (cf. Def. 2.2.9).

A closer inspection shows that breadth-first search does not exploit the hierarchical structure to its fullest extent and is systematically worse than a depth-first approach, regardless of which maximal node the depth-first search starts. This follows from the fact that a depth-first search from one maximal node may preempt the search from another maximal node by reaching a common lower bound.

J.-J. Jeng and B. Cheng [JC93, JC94] describe a two-tiered library organization. The lower tier uses their modified subsumption test (cf. Page 176) to order the library components into—relatively shallow—disjoint clusters called *sets of lattices*. The upper tier applies a conventional hierarchical clustering algorithm to combine these sets of lattices into a single connected hierarchy. The clustering algorithm is a variant of Kruskal's minimum spanning tree algorithm [Kru56]. The edge weights represent the perceived semantic distance between two components. They are calculated from the component specifications by a syntax-directed distance measure. During retrieval, the coarse-grained upper tier of the hierarchy is then used to prune away large parts of the library. Only the remainder needs to be inspected more closely, again using the modified subsumption test. However, it is not clear how well-suited the syntax-directed distance measure is in practice, and thus how well this approach works.

In J. Penix's feature-based indexing method, as implemented in the REBOUND-system [PBA95, Pen98, PA99], a pre-defined set Φ of features is used to construct

an index. Φ is checked off-line against the library, using conditional plug-in as match condition, and each component is indexed with the set of all matching features. In contrast to the other two methods, feature-based indexing is an *external* library organization method, because it relies on the explicit and external set of features and not on any intrinsic relation between the library components.

Formal Retrieval Models

Deduction-based software component retrieval is a thoroughly formal activity. Oddly enough, however, there is only very little work in formalizing this activity. The survey article [MMM98] by A. Mili, R. Mili, and R. Mittermeir also contains the core of a thesaurus for the retrieval domain. This can also be considered as a first step towards a formal model of component retrieval. D. Eichmann [Eic92] has developed a preliminary framework for multi-modal retrieval systems (i.e., systems applying more than one retrieval mechanism) based on different sublattices for the different mechanisms.

S. Atkinson [Atk95, Atk97, Atk98] has developed an abstract retrieval framework using the Object-Z notation. This framework contains Z-schemas for all the key concepts and operations in component retrieval, e.g., keys, orderings, or component extraction. Atkinson then instantiates this framework for a number of different component retrieval methods, including faceted classification, specification matching, and behavior sampling. He then extends the base framework to model multi-modal retrieval mechanisms and describes NORA/HAMMR's pipeline architecture within this extended framework.

Retrieval Systems

Most work on deduction-based retrieval is of theoretical nature, or describes only proof-of-concept "implementations" which are experimentally validated only with a small number of selected example proofs, for example [RW91, CJ92, Moo96, MW97b]. These implementations cannot really be considered to be retrieval systems. R. Mili et al. [MMM97] describe an approach in which OTTER is used to check a larger number of tasks but that work again relies on a mostly manual set-up.

J. Penix's REBOUND-system [PBA95, Pen98, PA99] is the only other example of a working prototype which is mature enough for a significant experimental evaluation. REBOUND is a combined component retrieval and adaptation system. Its retrieval subsystem is based on *feature-based indexing* which can be considered as the semantic equivalent to the usual keyword-based classification schemes. Feature-based indexing uses a set Φ of first-order formulas or *features* which are assigned to the components and to the query if they are logical consequences of the specifications. Hence, $\phi_i \in \Phi$ is assigned to c if $pre_c \wedge post_c \Rightarrow \phi_i$ can be proven. Retrieval is then reduced to comparison of the derived feature sets: a component

can be retrieved either if its feature set is a subset of the query's feature set (*exact retrieval* in Penix's terminology) or if it has a non-empty intersection with the query's feature set (*approximate retrieval*).

The original REBOUND-system uses the HOL-prover [GM93] for classification. A later re-implementation (under the name SOCCER) relies on PVS [ORS91]; this technology is currently commercialized for the digital signal processing domain by EDAptive Computing, Inc. [EDA]. Both HOL and PVS are interactive, higher-order systems and have been "tuned" by adding domain-specific tactics similar to the rewrite-based simplifications described in Chapters 5 and 6. Penix has employed a precursor of the test library used here to evaluate REBOUND in different set-ups. For his initial classification schema, comprising just three features, and proper match as relevance condition, he reports an average recall of $\overline{r} = 0.82$ and an average precision of $\overline{p} = 0.21$.[2] An extension of the classification scheme by two more features increases the average precision to $\overline{p} = 0.29$ but decreases the average recall to $\overline{r} = 0.69$. Retrieval times are consistently well below a second as each query induces only three (or five, respectively) relatively simple proofs.

In the design space spanned by precision, recall, and response time, feature-based indexing thus covers a position different from that of any of the filters in NORA/HAMMR. It is significantly faster than any other filter but for a fast rejection filter it has a relatively poor recall. Likewise, due to its low precision, it still requires a subsequent ATP-based filter.

10.3 Future Work

No system is ever finished and despite the significant efforts already invested into its implementation this is also true for NORA/HAMMR. Before NORA/HAMMR in particular and deduction-based software component retrieval in general are "ready for prime time" (i.e., can be used outside research labs), a number of improvements have to be made. Most of these improvements ultimately concern aspects of scaling-up deduction-based retrieval. In the following, I sketch some of the most important and interesting issues for future work.

10.3.1 Type-Based Retrieval

Type-based retrieval or *signature matching* has traditionally been considered as a computationally more tractable approximation of full deduction-based retrieval, interpreting types as approximations of full behavioral specifications. It has thus usually been applied only as a fast, independent prefilter. However, the discussions in Chapter 3 have shown that its role is more complicated and that a careful

[2] Standard deviations are not given; however, *min*- and *max*-values indicate that the standard deviations are as high as for NORA/HAMMR.

integration of type-based and full deduction-based retrieval is required. Future research must especially consider two important problems:

- How can type-based retrieval systematically be generalized to programming languages beyond the functional paradigm?
- How can type-based retrieval be used to construct type compatibility predicates systematically?

Both problems can be tackled by a systematic re-development of type-based retrieval within a strong type-theoretic framework, using *general type systems* [Car96]. Since type systems are purely syntactic calculi, they are also applicable to programming languages which do not have such nice semantic interpretations as functional languages. Moreover, an integration of type systems and R. DiCosmo's concept of constructive isomorphisms [DiC95] could be used as starting point for the automatic construction of type compatibility predicates.

10.3.2 Reduction Techniques

Complexity reduction is the key to a further scale-up of deduction-based retrieval. Reduction techniques can work either locally or globally. Local techniques try to reduce a single proof task at a time, e.g., as side effect of the simplification-based rejection filters described in chapters 5 and 6. Global techniques try to minimize the total proof effort required to achieve a certain recall level, e.g., by reordering the proof tasks such that tasks which are more likely associated with matches are preferred. Global reduction techniques offer more reduction potential but they are obviously harder to realize than local techniques. This subsection describes two different global techniques.

Feature-Based Indexing

Feature-based indexing can be integrated into NORA/HAMMR's pipeline; this is described in more detail in [FLP99]. The index can be used as an initial filter to reduce the number of emerging proof tasks "at the source". The global reduction effect follows from two facts. First, because the features are simpler than full queries, the proof tasks emerging during the indexing phase are relatively simple. Second, during retrieval, only the query itself needs to be indexed but not the library. This integration, however, requires some modifications because the feature-based indexing method is not only not recall-preserving but also possibly unsound. The cause for this unsoundness is that the failure to find a proof for the validity of a feature is identified with its invalidity. This identification is not correct in practice because the ATP is incomplete. Hence, if the ATP fails to derive a—necessary—feature for the query, too weak components (i.e., with too few features) may be retrieved.

This problem can be avoided if the feature set assigned to each component is divided into the *positive* features Φ^+ which are defined as above and the *negative* features Φ^- which can be *proven not to hold*. Hence, $\phi_i \in \Phi^-(c)$ iff $pre_c \wedge post_c \Rightarrow \neg\phi_i$. Obviously, the positive and negative feature sets of a non-trivial and implementable component c are always disjoint. It is then easy to see that a component c can (for the relevance condition of refinement) not be relevant if it has complementary feature to the query q, i.e., $\Phi^+(c) \cap \Phi^-(q) \neq \emptyset$ or $\Phi^-(c) \cap \Phi^+(q) \neq \emptyset$. These conditions can be used to improve recall as well as precision.

Abstract Behavior Sampling

The deduction-based techniques developed in this thesis can easily be extended to emulate *behavior sampling* [PP92, PP93b, PB97] within framework of a formal specifications (cf. Section 1.1.2 for a short overview of the basic ideas of behavior sampling). The global reduction effect of this *abstract behavior sampling* approach follows again from the fact that the emerging queries are simpler by construction than those emerging in full-fledged deduction-based retrieval.

In the most basic variant, a sample consisting of a single (*input, output*)-pair is simply specified as a (*pre, post*)-pair relating input- and output parameters of the specification; the datatypes of the specification language are used to construct the respective values. For example, the query

$$\begin{array}{l} \mathit{sample}\text{-}1\,(l : \mathit{list})\ r : \mathit{list} \\ \mathsf{pre}\ \ l = [1] \\ \mathsf{post}\ r = [] \end{array}$$

can be used to search for the *tail*-function. The specifications of sample and component are then matched against each other, using μ_{proper} as match relation.

Larger sample sets can also be specified, although the notation becomes slightly cumbersome: the precondition consists of a disjunction over all inputs, the postcondition has to relate each output-value to its respective input-value.

$$\begin{array}{l} \mathit{sample}\text{-}2\,(l : \mathit{list})\ r : \mathit{list} \\ \mathsf{pre}\ \ l = [1] \vee l = [1,2] \\ \mathsf{post}\ (l = [1]\ \Rightarrow\ r = []) \wedge (l = [1,2]\ \Rightarrow\ r = [2]) \end{array}$$

However, it is relatively simple to translate sample sets automatically into this form.

However, the main conceptual advantage of abstract behavior sampling over its implementation-based counterpart, is that it allows a smooth integration of concrete samples and abstract specifications, as shown by the abstract sample query

$$\begin{array}{l} \mathit{abstract\text{-}sample}\,(l : \mathit{list})\ r : \mathit{list} \\ \mathsf{pre}\ \ l \neq [\,] \end{array}$$

$$\mathsf{post}\ (l = [1,2,3]\ \Rightarrow\ r = [2,3]) \wedge \mathsf{len}\ l = \mathsf{len}\ r + 1$$

which combines a sample with a more abstract partial specification. It becomes even possible to model non-determinism which can in the implementation-based case only be achieved by "multiplying out" the different answers into different samples and using Boolean retrieval over this extended sample set.

10.3.3 Calculus and Prover Improvements

Another crucial—but more technical—aspect is to improve the core deductive machinery. While some improvements will happen quasi naturally by the general progress in the field of automated deduction, others are more specific to the characteristics of the emerging proof tasks. This subsection describes some of these domain-specific improvements.

Decision Procedures

Decision procedures replace search by calculation and can thus construct proofs "blindingly fast" [Bjø98]. Their usefulness has been proven in several areas, e.g., theorem proving (PVS [ORS91]), program verification (ESC [LN98]), or program synthesis (Meta-Amphion, [VR98]), and it can be expected that they improve deduction-based retrieval as well.

Decision procedures can be built directly into the theorem provers; however, this requires usually substantial modifications of the provers (and likely even their underlying calculus), as only very few provers have been developed with such theory extensions in mind. Even worse, this effort has to be repeated fore every prover.

Alternatively, decision procedures can be built into the rewrite machinery used for simplification and rejection. In fact, the applied rewrite rules for equality and freely generated datatypes are already very similar to the *congruence closure algorithms* [Sho78, Sho79, NO80, Opp80]. Replacing these rules by a specialized implementation should thus yield a significant speed-up. N. Bjørner has extended R. Shostak's algorithm [Sho78] to handle the quantifier-free theory of queues with prefix-, subqueue-, and membership-predicates and tested it on the original proof tasks [Bjø98]. However, since the tasks are not quantifier-free, his algorithm becomes incomplete but it still remains sound and terminating; moreover, it is very fast. A confirmation filter based on that implementation yields an average recall of $\underline{r} = 66.70\%$ (or 1226 proofs) within an average query response time of $\overline{T}_{query} = 5.84$ secs. ($\sigma_T = 3.29$ secs.) This compares very favorable with the rewrite-based filter (cf. Table 5.3).

Abstract Model Checking

Model checking has been highly successful for the verification of parallel and distributed systems and programs. This raises the question whether and how it could be used for deduction-based retrieval. Since model checking can usually find counterexamples for invalid proof tasks quickly, it is natural to apply it in a rejection filter. However, since model checking is essentially a propositional method, it terminates only if the domains of all sorts occurring in the proof task are finite. This is clearly not the case for the test library nor for most useful component libraries.

Finiteness can be enforced by the application of appropriate *abstractions*. Here, the basic idea is the same as in *abstract interpretation* of computer programs [CC77]. The infinite concrete domain is replaced by a finite abstract domain. Each abstract domain element represents an equivalence class. Lists for example may be abstracted into the classes $nil = \{[\,]\}$ and $non\text{-}nil = \{x \mid x \neq [\,]\}$. The choice of the equivalence relation is a crucial step of the abstraction process. The concrete functions and predicates are replaced by abstract counterparts which work on the equivalence classes. Determining these abstract counterparts is, however, even harder than choosing appropriate abstract domains. This abstraction process usually introduces imprecisions, either in the form of *underapproximations* or in the form of *overapproximations*. Depending on the introduced imprecisions and the form of the proof task, the rejection filter can then become unsound, incomplete, or both.

However, even the approximated finite domains can still cause practical problems. Since model checking is essentially propositional, first-order tasks are translated into a shallow relational form, i.e., nested subterms are replaced by variables, n-ary function symbols are replaced by $(n+1)$-ary predicate symbols, and the clause set is extended by new clauses which ensure the well-definedness of the fresh predicates. The extended clause set is then ground-instantiated over the finite domains; each distinct ground atom is replaced by a unique propositional variable. This instantiation step may cause an exponential blow-up as a clause with m variables produces m^n ground instances, assuming a domain size m.

In practice, blow-up and imprecision are limiting the suitability of rejection filters based on model checking (cf. [FSS98, FLP99] for more details). Even with very small domains (i.e., $|\,item\,| = 1$ and $|\,list\,| = 2$), MACE's [McC94a] memory limits are exceeded for more than 6% of the proof tasks. For $T_{max} = 0.5$ secs., almost 40% of the remaining tasks cannot be completed within the time limit. Together, this yields an average recall of $\underline{r} = 66.65\%$ and an average precision of $\underline{p} = 19.52\%$. Hence, rejection filters based on model checking are worse than those based on rewriting (cf. Chapters 5 and 6). This is also reflected in a relatively small precision leverage of $\underline{\delta_p} = 1.50$ and a relatively high relative defect ratio of $\underline{\delta_e} = 0.60$. However, abstract model checking is currently a very active research topic (e.g., [GS97, Sai99, VPP00]) and new, more precise and more aggressive

abstraction techniques could make rejection filters based on model checking more practical.

Sort Structures and Sort Encodings

The lemma selection experiments described in Section 9.5 have demonstrated that sort encodings can have a major impact on the performance of the ATPs and thus on the retrieval results. It may be expected that different sort structures (e.g., employing subsorts) have similar impacts. To investigate and quantify these impacts, the experimental evaluation presented in this thesis could be repeated with an accordingly recoded domain theory and library.

Two especially interesting questions are whether finer-grained sort structures are beneficial at all and whether SPASS with its built-in sort support can profit disproportionally from a such structures or whether term encodings remain competitive. If finer-grained sort structures prove to be beneficial, general techniques should be developed which extract the optimal sort structure from a domain theory and automatically transform the theory appropriately. This can be considered as building-in a decision procedure for sort reasoning, similar in spirit to the Meta-Amphion approach [Roa97, RVL97].

Simulating Induction

The inductive structure of generated datatypes means that pure (i.e., non-inductive) first-order theorem provers are theoretically incomplete and that inductive provers must be used. In practice, however, proper inductive provers are too general (i.e., too weak) for the emerging proof tasks and rely too much on interaction. Hence, it is necessary to find heuristics for first-order approximations of a proper induction.

The two main problems are to identify suitable induction variables and induction schemas. Due to the regular structure of the proof tasks and the meta-level information provided in the lemma library, these problems can be solved in a straightforward way. Since the query should ultimately guide the proof, its inductively defined parameters are good candidates for the use as induction variables. Similarly, simple structural induction is a good candidate for the use as induction schema because it does not require any semantic analysis of the proof tasks and can be derived automatically from the generator functions. A further simplification can be achieved if the proper structural induction schema is replaced by the surjective unrolling schema (cf. Def. 5.18). This "poor man's induction" has the advantage of generating smaller formulas in the step case, as the original formula need not be duplicated in the induction hypothesis and conclusion. However, it is an obviously even more incomplete approximation.

J. Schumann [Sch00] has used SETHEO to experiment with such induction approximations, using the same proof tasks as in chapters 7-9. A preprocessing

step applies the induction schema and splits the original task into multiple independent tasks; each of these tasks is then processed further by NORA/HAMMR. Due to differences in the experimental set-up, the results cannot be compared directly. However, they clearly indicate that such approximations are a feasible approach to the induction problem. They also indicate that a competition over the different induction schemas improves the efficiency of the approach further.

Induction approximation can also be used to build an unsound filter by retrieving components for which at least one of the associated inductive (sub-) tasks can be solved. Such filters should be well-suited for datatypes which are generated by relatively large signatures, e.g., enumeration types or syntax trees.

Parallel Architectures

As long as the pipeline does not contain a strong indexing schema at the source, the number of proof tasks emerging for a single query grows in step with the size of the library. A scale-up to very large libraries (i.e., thousands of components) can then only be achieved by distributing the proof tasks over a large number of processors. This is possible at two different levels of granularity.

At the coarser level, the proof tasks are considered as atomic units, i.e., a single proof task is executed on a single processor only. This requires a centralized "scheduling filter" which distributes the tasks and collects the results but no modification of the provers. In a experimental extension of NORA/HAMMR the ILF-system [DG$^+$97] was used as scheduler. The experiments (cf. [BF98, BFF99] for details) showed that the scheduling overhead is generally insignificant. Thus, this brute-force parallelization approach can be used to scale-up deduction-based retrieval.

At the finer level, the proof tasks are no longer considered as atomic units, i.e., a single proof task is distributed over a number of processors. This approach is technically more challenging because it requires communication between the different processors working on the same task in order to provide any further improvements over the coarse-grained parallelization. Some ATPs as for example PARTHEO [SL90] (which is a parallel version of SETHEO) already provide the necessary communication infrastructure. These ATPs can than again be used as black boxes. For other systems, a dedicated communication infrastructure is required. J. Denzinger and D. Fuchs [DF98] describe a system which allows parallelization and cooperation of different ATPs with only minor modifications of the provers. Such a system can be used as "meta-filter" in NORA/HAMMR, comprising multiple cooperating provers.

Moreover, the fine-grained parallelization approach has even more potential to improve the retrieval performance (i.e., the recall) than the coarse-grained approach, or in other words, cooperation beats competition. The reason for this is precisely the communication: cooperating provers exchange solved subgoals which may help a stuck prover to complete the proof. Preliminary experiments

[BF98, BFF99] show that a cooperation of different ATPs can improve the recall over a simple competition between the same ATPs by as much as almost 10%-points.

10.3.4 Integration with other Formal Techniques

Deduction-based retrieval has traditionally been investigated in isolation, and this thesis followed the tradition. This isolated view, however, ignores synergies which can result from combinations with other formal software development methods, e.g., program verification or program synthesis. These synergies can help to offset the relatively high up-front investments required to introduce deduction-based retrieval.

Integration into Program Verification

The major conceptual advantage of deduction-based retrieval is that it retrieves *proven* matches only. This conceptual advantage can unfold its full practical potential by an integration into a Hoare-style program verification system, e.g., the *Modula Proving System* (MOPS) [Kai98, KFS00].

The basic idea of such a combination is that the Hoare-triples, traditionally written as $\{P\}S\{Q\}$ [Apt81], can also be considered as (almost complete) queries. Since, given P, a terminating execution of S yields Q, the pre/post-pair (P, Q) can also be used to search for S if it is still unknown. The unique benefit of such a combination is that it lifts reuse from the pure code level to the proof level, provided that the library components are verified against their specifications.

The integration of NORA/HAMMR into MOPS is thus a promising project. It is made easier by the deliberate choice of VDM-SL as common contract language for both systems. MOPS uses *formal comments* to embed specifications into the source code such that the annotated program can still be compiled and executed by any Modula-2 compiler.

```
(*{ entry sum_loop
     pre  sum  = 0
     post sum' = n * (n+1) div 2      }*)
(*{ loopinv sum = ((i - 1) * i) div 2 }*)
FOR i := 1 TO n DO
    sum := sum + i;
END;
(*{ exit sum_loop }*)
```

Figure 10.3: Embedded specifications in MOPS

`entry`/`exit`-tags as shown in Figure 10.3 mark the verification segments; these

can be nested to break large proofs into separate, manageable pieces or—in combination with NORA/HAMMR—to decompose a large "gap" into separate, simple queries. The verification segments are, however, not yet suitable as queries because they do not describe proper components. The code segments are tightly coupled ("inlined") with their environments and lack explicit interfaces. For retrieval purposes, the verification segments need to be modified to include such an interface. This can be achieved by extending the simple `entry`-tag into a full VDM function- or operation signature, e.g., `sum_loop(n:int) ext rw sum:int`. These modified verification segments can then be used as hooks for deduction-based retrieval; the queries can be extracted from the annotated source file and submitted to NORA/HAMMR. However, a working integration still requires substantial infrastructure work.

Integration into Program Synthesis

The combination of implementation, verification, and retrieval described above can be considered as an ad-hoc program synthesis approach. A more principled approach is *deductive synthesis* which is going back to [Gre69, Wal69]. It is based on the Curry-Howard isomorphism [How80] or "proofs-as-programs"-paradigm which asserts that a constructive proof of a specification is equivalent to a functional program which is correct with respect to this specification. Its main problem, however, is that it does not scale up very well, as for example observed by [Kre98]:

> "The main problem of general approaches to program synthesis is that they force the synthesis system to derive an algorithm almost from scratch ... "

The integration of a specialized deduction-based retrieval subsystem effectively allows synthesis to bottom-out in previously synthesized components instead of the built-in operators of the language only and thus helps to raise the level of granularity. The main problem of such an integration is to prevent overloading of the retrieval subsystem with an excessive number of equivalent or redundant queries.

One integration approach, which is described in more detail in [FW99], is based on the deductive tableau approach [MW80, MW92], or more precisely, on its higher-order logic reinterpretation as given by A. Ayari and D. Basin [AB96, AB99]. In this interpretation, higher-order variables are used to represent the program fragments yet to be synthesized, similar to the meta-variables in the Hoare-triples. The tableau is represented by the *proof state*, as shown in Figure 10.4.

The integration exploits the idea that the introduction of a new meta-variable marks a *substantial change* of the proof state and thus warrants a new attempt to retrieve components. The technical challenge is to extract the *first-order* queries

$$
\begin{array}{l}
\mathcal{A}_1 \wedge \ldots \wedge \mathcal{A}_n \rightarrow \mathcal{SPEC} \\
1) \quad [\![\mathcal{H}_{1,1}; \quad \ldots \quad ; \mathcal{H}_{1,n_1}]\!] \quad \Longrightarrow \quad \mathcal{G}_1 \\
\qquad\qquad\qquad \vdots \\
m) \quad [\![\mathcal{H}_{m,1}; \quad \ldots \quad ; \mathcal{H}_{m,n_m}]\!] \quad \Longrightarrow \quad \mathcal{G}_m
\end{array}
$$

Figure 10.4: Higher-order representation of deductive tableaus

from the *higher-order* proof state. This can of course not be automated completely because higher-order logic is more expressive than first-order logic but good approximations can be found based on the restricted structure of the proof state. One of the remaining open problems is the construction of an appropriate signature for the query, similar to the problem faced in integrating retrieval and verification.

This integration can solve one of the most difficult conceptual problems of pure retrieval-oriented reuse approaches, namely, what to do when no perfectly matching components can be found for a particular query. It supports the the adaptation of retrieved "near-misses" in a clean, unified formal framework for software design and reuse.

Bibliography

[A+93] D. J. Andrews et al. Information Technology Programming Languages – VDM-SL: First Committee Draft Standard CD1387-1. Technical Report ISO/IEC JTC1/SC22/WG19 N-20, International Standards Organization, November 1993.

[AB96] A. Ayari and D. Basin. "Generic System Support for Deductive Program Development". In T. Margaria and B. Steffen, (eds.), *Proceedings of the 2nd International Workshop on Tools and Algorithms for the Construction and Analysis of Systems*, *Lecture Notes in Computer Science* **1055**, pp. 313–328, Passau, March 1996. Springer.

[AB99] A. Ayari and D. Basin. A Higher-Order Interpretation of Deductive Tableau. Technical report, Univ. of Freiburg, Germany, 1999.

[AC+91] M. Abadi, L. Cardelli, P.-L. Curien, and J.-J. Lèvy. "Explicit Substitutions". *Journal of Functional Programming*, **1**(4):375–416, 1991.

[Ahr00] W. Ahrendt. "A Basis for Model Computation in Free Data Types". In P. Baumgartner, C. Fermüller, N. Peltier, and H. Zhang, (eds.), *Proceedings of the CADE-17 Workshop on Model Computation - Principles Algorithms, Applications*, Pittsburgh, PA, July 2000. To appear.

[Ame90] P. America. "Designing an Object-Oriented Programming Language with Behavioural Subtyping". In J. W. de Bakker, W. P. de Roever, and G. Rozenberg, (eds.), *Proceedings of the REX School/Workshop on Foundations of Object-Oriented Languages*, *Lecture Notes in Computer Science* **489**, pp. 60–90. Springer, Noordwijkerhout, The Netherlands, 1990.

[Apt81] K. R. Apt. "Ten Years of Hoare's Logic: A Survey—Part I". *ACM Transactions on Programming Languages and Systems*, **3**:431–483, 1981.

[Atk95] S. R. Atkinson. A Unifying Model for Retrieval from Reusable Software Libraries. Technical Report 9l-41, Software Verification Research Centre, Dept. of Computer Science, Univ. of Queensland, 1995.

[Atk97] S. R. Atkinson. *Formal Engineering of Software Library System.* PhD thesis, University of Queensland, School of Information Technology, 1997.

[Atk98] S. R. Atkinson. "Modelling Formal Integrated Component Retrieval". In P. Devanbu and J. Poulin, (eds.), *Proceedings of the Fifth International Conference on Software Reuse*, pp. 337–346, Victoria, Canada, June 1998. IEEE Computer Society Press.

[Bau96] P. Baumgartner. "Linear and Unit-Resulting Refutations for Horn Theories". *Journal of Automated Reasoning*, **16**(3):241–319, June 1996.

[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.

[BF+93] J. C. Bicarregui, J. S. Fitzgerald, P. A. Lindsay, R. Moore, and B. Ritchie. *Proof in VDM: A Practitioner's Guide*. FACIT series. Springer-Verlag, Berlin, 1993.

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

[BF98] T. Baar and B. Fischer. "Solving Software Reuse Problems with Theorem Provers". In P. Baumgartner, U. Fuhrbach, M. Kohlhaase, W. McCune, W. Reif, M. Stickel, and T. Uribe, (eds.), *Proceedings of the CADE-15 Workshop on Problem Solving Methodologies with Automated Deduction*, pp. 217–247, Lindau, July 1998.

[BFF98] T. Baar, B. Fischer, and D. Fuchs. "Experiments with ATP Integration in a Software Engineering Application". In N. S. Bjørner, R. Hähnle, W. Menzel, W. Reif, and P. H. Schmitt, (eds.), *Proceedings of the CADE-15 Workshop on Integration of Deduction Systems*, pp. 19–27, Lindau, July 1998.

[BFF99] T. Baar, B. Fischer, and D. Fuchs. "Integrating Deduction Techniques in a Software Reuse Application". *J. Universal Computer Science*, **5**(3):52–72, 1999.

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

[BHR89] K. H. Bläsius, U. Hedtstück, and C.-R. Rollinger, (eds.). *Sorts and Types in Artificial Intelligence*, *Lecture Notes in Artifical Intelligence* **418**. Springer, April 1989.

[Bir35] G. Birkhoff. "On the Structure of Abstract Algebras". *Proceedings of the Cambridge Philosophical Society*, **31**:433–454, 1935.

[Bjø98] N. S. Bjørner. *Integrating Decision Procedures for Temporal Verfication*. PhD thesis, Stanford University, November 1998.

[BMM92] N. Boudriga, A. Mili, and R. Mittermeir. "Semantic-Based Software Retrieval to Support Rapid Prototyping". *Structured Programming*, **13**:109–127, 1992.

[BN92] S. M. Brien and J. E. Nicholls. Z Base Standard. Technical report, International Standards Organization, November 1992. Accepted for standardization under ISO/IEC JTC1/SC22. Also available as Technical Monograph PRG-107, Oxford University Computing Laboratory.

[BN98] F. Baader and T. Nipkow. *Term Rewriting and all that.* Cambridge University Press, Cambridge, 1998.

[Boo87] G. Booch. *Software Components with Ada: Structures, Tools, and Subsystems.* Benjamin-Cummings, Menlo Park, Calif., 1987.

[Bör95] J. Börstler. "Feature-oriented Classification for Software Reuse". In *Proceedings of the 7th International Conference on Software Engineering and Knowledge Engineering*, pp. 204–211, Rockville, MD, June 1995.

[BR69] J. N. Buxton and B. Randell, (eds.). *Software Engineering Techniques: Report on a Conference.* NATO Scientific Affairs Division, Brussels, 1969.

[Bra75] D. Brand. "Proving Theorems with the Modification Method". *SIAM Journal on Computing*, **4**(4):412–430, December 1975.

[Bro75] F. P. Brooks. *The Mythtical Man-Month.* Addison-Wesley, Reading, Mass., 1975.

[BS98] W. Bibel and P. H. Schmitt, (eds.). *Automated Deduction - A Basis for Applications.* Kluwer, Dordrecht, 1998.

[Bun94] A. Bundy, (ed.). *Proceedings of the 12th International Conference on Automated Deduction*, *Lecture Notes in Artifical Intelligence* **814**, Nancy, June-July 1994. Springer.

[Car96] L. Cardelli. "Type Systems". In A. B. Tucker, (ed.), *Handbook of Computer Science and Engineering.* CRC Press, 1996.

[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 *Proceedings of the 4th Annual ACM Symposium on Principles of Programming Languages*, pp. 238–252, Los Angeles, California, January 1977. ACM Press.

[CC99] Y. Chen and B. H. C. Cheng. "A Semantic Foundation for Specification Matching". In Sitaraman and Leavens [SL99].

[CJ92] B. H. C. Cheng and J. Jeng. "Using Automated Reasoning to Determine Software Reuse". *International Journal of Software Engineering and Knowledge Engineering*, **2**(4):523–546, December 1992.

[CL73] C.-L. Chang and R. C.-T. Lee. *Symbolic Logic and Mechanical Theorem Proving*. Academic Press, New York, 1973.

[Cle86] J. C. Cleaveland. "Building Application Generators". *IEEE Software*, **5**(4):25–33, July 1986.

[Dah96] B. I. Dahn. The ILF Input Language. Unpublished report, HU Berlin, May 1996.

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

[DB+91] P. Devanbu, R. J. Brachman, P. G. Selfridge, and B. W. Ballard. "LaSSIE: a knowledge-based software information system". *Communications of the ACM*, **34**(5):34–49, May 1991.

[DF98] J. Denzinger and D. Fuchs. "Enhancing conventional search systems with multi-agent techniques: a case study". In *Proceedings of theInternational Conference on Multi Agent Systems (ICMAS'98)*, 1998. To Appear.

[DG+97] B. I. Dahn, J. Gehne, T. Honigmann, and A. Wolf. "Integration of Automated and Interactive Theorem Proving in ILF". In McCune [McC97], pp. 57–60.

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

[DJ90] N. Dershowitz and J.-P. Jouannaud. "Rewrite Systems". In J. van Leeuwen, (ed.), *Handbook of Theoretical Computer Science B: Formal Methods and Semantics*, pp. 243–320. North-Holland, Amsterdam, 1990.

[DJ94] P. T. Devanbu and M. A. Jones. "The Use of Description Logics in KBSE Systems". In Fadini [Fad94], pp. 23–38.

[DL96] K. K. Dhara and G. T. Leavens. "Forcing Behavioral Subtyping Through Specification Inheritance". In Maibaum and Zelkowitz [MZ96], pp. 258–267.

[DLL62] M. Davis, G. Logemann, and D. W. Loveland. "A Machine Program for Theorem Proving". *Communications of the ACM*, **5**(3):394–397, July 1962.

[DP60] M. Davis and H. Putnam. "A Computing Procedure for Quantification Theory". *Journal of the ACM*, **7**(3):201–215, July 1960.

[EDA] `http://www.edaptive.com/`.

[Eic92] D. Eichmann. "Supporting Multiple Domains in a Single Reuse Repository". In *Proceedings of the 4th International Conference on Software Engineering and Knowledge Engineering*, Capri, Italy, June 1992. IEEE Computer Society Press.

[Fad94] B. Fadini, (ed.). *Proceedings of the 16th International Conference on Software Engineering*, Sorrento, Italy, May 1994. IEEE Computer Society Press.

[FF98] M. Fuchs and M. Fuchs. "Feature-based Learning of Search-guiding Heuristics for Theorem Proving". *AI Communications*, **11**, 1998.

[FG89] W. B. Frakes and P. B. Gandel. "Classification, Storage, and Retrieval of Reusable Components". In *Proceedings of the Twelfth Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pp. 251–254, 1989.

[Fis98] B. Fischer. "Specification-Based Browsing of Software Component Libraries". In D. F. Redmiles and B. Nuseibeh, (eds.), *Proceedings of the 13th International Conference on Automated Software Engineering*, pp. 74–83, Honolulu, Hawaii, October 1998. IEEE Computer Society Press.

[Fis00] B. Fischer. "Specification-Based Browsing of Software Component Libraries". *Automated Software Engineering*, **7**(2):179–200, 2000.

[FKS95a] B. Fischer, M. Kievernagel, and G. Snelting. "Deduction-Based Software Component Retrieval". In J. Köhler, F. Giunchiglia, C. Green, and C. Walther, (eds.), *Working Notes of the IJCAI-95 Workshop: Formal Approaches to the Reuse of Plans, Proofs, and Programs*, pp. 1–5, Montréal, August 1995.

[FKS95b] B. Fischer, M. Kievernagel, and W. Struckmann. "High-precision retrieval for high-quality software". In I. M. Marshall, W. B. Samson, and D. G. Edgar-Nevill, (eds.), *Proceedings of the 4th Software Quality Conference*, pp. 80–88, Dundee, July 1995. University of Abertay Dundee.

[FKS95c] B. Fischer, M. Kievernagel, and W. Struckmann. "VCR: A VDM-based Software Component Retrieval Tool". In M. Wirsing, (ed.), *Working Notes of the ICSE-17 Workshop on Formal Methods Application in Software Engineering Practice*, pp. 30–38, Seattle, Wash., April 1995.

[FLP99] B. Fischer, M. Lowry, and J. Penix. "Intelligent Component Retrieval via Automated Reasoning". In *Proc. AAAI-99 Workshop on Intelligent Software Engineering*, Orlando, FL, July 1999.

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

[FPC89] *Proceedings of the 4th Conference on Functional Programming Languages and Computer Architecture*, London, September 1989. ACM Press.

[Fra92] W. B. Frakes. "Introduction to Information Storage and Retrieval Systems". In W. B. Frakes and R. Baeza-Yates, (eds.), *Information Retrieval: Data Structures & Algorithms*, pp. 1–12. Prentice-Hall, Englewood Cliffs, NJ, 1992.

[FSS98] B. Fischer, J. M. P. Schumann, and G. Snelting. "Deduction-Based Software Component Retrieval". In Bibel and Schmitt [BS98], pp. 265–292.

[Fuc95] M. Fuchs. "Learning proof heuristics by adapting parameters". In A. Prieditis and S. J. Russell, (eds.), *Proceedings of the 12th International Conference on Machine Learning*, pp. 235–243, Tahoe City, California, July 1995. Morgan Kaufmann.

[Fuh95] N. Fuhr. Information Retrieval. Lecture Notes, University of Dortmund, 1995.

[FW99] B. Fischer and J. Whittle. "An Integration of Deductive Retrieval into Deductive Synthesis". In Hall and Tyugu [HT99], pp. 52–61.

[GH86] J. V. Guttag and J. J. Horning. "Report on the Larch Shared Language". *The Science of Computer Programming*, **6**(2):103–134, March 1986.

[GH+96] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley, Reading, MA, 1996.

[GHW85] J. V. Guttag, J. J. Horning, and J. M. Wing. "The Larch Family of Specifications Languages". *IEEE Software*, pp. 24–36, September 1985.

[GM93] M. J. C. Gordon and T. F. Melham. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic*. Cambridge University Press, 1993.

[GMW97] H. Ganzinger, C. Meyer, and C. Weidenbach. "Soft Typing for Ordered Resolution". In McCune [McC97], pp. 321–335.

[GN+96] J. Goguen, D. Nguyen, J. Meseguer, Luqi, D. Zhang, and V. Berzins. "Software Component Search". *Journal of Systems Integration*, **6**(1/2):93–134, 1996.

[Gog85] J. Goguen. "Suggestions for Using and Organizing Libraries in Software Development". In S. Kartashev and S. Kartashev, (eds.), *Proceedings of the First International Conference on Supercomputing Systems*, pp. 349–360. IEEE Computer Society, 1985. Also in *Supercomputing Systems*, Steven Kartashev and Svetlana Kartashev (eds.), Elsevier, 1986.

[Gra96] P. Graf. *Term Indexing*, *Lecture Notes in Computer Science* **1053**. Springer, 1996.

[Gre69] C. Green. "Application of Theorem Proving to Problem Solving". In Walker and Norton [WN69], pp. 219–240.

[GS92] D. Garlan and M. Shaw. "An Introduction to Software Architecture". In V. Ambriola and G. Tortora, (eds.), *Advances in Software Engineering and Knowledge Engineering*, pp. 1–40. World Scientific Publishing Co., 1992.

[GS97] S. Graf and H. Saidi. "Construction of Abstract State Graphs with PVS". In O. Grumberg, (ed.), *Proceedings of the 9th Conference on Computer-Aided Verification*, *Lecture Notes in Computer Science* **1254**, pp. 72–83. Springer, June 1997.

[Hal93] R. J. Hall. "Generalized Behavior-based Retrieval". In V. R. Basili, (ed.), *Proceedings of the 14th International Conference on Software Engineering*, pp. 371–380, Baltimore, Maryland, 1993. IEEE Computer Society Press.

[Har95] J. Harrison. "Inductive definitions: automation and application". In P. J. Windley, T. Schubert, and J. Alves-Foss, (eds.), *Proceedings of the 8th InternationalWorkshop on Higher Order Logic Theorem Proving and Its Applications*, *Lecture Notes in Computer Science* **971**, pp. 200–213, Aspen Grove, Utah, 1995. Springer.

[Hen94] S. Henninger. "Using Iterative Refinement to Find Reusable Software". *IEEE Software*, **11**(5):48–59, September 1994.

[Hen96] S. Henninger. "Supporting the construction and evolution of component repositories". In Maibaum and Zelkowitz [MZ96], pp. 279–288.

[HKW96] R. Hähnle, M. Kerber, and C. Weidenbach. Common Syntax of DFG-Schwerpunktprogramm "Deduktion". Interner Bericht 10/96, Universität Karlsruhe, Fakultät für Informatik, 1996.

[HM91] R. Helm and Y. S. Maarek. "Integrating Information Retrieval and Domain Specific Approaches for Browsing and Retrieval in Object-Oriented Class Libraries". In A. Paepcke, (ed.), *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages and Applications*, pp. 267–295, Phoenix, Arizona, 1991.

[How80] W. Howard. "The Formulas-as-Types Notion of Construction". In J. P. Seldin and J. R. Hindley, (eds.), *To H. B. Curry: Essays on Combinatory Logic, Lambda-Calculus, and Formalism*, pp. 479–490. Academic Press, 1980.

[Hsi85] J. Hsiang. "Refutational Theorem Proving using Term-Rewriting Systems". *Artificial Intelligence*, **25**:255–300, 1985.

[HT99] R. J. Hall and E. Tyugu, (eds.). *Proceedings of the 14th International Conference on Automated Software Engineering*, Cocoa Beach, Florida, October 1999. IEEE Computer Society Press.

[JC93] J. Jeng and B. H. C. Cheng. "Using formal methods to construct a software component library". In I. Sommerville and M. Paul, (eds.), *Proceedings of the 4th European Software Engineering Conference*, *Lecture Notes in Computer Science* **717**, pp. 397–417, Garmisch-Partenkirchen, September 1993. Springer.

[JC94] J.-J. Jeng and B. H. C. Cheng. "A Formal Approach to Using More General Components". In R. Jullig, (ed.), *Proceedings of the 9th Knowledge-Based Software Engineering Conference*, pp. 90–97, Monterey, CA, September 20-23 1994. IEEE Computer Society Press.

[JD+97] L. L. Jilani, J. Desharnais, M. Frappier, R. Mili, and A. Mili. "Retrieving Software Components that Minimize Adaptation Effort". In Lowry and Ledru [LL97], pp. 255–262.

[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.

[Kai98] T. Kaiser. Behandlung von Datenstrukturen in einem VDM-basierten Prädikatentransformer für Modula-2. Master's thesis, Technical University Braunschweig, September 1998.

[Käu88] T. Käufl. *Vereinfachung logischer Formeln in einem Vorbeweiser*. PhD thesis, University of Karlsruhe, February 1988.

[KFS00] T. Kaiser, B. Fischer, and W. Struckmann. "MOPS: Verifying Modula-2 programs specified in VDM-SL". In *Proceedings of the 4th Workshop Tools for System Design and Verification*, pp. 163–167, Reisensburg, July 2000.

[Klo92] J. W. Klop. "Term Rewriting Systems". In S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum, (eds.), *Handbook of Logic in Computer Science 2*, pp. 1–117. Oxford University Press, Oxford, 1992.

[Kre98] C. Kreitz. "Program Synthesis". In Bibel and Schmitt [BS98], pp. 105–134.

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

[Kru56] J. B. Kruskal. "On the shortest spanning subtree of a graph and the travelling salesman problem". *Proceedings of the American Mathematical Society*, **7**:48–50, 1956.

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

[KST97] S. Kahrs, D. Sannella, and A. Tarlecki. "The definition of Extended ML: a gentle introduction". *Theoretical Computer Science*, **173**(2):445–484, February 1997.

[LEW96] J. Loeckx, H.-D. Ehrich, and M. Wolf. *Specification of Abstract Data Types*. John Wiley / B. G. Teubner, New York, USA, 1996.

[LL97] M. Lowry and Y. Ledru, (eds.). *Proceedings of the 12th International Conference on Automated Software Engineering*, Lake Tahoe, November 1997. IEEE Computer Society Press.

[LN98] K. R. M. Leino and G. Nelson. "An extended static checker for Modula-3". In K. Koskimies, (ed.), *Compiler Construction (CC'98)*, pp. 302–305, Lisbon, 1998.

[Lov68] D. W. Loveland. "Mechanical Theorem Proving by Model Elimination". *Journal of the ACM*, **15**(2):236–251, April 1968.

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

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

[LV95] M. Lowry and J. Van Baalen. "Meta-Amphion: Synthesis of Efficient Domain-Specific Program Synthesis Systems". In Setliff [Set95], pp. 2–10.

[LV97] M. Lowry and J. Van Baalen. "Meta-Amphion: Synthesis of Efficient Domain-Specific Program Synthesis Systems". *Automated Software Engineering*, **4**(2):199–247, 1997.

[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.

[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*, **17**(8):800–813, 1991.

[McC94a] 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.

[McC94b] W. W. McCune. OTTER 3.0 Reference Manual and Guide. Technical report ANL-94-6, Argonne National Laboratory, Argonne, IL, USA, 1994.

[McC97] W. McCune, (ed.). *Proceedings of the 14th International Conference on Automated Deduction*, *Lecture Notes in Artifical Intelligence* **1249**, Townsville, July 1997. Springer.

[McI68] M. D. McIlroy. "Mass Produced Software Components". In Naur and Randell [NR68], pp. 138–150.

[Mel88] C. S. Mellish. "Implementing Systemic Classification by Unification". *Journal of Computational Linguistics*, **14**(1):40–51, 1988.

[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 *Proceedings of the 4th International Symposium 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 Fadini [Fad94], pp. 91–102.

[MMM97] A. Mili, R. Mili, and R. Mittermeir. "Storing and Retrieving Software Components: A Refinement-Based System". *IEEE Transactions on Software Engineering*, **23**(7):445–460, July 1997.

[MMM98] A. Mili, R. Mili, and R. Mittermeir. "A Survey of Software Reuse Libraries". *Annals of Software Engineering*, **5**:349–414, 1998.

[Moo96] A. Moorman Zaremski. *Signature and Specification Matching.* PhD thesis, Carnegie Mellon University, Pittsburgh, PA, January 1996.

[MR87] R. T. Mittermeir and W. Rossak. "Software Bases and Software Archives - Alternatives to support Software Reuse". In *Proceedings of the IEEE-ACM Fall Joint Computer Conference*, pp. 21–28, Dallas, TX, October 1987.

[MR90] R. T. Mittermeir and W. Rossak. "Reusability". In P. A. Ng and R. T. Yeh, (eds.), *Modern Software Engineering - Foundations and Current Perspectives*, pp. 205–235. Van Nostrand Reinhold, 1990.

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

[MW80] Z. Manna and R. J. Waldinger. "A Deductive Approach to Program Synthesis". *ACM Transactions on Programming Languages and Systems*, **2**(1):90–121, January 1980.

[MW92] Z. Manna and R. J. Waldinger. "Fundamentals of Deductive Program Synthesis". *IEEE Transactions on Software Engineering*, **18**(8):674–704, August 1992.

[MW93] A. Moorman Zaremski and J. M. Wing. "Signature Matching: A Key to Reuse". In D. Notkin, (ed.), *Proceedings of the 1st ACM SIGSOFT Symposium on the Foundation of Software Engineering*, pp. 182–190, Los Angeles, CA, December 1993. ACM, ACM Press.

[MW95a] A. Moorman Zaremski and J. M. Wing. "Signature Matching: A Tool for Using Software Libraries". *ACM Transactions on Software Engineering and Methodology*, pp. 146–170, 1995.

[MW95b] A. Moorman Zaremski and J. M. Wing. "Specification Matching of Software Components". In G. E. Kaiser, (ed.), *Proceedings of the 3rd ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pp. 6–17, Washington, DC, October 1995. ACM Press.

[MW97a] W. McCune and L. Wos. "Otter—The CADE-13 Competition Incarnations". *Journal of Automated Reasoning*, **18**(2):211–220, April 1997.

[MW97b] A. Moorman Zaremski and J. M. Wing. "Specification Matching of Software Components". *ACM Transactions on Software Engineering and Methodology*, **6**(4):333–369, October 1997.

[MZ96] T. Maibaum and M. Zelkowitz, (eds.). *Proceedings of the 18th International Conference on Software Engineering*, Berlin, March 1996. IEEE Computer Society Press.

[NO80] G. Nelson and D. C. Oppen. "Fast Decision Procedures Based on Congruence Closure". *Journal of the ACM*, **27**(2):356–364, April 1980.

[NP95] T. Nipkow and C. Prehofer. "Type Reconstruction for Type Classes". *Journal of Functional Programming*, **5**(2):201–224, 1995.

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

[NR68] P. Naur and B. Randell, (eds.). *Software Engineering: Report on a Conference*. NATO Scientific Affairs Division, Brussels, 1968.

[NRW98] A. Nonnengart, G. Rock, and C. Weidenbach. "On Generating Small Clause Normal Forms". In C. Kirchner and H. Kirchner, (eds.), *Proceedings of the 15th International Conference on Automated Deduction*, *Lecture Notes in Artifical Intelligence* **1421**, pp. 397–411, Lindau, July 1998. Springer.

[NS91] T. Nipkow and G. Snelting. "Type Classes and Overloading Resoloutin via Order-Sorted Unification". In *Proceedings of the 5th Conference on Functional Programming Languages and Computer Architecture*, *Lecture Notes in Computer Science* **523**, pp. 1–14. Springer, 1991.

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

[Obe89] A. Oberschelp. "Order Sorted Predicate Logic". In Bläsius et al. [BHR89], pp. 8–17.

[Opp80] D. C. Oppen. "Reasoning About Recursively Defined Data Structures". *Journal of the ACM*, **27**(3):403–411, July 1980.

[ORS91] S. Owre, J. Rushby, and N. Shankar. "PVS: A Prototype Verification System". In D. Kapur, (ed.), *Proceedings of the 11th International Conference on Automated Deduction*, *Lecture Notes in Artifical Intelligence* **607**, pp. 748–752, Saratoga Springs, NY, July 1991. Springer.

[PA89] R. Prieto-Diaz and G. Arango. *Domain Analysis: Acquistion of Reusable Information for Software Construction*. IEEE Computer Society Press, 1989.

[PA99] J. Penix and P. Alexander. "Efficient Specification-Based Component Retrieval". *Automated Software Engineering*, **6**(2):139–170, April 1999.

[Pau94] L. C. Paulson. *Isabelle: A Generic Theorem Prover*, *Lecture Notes in Computer Science* **828**. Springer, 1994.

[PB97] Park and Bai. Generating Samples for Component Retrieval by Execution. Technical report, University of Windsor, Windsor, Ontario, 1997.

[PBA95] J. Penix, P. Baraona, and P. Alexander. "Classification and Retrieval of Reusable Components Using Semantic Features". In Setliff [Set95], pp. 131–138.

[Pen98] J. Penix. *Automated Component Retrieval and Adaptation Using Formal Specifications*. PhD thesis, University of Cincinnati, April 1998.

[Per89] D. E. Perry. "The Inscape Environment". In *Proceedings of the 11th International Conference on Software Engineering*, pp. 2–12, Pittsburg, PA, May 1989. IEEE Computer Society Press.

[PL92] N. Plat and P. G. Larsen. "An Overview of the ISO/VDM-SL Standard". *ACM SIGPLAN Notices*, **27**(8):76–82, August 1992.

[Plo72] G. Plotkin. "Building in Equational Theories". In B. Meltzer and D. Michie, (eds.), *Machine Intelligence 7*, pp. 73–90, Edinburgh, Scotland, 1972. Edinburgh University Press.

[PP92] A. Podgurski and L. Pierce. "Behavior Sampling: A Technique for Automated Retrieval of Reusable Components". In *Proceedings of the 14th International Conference on Software Engineering*, pp. 349–360. IEEE Computer Society Press, 1992.

[PP93a] D. E. Perry and S. S. Popovitch. "Inquire: Predicate-Based Use and Reuse". In *Proceedings of the 8th Knowledge-Based Software Engineering Conference*, pp. 144–151, Chicago, IL, September 20-23 1993. IEEE Computer Society Press.

[PP93b] A. Podgurski and L. Pierce. "Retrieving Reusable Software by Sampling Behaviour". *ACM Transactions on Software Engineering and Methodology*, **2**(3):286–303, July 1993.

[Pri87] R. Prieto-Díaz. "Classifying Software for Reusability". *IEEE Software*, **4**(1):6–16, January 1987.

[Pri91] R. Prieto-Díaz. "Implementing Faceted Classification for Software Reuse". *Communications of the ACM*, **34**(5):89–97, May 1991.

[Rit89] M. Rittri. "Using types as search keys in function libraries". In FPCA-4 [FPC89], pp. 174–183.

[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". *Journal of 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.

[RM93] E. Reiter and C. Mellish. "Using Classification as a Programming Language". In *IJCAI-93*, 1993.

[Roa97] S. M. Roach. *TOPS: Theory Operationalization for Program Synthesis*. PhD thesis, University of Wyoming, August 1997.

[ROS98] J. Rushby, S. Owre, and N. Shankar. "Subtypes for Specifications: Predicate Subtyping in PVS". *IEEE Transactions on Software Engineering*, **24**(9):709–720, September 1998.

[RS98] W. Reif and G. Schellhorn. "Theorem Proving in Large Theories". In Bibel and Schmitt [BS98].

[RT89] C. Runciman and I. Toyn. "Retrieving re-usable software components by polymorphic type". In FPCA-4 [FPC89], pp. 166–173.

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

[RVL97] S. M. Roach, J. Van Baalen, and M. Lowry. "Meta-Amphion: Scaling up High Assurance Deductive Program Synthesis". In *Proceedings of the IEEE High Integrity Software Symposium*, Albuquerque, New Mexico, October 1997.

[RW88] C. Rich and R. C. Waters. "The Programmers Apprentice: A Research Overview". *IEEE Computer*, **21**(11):11–25, November 1988.

[RW90a] C. Rich and L. M. Wills. "Recognizing a Programs's Description: A Graph-Parsing Approach". *IEEE Software*, **7**(1):82–89, 1990.

[RW90b] E. J. Rollins and J. M. Wing. Specifications as Search Keys for Software Libraries. Technical Report CMU-CS-90-159, Department of Computer Science, Carnegie-Mellon University, October 1990.

[RW91] E. J. Rollins and J. M. Wing. "Specifications as Search Keys for Software Libraries". In K. Furukawa, (ed.), *Proceedings of the 8th International Conference and Symposium on Logic Programming*, pp. 173–187, Paris, June 24-28 1991. MIT Press.

[SA91] R. Socher-Ambrosius. "Boolean Algebra Admits No Convergent Term Rewriting System". In R. V. Book, (ed.), *Proceedings of the 4th International Conference on Rewriting Techniques and Applications*, *Lecture Notes in Computer Science* **488**, pp. 264–274, Como, Italy, April 1991. Springer.

[Sai99] H. Saidi. "Modular and Incremental Analysis of Concurrent Software Systems". In Hall and Tyugu [HT99], pp. 92–101.

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

[Sch98] J. Schumann, 1998. Personal communication.

[Sch00] J. M. P. Schumann. *Automated Theorem Proving in Software Engineering.* Habilitation, Technische Universität München, Institut für Informatik, 2000.

[Set95] D. Setliff, (ed.). *Proceedings of the 10th Knowledge-Based Software Engineering Conference*, Boston, MA, November 12-15 1995. IEEE Computer Society Press.

[SF97] J. M. P. Schumann and B. Fischer. "NORA/HAMMR: Making Deduction-Based Software Component Retrieval Practical". In Lowry and Ledru [LL97], pp. 246–254.

[SG95] M. Shaw and D. Garlan. "Formulations and Formalisms in Software Architecture". In J. van Leeuwen, (ed.), *Computer Science Today*, *Lecture Notes in Computer Science* **1000**, pp. 307–323. Springer, 1995.

[Sho78] R. E. Shostak. "An Algorithm for Reasoning About Equality". *Communications of the ACM*, **21**(7):583–585, July 1978.

[Sho79] R. E. Shostak. "A Practical Decision Procedure for Arithmetic with Function Symbols". *Journal of the ACM*, **26**(2):351–360, April 1979.

[SL90] J. M. P. Schumann and R. Letz. "PARTHEO: A High-Performance Parallel Theorem Prover". In Stickel [Sti90], pp. 40–56.

[SL91] R. A. Steigerwald and Luqi. "CASE tool for reusable software component storage and retrieval". *Information and Software Technology*, **33**:698–706, November 1991.

[SL99] M. Sitaraman and G. T. Leavens, (eds.). *Foundations of Component-Based Systems*. Cambridge University Press, Cambridge, 1999.

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

[Spi92] J. M. Spivey. *The Z Notation: A Reference Manual*. International Series in Computer Science. Prentice-Hall, 2nd edition, 1992.

[SS97] G. Sutcliffe and C. B. Suttner. "The CADE-13 ATP System Competition". *Journal of Automated Reasoning*, **18**(2):137–138, April 1997.

[SS98] G. Sutcliffe and C. B. Suttner. The CADE-14 ATP System Competition. Technical Report JCU-CS-98/01, Department of Computer Science, James Cook University, March 1998.

[SSY94] G. Sutcliffe, C. B. Suttner, and T. Yemenis. "The TPTP Problem Library". In Bundy [Bun94], pp. 252–266.

[Ste91] R. A. Steigerwald. *Reusable Software Component Retrieval via Normalized Algebraic Specifications*. PhD thesis, Naval Postgraduate School, December 1991.

[Sti88] M. E. Stickel. "A Prolog Technology Theorem Prover: implementation by an extended Prolog compiler". *Journal of Automated Reasoning*, **4**:353–380, 1988.

[Sti90] M. E. Stickel, (ed.). *Proceedings of the 10th International Conference on Automated Deduction*, *Lecture Notes in Computer Science* **449**, Kaiserslautern, July 1990. Springer.

[Sti92] M. E. Stickel. "A Prolog technology theorem prover: a new exposition and implementation in Prolog". *Theoretical Computer Science*, **104**(1):109–128, October 1992.

[Sut98] G. Sutcliffe, 1998. Personal communication.

[SW89] P. H. Schmidt and W. Wernecke. "Tableau Calculus for Order Sorted Logic". In Bläsius et al. [BHR89], pp. 49–60.

[Tam97a] T. Tammet. "Gandalf". *Journal of Automated Reasoning*, **18**(2):199–204, April 1997.

[Tam97b] T. Tammet. Reference Manual, GANDALF c-1.0c, October 1997.

[Tra88a] W. Tracz. "RMISE Workshop on Software Reuse Meeting Summary". In *Software Reuse: Emerging Technology* [Tra88b], pp. 41–53.

[Tra88b] W. Tracz, (ed.). *Software Reuse: Emerging Technology*, Washington, D.C., 1988. IEEE, IEEE Computer Society Press.

[VK85] D. M. Volpano and R. B. Kieburtz. "Software Templates". In *Proceedings of the 8th International Conference on Software Engineering*, pp. 55–60, London, August 1985. IEEE Computer Society Press.

[VPP00] W. Visser, S. Park, and J. Penix. "Using Predicate Abstraction to Reduce Object-Oriented Programs for Model Checking". In *Proceedings of the 3rd ACM SIGSOFT Workshop on Formal Methods in Software Practice*, August 2000. To appear.

[vR79] C. J. van Rijsbergen. *Information Retrieval*. Butterworth, London, 1979.

[VR98] J. Van Baalen and S. M. Roach. "Using Decision Procedures to Build Domain-Specific Deductive Synthesis Systems". In *Proceedings of the Eighth International Workshop on Logic Program Synthesis and Transformation (LOPSTR '98)*, Manchester, UK, June 1998.

[Wal69] R. J. Waldinger. "PROW: a step towards automatic program writing". In Walker and Norton [WN69], pp. 241–252.

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

[Wei96] C. Weidenbach. *Compuational Aspects of a First-Order Logic with Sorts*. PhD thesis, Universität des Saarlandes, December 1996.

[Wei97] C. Weidenbach. "SPASS—Version 0.49". *Journal of Automated Reasoning*, **18**(2):247–252, April 1997.

[Wei98] H.-K. Weiler. Axiomatisierung und Lemmaauswahl für Beweisbedingungen. Master's thesis, Technical University Braunschweig, 1998.

[WGR96] C. Weidenbach, B. Gaede, and G. Rock. "Spass and Flotter version 0.42". In M. A. McRobbie and J. K. Slaney, (eds.), *Proceedings of the 13th International Conference on Automated Deduction*, *Lecture Notes in Artifical Intelligence* **1104**, pp. 141–145, New Brunswick, NJ, July-August 1996. Springer.

[WN69] D. E. Walker and L. M. Norton, (eds.). *Proceedings of the 1st International Joint Conference on Artificial Intelligence*, Washington, DC, May 1969. William Kaufmann.