Johann M. Schumann

Automated Theorem Proving in Software Engineering

Springer

Automated Theorem Proving in Software Engineering

Springer-Verlag Berlin Heidelberg GmbH

Johann M. Schumann

Automated Theorem Proving in Software Engineering

Foreword by Donald Loveland

Springer

Dr. Johann M. Schumann

Automated Software Engineering
RIACS / NASA Ames Research Center
Moffett Field, CA 94035
USA

schumann@ptolemy.arc.nasa.gov

With 46 Figures and 41 Tables

Library of Congress Cataloging-in-Publication Data

Schumann, Johann M., 1960–
Automated theorem proving in software engineering/Johann M. Schumann.
p. cm.
Includes bibliographical references and index.
ISBN 978-3-642-08759-2 ISBN 978-3-662-22646-9 (eBook)
DOI 10.1007/978-3-662-22646-9
1. Software engineering. 2. Automatic theorem proving. I. Title.

QA76.758.S38 2001
005.1–dc21 2001020538

ACM Computing Classification (1998):
I.2.2-3, D.2.4, D.1.2, F.3.1, F.4.1, C.2.2, D.4.6, D.2.13

ISBN 978-3-642-08759-2

Originally published by Springer-Verlag Berlin Heidelberg New York in 2001
Softcover reprint of the hardcover 1st edition 2001

Cover Design: KünkelLopka, Heidelberg
Typesetting by the author using a Springer TEX macro-package
Printed on acid-free paper SPIN 10780856 45/3142SR – 5 4 3 2 1 0

Foreword

This book can mark the coming of age of automated theorem proving (ATP). The process to maturity has been a continuum, as it is for humans, but this book serves to mark the emergence of ATP into the marketplace. For this book is arguably the first to present for the general computer scientist or mathematician in some technical depth the ability of automated theorem provers to function in the realm where they will earn their living. That realm is as the reasoning engines of verifiers and generators of computer programs, hardware and related products. (We do note some excellent edited collections exist; one of the best is by Bibel and Schmitt, 1998: see this book's bibliography.) As we note below, this book does not simply document a brilliant but isolated undertaking. Rather, the book makes clear that a small but steady, and increasing, stream of real-world applications is now appearing.

The childhood and adolescence of ATP was both prolonged and spiked with brilliance. The birth year of the field should probably be set as 1956, when the Logic Theorist paper was published by Newell, Shaw and Simon. (However, most likely the first computer generated mathematical proof appeared in 1954 as output of a program for Pressburger arithmetic, written by Martin Davis. The work was not published at the time.) Thus the ATP field is at least 45 years old, well time for passage into the workplace of the world. Some superb accomplishments marked the period of adolescence, the Robbins algebra conjecture proof discovered by McCune's OTTER program and the sophisticated geometry provers of Chou et al. being perhaps most notable. (The discovery of a proof for the Robbins algebra conjecture closed a famous open problem.) But, despite the above and other accomplishments, it will never be possible to live up to the visions of the 1950s for the infant discipline of ATP. The first visions of the artificial intelligence (AI) world had ATP at the core of automated intelligence, for it seemed natural to assert that human reasoning could not be simulated without a central deductive engine of substantial power. Though many had moved away from this envisioned architecture by the late 1960s, the thesis got a shot in the arm from the work of Cordell Green who showed how proof could be converted to computation, and devised robot plans using a theorem prover. By 1970, however, the explosive cost of search through proof space was apparent to all users

of automated provers. This lead to a clouded, if not dark, adolescence from which ATP is just now emerging.

Ever since the end of the early illusions of ATP and AI, the focus of opportunity has been the use of ATP in program and hardware verification. Much time and funds were invested in the 1970s and early 1980s in the attempt to bring general verifiers into practical use. This failed, although multiple developmental efforts continue with increasing success. Many people feel the future lies with semi-automated provers; that is, computer systems where the human and machine constantly interact. But in recent years a number of ATP groups have developed applications admittedly modest but with real-world workplace importance.

In fairness we must note that earlier points in time could have been selected as the "coming of age". The important AI programming language Prolog incorporates an automated theorem prover. Bob Boyer and J. Moore published a book in 1988 that documented their interactive theorem prover, which has been used to verify significant parts of commercial chips. But these and a few other accomplishments have been isolated accomplishments. The important difference is that now there is a flow of applications, however thin, that clearly will increase.

Both the ATP and the software engineering fields are fortunate to have a book such as prepared by Johann Schumann to document this coming of age. The ATP technology is presented in depth but with the software engineer as target. Careful analysis is given of the methodologies of ATP and related disciplines and what features are applicable to formal methods in software engineering. The case studies give excellent detail of current applications. Johann spent several years in industry prior to writing this treatise, and presently is engaged in just the type of software applications which he treats here. No one is better qualified to document this coming of age of ATP, and coming of a new age in software engineering.

It is a pleasure to recommend this unique book to all interested in this corner of the future.

March 2001 *Donald Loveland*

Preface

The growing demand for quality, safety, and security of software systems can only be met by rigorous application of formal methods during software design. Tools for formal methods, however, in general do not have a sufficient level of automatic processing. This book investigates the potential of first-order logic automated theorem provers (ATPs) for applications in the area of software engineering. It tries to close the gap between the world of proof obligations arising from the application of formal methods in software engineering and that of automated theorem proving. The following central questions are investigated:

1. *Which requirements must an ATP fulfill in order to be applied successfully?*
   Application of an automated theorem prover in the area of software engineering means to integrate the ATP into the application system (usually a formal methods tool) or to use it as a stand-alone system. In both cases of practical application requirements for the ATP show up which are quite different from those needed to handle mathematical and logical problems. These requirements are studied in detail in this book and are clarified by setting up a systematic characterization of proof tasks.
2. *Which kinds of proof tasks are suited for automatic processing?*
   In this book we identify promising application areas. The book contributes to the field by a detailed description of design and experimentation of several successful case studies: verification of communication protocols, formal analysis and verification of authentication protocols, and logic based software reuse. These case studies range from a feasibility study to an almost-industrial prototypical tool with an experimental data base of more than 14,000 proof tasks.
3. *What techniques can we implement and use together with state-of-the-art automated theorem provers to meet these requirements for practical applicability?*
   In this book we provide an in-depth analysis of central techniques and methods for the adaptation of the automated prover. Performance and importance is illustrated by examples and experiments from our case studies. The main contributions of the author to the field concerns the development and adaptation of techniques (like handling of inductive

problems, preselection of axioms, handling of sorts and non-theorems, control of the prover) and their integration into a generic system architecture. Considerable progress has been made with respect to the control of the automated system for maximum performance by exploiting parallelism and combining search paradigms.

All topics mentioned above are of major importance. Many possible applications of techniques of automated deduction had not been tried or had failed, because the potential users (i.e., developers of tools for formal methods) had too little knowledge about the (often necessary) details of automated provers (e.g., their many, often poorly documented parameters). On the other hand, the community of ATP developers had (and still often has) no consciousness with respect to applicability of the systems and algorithms they develop. Rather, there is a strong focus on theory and applications in mathematics. The capacity of systems are in most cases only tested and assessed against few toy examples or benchmark data bases. Most provers per se lack major features for practical applicability. Therefore, the book proceeds along the following lines:

1. We explore the rising demands on software engineering and conclude that within many phases of the software life-cycle tasks arise which require the use of *formal methods* (Chapter 2).
2. A significant portion of the *actions* carried out with formal methods require solving of proof tasks. We describe the structure and capabilities of inference systems with a focus on *automated theorem proving* (Chapter 3).
3. Knowing what an inference system is and what it can do, we *characterize* the proof tasks arising from applications and illustrate this characterization by examining some existing tools and applications (Chapter 4).
4. Then, we will have a close look at the problems which arise when a bare-bone theorem prover is tried to be applied to a given application. From there, we will develop *requirements* which an ATP must fulfill for a successful application (Chapter 5).
5. We will demonstrate our techniques and methods with successful *case studies* from important application areas which were performed by the author. The case studies cover important areas of verification of communication protocols, the analysis of security protocols, and logic-based component retrieval, a well-known technique of software reuse (Chapter 6).
6. For the central requirements in Chapter 5 and the applications of Chapter 6, we have developed and adapted techniques for ATPs and give now a concerted and detailed description of these techniques (Chapter 7).
7. Finally, we will summarize our approach and raise some important open questions in ATP applications to software engineering (Chapter 8).
8. An extensive bibliography (more than 300 entries) contains pointers to relevant work and related topics.

This book is intended for readers who wish or have to apply techniques of automated deduction for specific tasks in the area of Software Engineering. Therefore, it addresses readers from the field of automated deduction and theorem proving, as well as readers from the area of formal methods. Because the book is designed to be self-contained and contains a number of extended case studies, it is also of interest for a computer scientist in general who wants to get an overview of what can be done with today's automated theorem provers and which problems have to be overcome.

Acknowledgements

This book is a revised version of my habilitation thesis presented at the Technische Universität München. Many people have contributed to this book. First of all, I thank my advisor, Prof. Eike Jessen, for his constant support, valuable feedback and encouragement. The other reviewers, Prof. Manfred Broy and Prof. Donald Loveland, provided important advice and feedback. Most of this work has been done at the research group "Automated Reasoning" at the Technische Universität München. I would like to thank all my colleagues and students for an inspiring and open working atmosphere. Part of this work was done during a research stay at ICSI, Berkeley, U.S.A. Furthermore, I would like to thank Bernd Fischer, Claudia Eckert, and Ingo Dahn for their cooperation and help. I thank Dr. Hans Wössner from Springer-Verlag for his careful and thorough proof reading and for many improvements of this book. Finally, Marleen Schnetzer helped substantially with typesetting and correcting the text.

The work described here has been supported by the Deutsche Forschungsgemeinschaft (DFG) within the Habilitation grant Schu908/5-1,2, the Sonderforschungsbereich SFB342/A5 at the Technische Universität München, and the "Schwerpunkt Deduktion". Final revisions of the text have been made at NASA/Ames, CA, and were supported by NASA Aero IT Base Program Synthesis, program code number 704–50.

Mountain View, Spring 2001 *Johann Schumann*

Contents

1. Introduction

The amount and complexity of software developed and used has grown tremendously during the past years. The number of processors on which that software is supposed to run has by far outnumbered human beings [Storey, 1996]. Most of these processors are in fact not "classical" computers like mainframes, workstations, or PCs. Rather they are components of embedded systems which control applications and machinery from car brakes and washing machines to central components of nuclear plants[1]. The software which is running on the processors has reached many billions of lines of code. Therefore, the development and maintenance of software poses rising demands on software engineering technology. Software and hardware are expected to work *error-free*, *safe*, and *reliable*. Whereas the crash of a word processor or your favorite game may be a nuisance, a malfunction in software controlling a nuclear plant can possibly cost thousands of human lives and can endanger millions more. Many applications exist where a high reliability must be ensured, because failures are costly (with respect to human lives, environmental issues, or money). Such applications can be found in nearly all areas, e.g., aviation, (nuclear) power plants, medicine, transportation, space technologies, process control, or banking. Reliability must not only be guaranteed with respect to possible bugs and errors in the software. The safety of a computer system against malevolent attacks (from intruders like hackers or criminal persons) is also of growing importance. Particularly endangered are the extremely fast growing areas of electronic commerce, (tele-) banking, or remote access to computer systems (e.g., remote login). A good collection of hazardous incidents with computer systems is presented in P. Neumann's book [Neumann, 1995] where also an assessment of computer related risks is given.

Another important aspect of software engineering concerns *maintainability* of code and data. In general, software systems are "living" much longer than originally intended; usually *much* longer than any single hardware generation. Maintaining these huge amounts of code, often poorly documented, is extremely costly and resource-consuming. This problem became evident when systems had to checked for Year 2000 compliance with a cost of hun-

[1] For example, the shutdown system of the Darlington nuclear generating station (Ontario, Canada) is controlled by computers ([Storey, 1996], Chapter 15.4).

dreds of billion U.S.$. Typical activities required in this area are program understanding, program validation, and testing.

Another core topic in the area of dealing with existing code is *software reuse*. Here, attempts are being made to make parts of the software easily available for further program developments. However, activities in software reuse are not risk-less in their application. A prominent example of sloppy software reuse is the crash of the new Ariane V carrier-rocket in 1996. According to [Lions et al., 1996], an unmodified software module from Ariane IV caused an uncaught exception during the launch which in turn led to malfunction of the entire control program and destruction of the rocket.

1.1 Formal Methods

Numerous approaches and methods to increase safety, reliability, and maintainability of computer systems and software have been developed. One promising approach is to use *formal methods* during *all* phases of the software life cycle. Formal methods employ techniques from mathematical logic and discrete mathematics for the description, specification, and construction of programs. Commonly based on mathematical logic, formal methods allow to write down the behavior of the system under consideration (in a very broad sense) and to express properties about the system. This requires to explicitly address all assumptions and allows to draw all conclusions within the logic framework. Reasoning is done with a small number of formally defined inference rules.

However, there are also a number of reasons against the application of formal methods, as numerous authors have pointed out (e.g., [Stevenson, 1990; Weber-Wulff, 1993; Knight *et al.*, 1997]). Most formal methods are considered to be too clumsy and time-consuming in their application. They usually require a long learning curve for the software engineer and are often not compatible with the "as-we-always-did" software development procedures. Together with the — in general very tight — project schedules, an application of formal methods in an industrial environment is often not accepted by the project team. Furthermore, formal methods have been developed in academia with usually complex, but small "toy"-examples. The question of scaling-up to real-world problems is still open. Industrial-size applications of formal methods require powerful, user-friendly and effective tools which are not yet available. The improvement of such tools (with respect to their degree of automatization) will be a core topic of this book.

The first approaches to formal methods proposed in classical papers (e.g., [Hoare, 1983]) initially led to enthusiasm which, however, was — due to the reasons above — quickly disappointed. In the years thereafter, formal methods in software engineering led a more hidden life, a situation as depicted in Figure 1.1. Now, formal methods are being used more and more. Growing

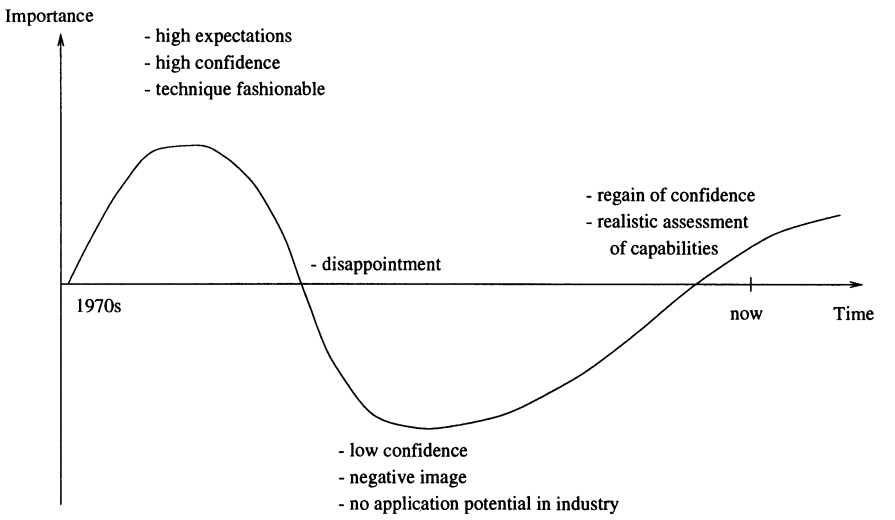

Fig. 1.1. Level of interest (and expectations) for formal methods (and/or theorem proving) in industry (or industrial research) over the time

research activities are reflected in a series of major conferences, e.g., Formal Methods (FME, e.g., [Woodcock and Larsen, 1993; Naftalin *et al.*, 1994; Gaudel and Woodcock, 1996; Fitzgerald *et al.*, 1997], Computer Aided Verification (CAV, e.g., [Dill, 1994; Alur and Henzinger, 1996; Hu and Vardi, 1998], Automated Software Engineering (ASE, e.g., [Welty *et al.*, 1997]), Knowledge-based Software Engineering (KBSE, e.g., [Selfridge *et al.*, 1991; KBSE-10, 1995; KBSE-11, 1996]) and many books. R. B. Jones's archive [URL Formal Methods Archive, 2000] lists a huge amount of research papers, case studies and tools for formal methods. The impact of formal methods in industry and industrial applications has been studied in various surveys [Craigen, 1999; Gerhart *et al.*, 1994; Craigen *et al.*, 1993; Abrial *et al.*, 1996; Weber-Wulff, 1993].

1.2 Formal Methods Tools

As mentioned above, one major reason for not using formal methods more widely is the lack of user-friendly, yet powerful tools. Such tools should not only support all relevant phases of a software project without obstructing the default engineering procedures (e.g., the development/coding/testing cycles used at Microsoft [Cusumano and Selby, 1997]). Rather should these tools increase quality and reliability of the software by simultaneously raising productivity. All persons involved in a software project should be able to work with these tools in a problem-oriented, user-friendly way on all required oper-

ations. Such operations are typically developing and entering (requirements-) specifications, debugging them, checking consistency, refinement, verification and validation, simulation, testing and documentation.

Due to the high complexity of a formal method alone, a usable tool must not impose any avoidable burden upon the user by requiring detailed knowledge about underlying semantics, details of logic and inference components. Rather, any evidence of internal formal operations (like theorem proving) should be hidden from the user and should be performed with a high degree of automatic processing.

Whereas simple tasks like syntax checking, static type checking, etc., are already default for most today's tools, more complex tasks are still mostly unsupported. In particular, the ability to *prove* certain properties (e.g., invariances, deadlock freedom, termination) is central to any operation regarding formal methods. Such proof obligations (in the following, we will call them *proof tasks*) can be of very different structure and complexity. Examples for such proof tasks are to prove liveness and safety properties or to show that the formal parameters of a procedure match with the actual given ones. Also, for each refinement of a specification it is necessary to prove that the important properties of the specification are not violated.

Example 1.2.1. Let us consider the specification of the function `max3` which returns the maximum of three integer values. The specification is written using simple pre- and postconditions in a VDM/SL-like syntax [Dawes, 1991]. An implementation of this function using pseudo-code is shown in Figure 1.2.

SPEC: $\max_3(i_1 : \mathsf{int}, i_2 : \mathsf{int}, i_3 : \mathsf{int}) \longrightarrow \mathsf{int}$
PRE: $-$
POST: $\forall \max_3, i_1, i_2, i_3 : \mathsf{int} \cdot (\max_3 = i_1 \vee \max_3 = i_2 \vee \max_3 = i_3) \wedge$
$\max_3 \geq i_1 \wedge \max_3 \geq i_2 \wedge \max_3 \geq i_3$

```
int max3(i1:int,i2:int,i3:int)     // implementation of "max3"
if (i1 > i2) then
  if (i1 > i3) then
    return i1;                     // i1 > i2, i1 > i3
  else
    return i3;                     // i1 > i2, i3 >= i1
  fi
else
  if (i2 > i3) then
    return i2;                     // i2 >= i1, i2 > i3
  else
    return i3;                     // i2 >= i1, i3 >= i2
  fi
fi
```

Fig. 1.2. Implementation of function `max3`

In order to *verify* that this implementation satisfies the specification, we have to *prove* that

$$\forall i_1, i_2, i_3 : \mathsf{int} \cdot \mathbf{max3}(i_1, i_2, i_3) = \max_3(i_1, i_2, i_3)$$

A proof for this obligation requires to look at all 6 different cases of orderings for i_1, i_2, i_3. If we succeed to show all cases our proof task has been solved. As an example let us consider the case $i_1 \leq i_2, i_1 \leq i_3, i_2 \leq i_3$. The conjecture for this case is (1.1) and a formal (and very detailed) proof for it is shown in Figure 1.3.

$$\forall i_1, i_2, i_3, \max_3, \mathbf{max3} : \mathsf{int} \cdot i_1 \leq i_2 \wedge i_1 \leq i_3 \wedge i_2 \leq i_3 \wedge \text{POST} \wedge \text{IMPL} \Rightarrow \mathbf{max3} = i_3 \tag{1.1}$$

$\forall i_1, i_2, i_3, \max_3, \mathbf{max3} : \mathsf{int} \cdot i_1 \leq i_2 \wedge i_1 \leq i_3 \wedge i_2 \leq i_3 \wedge \text{POST} \wedge \text{IMPL} \Rightarrow \mathbf{max3} = i_3$
[expand definition of ''post'']
$\forall i_1, i_2, i_3, \max_3, \mathbf{max3} : \mathsf{int} \cdot i_1 \leq i_2 \wedge i_1 \leq i_3 \wedge i_2 \leq i_3 \wedge ((\max_3 = i_1 \vee \max_3 = i_2 \vee \max_3 = i_3) \wedge \max_3 \geq i_1 \wedge \max_3 \geq i_2 \wedge \max_3 \geq i_3) \wedge \text{IMPL} \Rightarrow \mathbf{max3} = i_3$
[set value $\max_3 = i_3$ and via simplification]
$\forall i_1, i_2, i_3, \max_3, \mathbf{max3} : \mathsf{int} \cdot i_1 \leq i_2 \wedge i_1 \leq i_3 \wedge i_2 \leq i_3 \wedge \max_3 = i_3 \wedge \text{IMPL} \Rightarrow \mathbf{max3} = i_3$
[expand definition of ''Impl'']
$\forall i_1, i_2, i_3, \max_3, \mathbf{max3} : \mathsf{int} \cdot i_1 \leq i_2 \wedge i_1 \leq i_3 \wedge i_2 \leq i_3 \wedge \max_3 = i_3 \wedge \mathbf{max3} = (\text{ if } i_1 > i_2 \text{ then if } i_1 > i_3 \text{ then } i_1 \text{ else } i_3 \text{ fi else if } i_2 > i_3 \text{ then } i_2 \text{ else } i_3 \text{ fi fi }) \Rightarrow \mathbf{max3} = i_3$
[via evaluation]
$\forall i_1, i_2, i_3, \max_3, \mathbf{max3} : \mathsf{int} \cdot i_1 \leq i_2 \wedge i_1 \leq i_3 \wedge i_2 \leq i_3 \wedge \max_3 = i_3 \wedge \mathbf{max3} = (\text{ if FALSE then if FALSE then } i_1 \text{ else } i_3 \text{ fi else if FALSE then } i_2 \text{ else } i_3 \text{ fi fi }) \Rightarrow \mathbf{max3} = i_3$
[via simplification]
$\forall i_1, i_2, i_3, \max_3, \mathbf{max3} : \mathsf{int} \cdot i_1 \leq i_2 \wedge i_1 \leq i_3 \wedge i_2 \leq i_3 \wedge \max_3 = i_3 \wedge \mathbf{max3} = i_3 \Rightarrow \mathbf{max3} = i_3$
[via simplification]
[QED]

Fig. 1.3. Detailed proof of example conjecture (1.1)

The previous example shows that some proof tasks (but by far not all) are rather trivial. Nevertheless, they must be solved on the full level of detail. This then easily becomes tedious and error-prone work, in particular if many proof tasks must be processed.

1.3 Inference Systems

For wide-spread tool acceptance, *automatic inference systems* should be applied to automatically process as many proof tasks as possible. Such systems

— although only a small "kernel" of a formal methods tool — are extremely important for the tool's power and usability. Here, we refer to an inference system as a program or computer-assisted tool which is able to perform logical operations in the framework of the formal method(s) under consideration. Inference systems can, for example, be decision procedures, symbolic manipulation systems (e.g., rewriting systems, symbolic algebra systems), model checkers, and interactive or automatic theorem provers.

In this book, we will focus on the question, how automated theorem provers (ATPs) (in particular those for first-order predicate logic) can be successfully applied to process proof obligations arising from software engineering applications, and which requirements they must fulfill. Obviously, ATPs must be *adapted* in order to be *combined* with other inference components and application systems. The importance of this question will become clear, when we have looked at the major characteristics of model checkers, interactive theorem provers and automated theorem provers, and their advantages and disadvantages.

1.3.1 Model Checkers

Model Checkers are decision procedures for temporal propositional logic. They are particularly suited to show properties of finite state-transition systems. Due to powerful implementations using binary decision diagrams (BDDs) or propositional satisfiability procedures, model checkers are now quite often used in industry (mostly in hardware-oriented areas or relatively low-level protocol and control applications). Prominent systems are, e.g., SMV [McMillan, 1993], SPIN [Holzmann, 1991], Step, murphi [Dill, 1996], Java Pathfinder[Visser *et al.*, 2000], or μcke. Logics, specifically suited for the description of finite automata and their properties (e.g., CTL [Burch *et al.*, 1990]) and convenient input languages facilitate the use of model checkers. Their ability to generate *counterexamples* when a conjecture cannot be proven provides valuable feedback for the user.

Recently, model checkers have been extended (see, e.g. [Burkart, 1997] or the system Mona [Klarlund and Møller, 1998]) to handle infinite domains which have a finite model property. Nevertheless, model checkers usually cannot be applied in applications where recursive functions and data structures are used. Furthermore, most systems are not able to produce a formal proof (as a sequence of inference steps) which can be checked (or proofread) externally (but see, e.g., the Concurrency Workbench [Moller, 1992; Cleaveland *et al.*, 1993]). Rather, the user has to rely on the correctness of the implementation[2]. The most severe reduction for the practical applicability of model checkers is the limit of the size of the state space they can handle.

[2] This was also a point of critique when model generators (Finder [Slaney, 1994] and MACE [McCune, 1994b]) were used to prove prominent conjectures in the area of finite quasi-groups.

Despite numerous approaches (e.g., [Burch *et al.*, 1992]), proof tasks must be broken down or abstracted carefully in order to avoid state space explosion. For a survey on current research in model checking see [Clarke *et al.*, 2000].

1.3.2 Interactive Theorem Provers

Interactive theorem provers (ITPs) are systems which allow to process logical formulas and to apply inference rules upon them. Application of operations and inference rules is performed manually by executing commands and creating command scripts (often also called tactics). Then, the proof is constructed step by step with manual interactions and predefined actions or action schemas. Most prominent interactive theorem provers are NuPrl [Constable *et al.*, 1986], PVS [Crow *et al.*, 1995], Isabelle [Paulson, 1994], HOL [Gordon, 1988], or Larch [Guttag *et al.*, 1993], just to name a few.

The major advantage of using an interactive theorem prover is its high expressive power and flexibility. It is possible to construct theorem provers for "customized logics", higher-order logics, and typed logics. With this kind of provers, the proof tasks can be written down in a relatively natural, problem-oriented way. Complex proof operations, e.g., induction, usually can be expressed within the framework of the underlying logic. Thus, really complex proofs can be constructed with an interactive theorem prover. System features often include mechanisms for an automatic "replay" of a proof which already has been discovered and tools for proof and formula management (e.g., proved theorems can be added as lemmata to a data base). Furthermore, ITPs can be customized to specific application domains by providing libraries of tactics, thus increasing the amount of automatic processing. For several areas in software engineering, interactive theorem provers (e.g., PVS, Isabelle) and verification systems with interactive theorem provers (e.g., KIDS [Smith, 1990], KIV [Reif *et al.*, 1997]) are being used successfully in verification of compilers, protocols, hardware, algorithms, in refinement, and in program transformation. On the other hand, interactive theorem provers have a number of serious disadvantages: interactive theorem provers are often *too* interactive. Even the smallest steps in the proof (e.g., to set a variable to 0 in the base case of an induction proof) must be carried out by hand. Large and complicated proofs take weeks to months to be completed. E.g., [Reif and Schellhorn, 1998] report proof times of several months for carrying out single proofs for the refinement of a specification for the Warren Abstract Machine[3]. Similar proof times could be observed by [Havelund and Shankar, 1996]. Furthermore, most interactive theorem provers can be used only by persons who have much knowledge and experience in the application domain, the calculus, the tactics language of the prover, and sometimes even in the implementation language of the prover (e.g., ML for Isabelle). For example,

[3] Although the topic of this case study might seem a little academic, its complexity and size resembles a typical demanding industrial application.

many proofs in [Paulson, 1997b] "[...] require deep knowledge of Isabelle". Therefore, the use of interactive theorem provers requires specifically trained and experienced persons.

1.3.3 Automated Theorem Provers

Automated Theorem Provers are programs which, given a formula, try to evaluate if the formula is universally valid or not. This search for a proof is carried out fully automatically. Efficient automated theorem provers have been mostly developed for first-order predicate logic or subsets thereof. The reason for this is that for this logic, effective inference rules and proof procedures exist.

Although automated theorem provers have gained tremendously in power during the last few years, automated theorem provers for first-order predicate logic have almost *not* been used at all in the area of software engineering. There are several reasons for this.

In the past, automated theorem provers were just *too weak* to be applied for real-world applications. Even small toy examples needed extremely long run-times, if a proof could be found at all. More complex examples as they occur in most applications have been well beyond the capacity of those systems. Only quite recently, a number of extremely powerful automated theorem provers have been developed (e.g., OTTER [McCune, 1994a; Kalman, 2001], SPASS [Weidenbach *et al.*, 1996], Gandalf [Tammet, 1997], or SETHEO; see [Sutcliffe and Suttner, 1997; Suttner and Sutcliffe, 1998; Suttner and Sutcliffe, 1999] for an overview on existing systems).

A further reason for not using ATPs is the inherent *low expressiveness* of their input language (first-order predicate logic or subclasses). The natural representation of many application problems, however, often uses higher-order logic or requires higher-order concepts for the proof. One only has to think of the well-known induction principle, or of expressions like:

$$\forall p \in \{\mathsf{send}, \mathsf{receive}\} \quad \forall X : \mathsf{message} \cdot p(X)$$

Further reasons which prevent the application of ATPs are of a more technical nature. Currently, most ATPs are like racing cars: although very fast and powerful, they cannot be used for everyday traffic, because essential things (like headlights) are missing. The classical architecture of an ATP (i.e., a highly efficient and tuned search algorithm) will and must be extended into several directions in order to be useful for real applications. Although the extensions should be especially tailored to the needs of proof tasks arising in the domain of software engineering, many of these extensions which will be investigated in this book, are helpful for any kind of application.

In this book we will look at the characteristics of proof obligations in software engineering and the requirements they pose upon automated theorem provers. With case studies and detailed discussion of selected adaptation

techniques we will demonstrate, that ATPs are already well-suited for quite a number of applications.

Automated theorem provers, however, are not a universal cure for every application and every proof obligation — even if it could be processed by the prover. In many cases, user interactions will always be necessary to perform complex core proof operations ("central proof ideas"). However, the goal is to relieve the user from tedious low-level detail-work. Ideally, automated theorem provers can play a role in software engineering equivalent to a pocket calculator in classical engineering: simple tasks are done automatically but there is still work to be done for the engineer. On the other hand, ATP systems by their very nature always perform time-consuming search operations. Care should be taken to select problems that actually require considerable search. For example, in a logic-based system for automatic synthesis of efficient schedulers for logistic problems (PLANWARE, cf. [Burstein and Smith, 1996]), the designers had enough information about the operations required for the synthesis that they were able to "throw out the theorem prover" [Smith, 1998].

Thus in cases where efficient algorithmical ways of obtaining the solution are known, one should "calculate the proof" rather than "search for a proof". There, ATP systems will be inferior due to their overhead in processing a proof task.

Current automated theorem provers are mostly intended to find relatively complex structured proofs in very small formulae (consisting up to perhaps about 50 clauses). This is mostly due to historic reasons where most benchmark examples (e.g., the TPTP library [Sutcliffe *et al.*, 1994]) consist of problems formulated with the absolutely necessary axioms only. In practical applications, however, formulas are structured often quite differently: a typical formula consists of the theorem to be proven (mostly quite short) and a large set of axioms (e.g. taken out of a knowledge base). The proofs which are sought for mostly consist out of a few inference steps only and should be found easily. However, many automated provers are just overwhelmed by the pure size of the formula, and they take much too long for compiling and preprocessing.

Another problem with automated theorem provers is that in most cases they lack any user interface and any possibility for debugging a formula. Current implementations merely provide a sophisticated search algorithm, but expect a syntactically (and semantically) correct formula. They provide no feedback, if a proof could not be found.

Practical usability of an automated theorem prover requires that these and other important issues are addressed. These requirements are a central topic of this book which will be covered in detail after the introductory chapters on formal methods in software engineering and processing of logic with automated theorem provers.

2. Formal Methods in Software Engineering

2.1 Introduction

Formal methods in general refer to the use of techniques from logic and discrete mathematics to specification, design, construction, and analysis of computer systems and software [Kelly, 1997; Storey, 1996]. In this chapter, we will give a short overview of the role of formal methods in the area of software engineering. Formal methods are very important in order to avoid incomplete, inconsistent and ambiguous requirements, specifications or system descriptions. In contrast to documents written in natural language, formal methods are based on formal languages (e.g., mathematical logic) and require the explicit and concise notation of all assumptions.

Formal methods can be applied to all stages of the software life-cycle and on different degrees of formalization (Section 2.3). We are in particular interested in such levels which support the usage of specification languages (Section 2.4) and allow a formal analysis of the software system under development. In this formal analysis, reasoning is performed by a series of inference steps of the logic underlying the specification language (formal proof). The occurring *proof tasks* which will be characterized in detail in Chapter 4 are the basis for any application of automated theorem provers. Here, computer support and in particular, the application of *automatic* tools is of great importance, since only tools with a high degree of automatic processing facilitate the use of formal methods in industry. The current situation of formal methods will be shortly reviewed in Section 2.5.

This chapter is not intended to give a complete introduction into the huge field of formal methods, nor does it try to give a comprehensive overview of specification languages, tasks of verification, or current applications in industry. For greater depth, a variety of books on formal methods and on specification languages is available. A very comprehensive (and up to date) overview can be found in the Oxford formal methods database [URL Formal Methods Archive, 2000].

2.2 Formal Methods and the Software Life Cycle

The development of software is usually not done in one big step. All steps from first ideas to maintenance of the finished and shipped software product describes the software life cycle. There is a large number of different software processes on how software can be developed. A detailed discussion of various processes is outside the scope of this book. Therefore, we shortly discuss the two major models of a software life cycle, the *waterfall* model and the *spiral* or iterative life cycle model.

2.2.1 The Waterfall Model

The traditional model for the development of software, the *waterfall model* is depicted in Figure 2.1. (see, e.g., [Storey, 1996; Kelly, 1997] for more details and examples). Starting with the customer's wishes, a *requirements specification* of the proposed system is set up. From these requirements, a detailed *system specification*, elaborating precisely on the system's functionality is developed. This specification does not yet (or at least should not) include any details of the system's design or its implementation. These are worked out in the next phase, *software design* (architectural design) phase which is followed by the *implementation* phase.

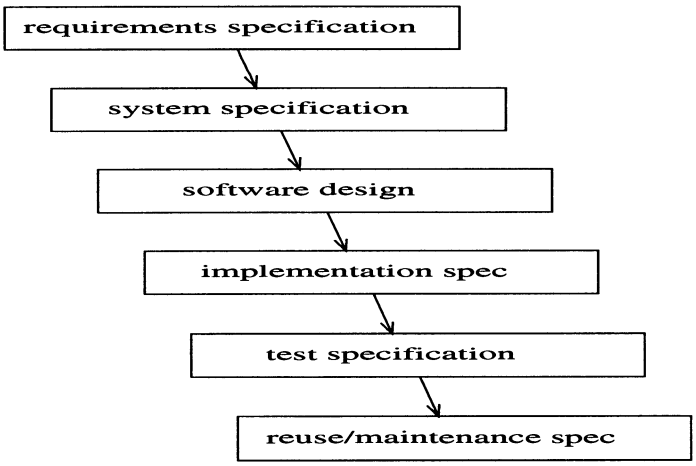

Fig. 2.1. Typical software life cycle (waterfall model)

Once implemented the software system must undergo a testing phase. Which tests are to be performed is described in a *test specification*. These tests must give reasonable assurance that the programs comply to all customer requirements. A final phase comprises activities which are performed after

shipping the software system: *maintenance* and *preparation for reuse*. Bugs which occur when the software is already in operation or required extensions of functionality must be handled within maintenance. If parts of the software are to be used again in other projects, preparations for software reuse (e.g., storage in libraries, documentation, access structures) have to be made.

2.2.2 An Iterative Life Cycle

The traditional waterfall model has, despite its wide usage, a number of severe drawbacks. Although theoretically, all steps of the software development process need to be performed only once, in practice this situation does not occur. Changes of customer requirements or errors detected in earlier stages of the software life cycle make it necessary to go back to the earlier stages and apply changes. This leads to the situation that software bugs are extremely more expensive to fix (up to approx. 100 times [Pressman, 1997]), the later they are detected during software development (Figure 2.2).

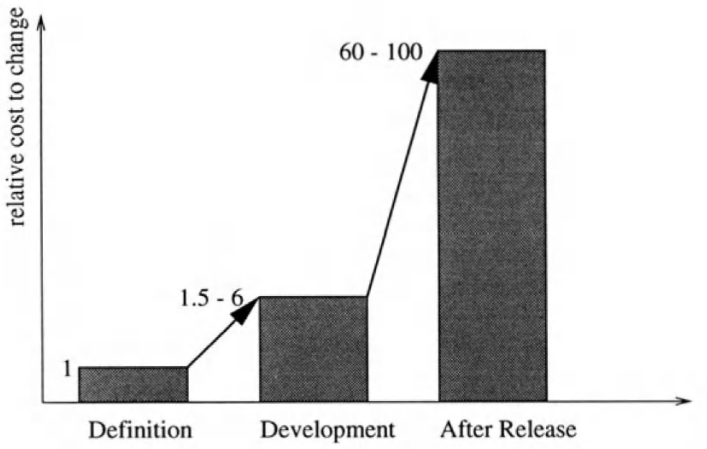

Fig. 2.2. Relative costs of a bug with respect to the time of its detection

Therefore, many modern methods prefer a software life cycle which introduces small and fast loops into the software life cycle. The result is a *spiral* or *iterative* model. Figure 2.3 shows a sketch of such a life cycle. Here again, we have the major development steps as before. By analysis of the real-world process, we obtain the requirements specifications. These are refined and elaborated into a design. During this process, however, the requirements and the design are regularly checked against the process and the requirements, respectively. In most cases, simulation (as depicted in the left triangle) will be used to accomplish this task. Going over several iterations, requirements and

design gain maturity. The design is the basis of the software system which is constructed by implementation. In order to ensure consistency with respect to design and requirements, validation through testing is done. During each iteration, production-quality software is produced. Finally, the piece of software needs to be tuned and verified with respect to the real-world process. By introducing small iterative loops into the process it is hoped that most bugs and change requests are detected in early stages, thus reducing development costs. Such iterative life cycles are typically used in object-oriented software development using the Unified Modeling Language UML [Fowler, 1997].

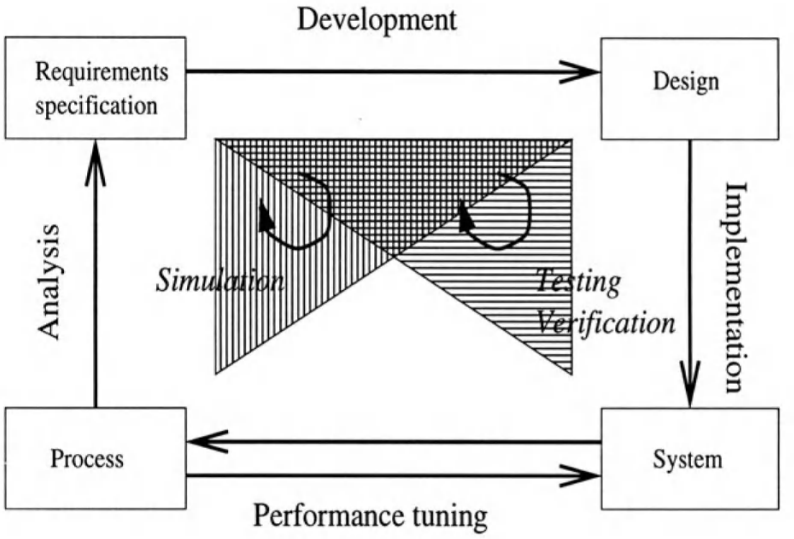

Fig. 2.3. Iterative software life cycle

2.2.3 Refinements and Formal Methods

In each life cycle model the transition from one phase to the next in the software life cycle is actually a *refinement*: on each phase details and design decisions about the "how to" are added. They constitute potential sources for errors within the development. In particular when producing safety-critical systems it is therefore necessary to confirm that the refinements have been performed correctly. If the confirmation is obtained by testing, simulation, or measurements we say that the transformation has been *validated*. *Verification*, on the other hand, confirms correctness on an abstract and formal level by formal proofs. Because verification needs a formal framework, the first development step from the customer's requirements to the system specification

can only be validated, since the requirements represent an informal system description.

These transitions are the main places where formal methods are being used. Each transition gives rise to proof obligations which need to be processed. Iterations require the verification processes to be done all over again, Although in many cases much of the earlier verification work is of relevance to the current verification, this is a costly and time-consuming process. Especially for iterative life cycles where such transitions are made over and over again, automatic computer support is essential.

2.3 Degree and Scope of Formal Methods

In an informal software development process, verification and validation steps are done by the software engineer in a way of convincing him/herself that things are correct. When formal methods are applied more rigorously, these steps are more standardized and thus more objective. However, they involve more effort. J. Rushby distinguishes four levels of rigor (or degree of formalization) [Rushby, 1993]:

0 *No use of formal methods:* All specifications are written in natural language (possibly with graphical elements like diagrams), a programming language, or pseudo-code. The analysis (if there is any) is done in an entirely informal way by review and inspection or by testing.

1 *Use of concepts and notation from (discrete) mathematics:* Some of the requirements and specification documents are replaced with (or augmented by) notations derived from mathematical logic and discrete mathematics. As before, analysis and proofs are done in an informal way. The main advantage of this level over level 0 is the compactness and preciseness of the employed notation which helps to reduce ambiguities, while smoothly fitting into existing development procedures.

2 *Use of formalized specification languages with some tool support:* The specification languages are supported by mechanized tools ranging from syntax-controlled editors, syntax checkers and pretty-printers to type checkers, interpreters and animators. Proofs on this level are enforced rigorously, although the proofs are still done in an informal way. However, several methods (e.g., Larch [Guttag *et al.*, 1993; Taylor, 1990], RAISE [George *et al.*, 1992], VDM/SL [Jones, 1990], Z [Spivey, 1988; Wordsworth, 1992]) provide explicit means for performing formal proofs manually.

3 *Use of fully formalized specification languages with comprehensive support environments, including mechanized theorem proving or proof checking:* Formal specification languages have a fully defined and concise semantics and provide corresponding formal proof methods. The complete formalism of proving methods permits and encourages the use of mechani-

cal techniques like theorem proving, model checking and proof checking. Classical examples here include HOL [Gordon, 1988], Nqthm [Boyer and Moore, 1988] ACL2 [Kaufmann, 1998], EVES [Craigen and Saaltink, 1996], PVS [Crow *et al.*, 1995], or KIV [Reif *et al.*, 1997].

In this book, we naturally focus on level 3, because only the use of concisely defined specification languages (or pure logic) allows to use formal deductive systems and hence possibly automated theorem provers.

In practice, employing formal methods on level 3 for the entire software product and the entire software life cycle is far from realistic. Even if fully automatic and powerful tools existed the effort would still be much too high, except for highly sensitive systems. Therefore, the levels of formalization and the phases/places where they are used will vary from project to project. In [Kelly, 1997], a distinction on the *scope* of the use of formal methods is made:

1. *Stages of the development life cycle:* Here, formal methods are used for specification of certain stages of the life cycle and for verification of translations between phases. In general, the biggest payoff from the use of formal methods occurs in the early stages. Errors, coming in during these stages are very costly if they proceed undetected through the development stages. Bug-fixes in a product already shipped can be up to 60–100 times more expensive [Pressman, 1997] (see also Figure 2.2). Furthermore, formal methods used during the early phases often help to clarify customer requirements.
2. *System components:* Instead of formally verifying the entire system, verification is restricted to components or submodules which are identified as extremely sensitive and critical for the operation of the entire system. Typically, core algorithms and (communication, authentication, or access) protocols are of primary importance[1].
3. *System functionality:* Although the traditional formal methods' approach is to "prove correctness" of the system, in practice, it is often sufficient to show only important system properties, such as termination, deadlock freedom, or safety properties (e.g., a secret key cannot be read from the outside). As becomes obvious from the above examples, it is often more important to show the absence of negative properties (e.g., deadlocks) than the confirmation of all positive properties.

2.4 Specification Languages and Proof Tasks

With different levels of formalization and various scopes of the use of formal methods, considerable flexibility of formal verification in actual projects can

[1] Nevertheless, the other components should not be neglected, because they — although non-critical and often rather trivial in design and coding — can contain errors which are costly and difficult to detect.

be obtained. Nevertheless, stages, components, or functionalities of a system which are critical and for which properties have to be guaranteed must be developed with a high degree of formalization, using formal specification languages.

Formal specification languages provide a precise vocabulary, syntax and semantics for the description of system characteristics. Numerous different specification languages have been developed and it is outside the scope of this book to give an overview (see, e.g., [URL Formal Methods Archive, 2000]). Rather do we note here that most specification languages are only suited to describe certain development phases and characteristics. So, usually, more than one formalism will be used for handling different phases of the life cycle and to show different kinds of system properties. Furthermore, instead of a specification language, often a specific logic is used directly. Examples are here first-order logic, higher-order logic, temporal logics (like CTL [Burch *et al.*, 1990] for hardware verification) or modal logics for the verification of security protocols.

Formal specification languages and these logics share the property that *all* operations carried out with them (e.g., verification, refinement, matching of specifications, retrieval of specifications) involve *formal reasoning*, that is checking of logical validity by virtue of the syntactical form, and not of the content. This means that reasoning is done with a relatively small number of formally defined inference rules. In this book, we will focus on such activities. In the following, we will call the task of showing the validity of a formula (with respect to a given logic) a *proof task* or *proof obligation*. Such a proof task is the central piece of information for the application of an automated theorem prover. Regardless of the specification language used or the kind of operation carried out, such a proof task contains the following components:

- the *theorem(s)* to be proven,
- the corresponding *specification(s)*, or formulas describing the system properties,
- *axioms* and *theories*, defining the syntactic functions, constants, and operators,
- the *source logic* which is the basis for reasoning, and
- (possibly) additional information (e.g., semantics, control information).

Example 2.4.1. Consider the following VDM/SL [Dawes, 1991] specification of a function returning the head of a list. In VDM, a specification contains the signature, preconditions, and postconditions. In our case, the precondition has to ensure that the given list must not be empty. In the postcondition we specify the return value which represents the head of the list. "$\frown$" denotes the append operator on lists, [] the empty list. The variable m holds the result of our function.

$$\begin{array}{l} \textsc{Spec}\text{: } \mathsf{head}(l) : \mathsf{list} \quad m : \mathsf{list} \\ \textsc{Pre}\text{: } \quad l \neq [\,] \\ \textsc{Post}\text{: } \exists i : \mathsf{item}, \exists l_1 : \mathsf{list} \cdot \; l = [i] \frown l_1 \wedge m = [i]; \end{array} \tag{2.1}$$

When we apply this function to a list of the structure `cons(a,l)` for an item `a` and list `l`, a system property we might want to show is that head(`cons(a,l)`) = `a`. The corresponding proof task is shown in Table 2.1.

Before we will focus on techniques of automatic handling of such proof tasks in the next chapter and on characteristics of the proof tasks in Chapter 4, we will shortly review the current situation of formal methods use in industry.

Theorem:	$\forall$ a : list $\forall$ l : list : head(cons(a, l)) = a
Specification:	See (2.1)
Axioms:	a number of axioms, defining the properties of cons and append $\hat{}$, e.g., associativity: $\forall X, Y, Z$: list · $(X \hat{} Y) \hat{} Z = X \hat{} (Y \hat{} Z)$
Source Logic:	VDM's three-valued logic LPF (logic of partial functions, [Barringer *et al.*, 1984]).
Additional Information:	none

Table 2.1. Proof task for the specification in Example 2.4.1

2.5 Current Situation

Formal methods and their tools play a prominent role in the area of hardware design, mainly in the design and production of highly complex processing elements. The breakthrough for the widespread use of formal methods in this area probably came in 1994 when Intel shipped versions of its Pentium processor with a bug in its division algorithm. This bug eventually costed Intel almost 500 million dollars. Using formal methods and automatic model checking, this bug would have been detected. This incident caused the industry to focus on formal methods tools for the design of their chips. Highly efficient algorithms (e.g., model checkers like SPIN [Holzmann, 1991] or SMV [McMillan, 1993]) were integrated into the CAD systems and silicon compilers and led to much improved reliability and reduced cycle times. Highly complex chip designs can be produced which run flawlessly on the first fabrication run, as reported by Silicon Graphics [Lowry *et al.*, 1998]. This is particular important in the light of Moore's Law which says that about every 18 months the speed and complexity of processor chips can be doubled. The high pressure to bring new chip designs to market on time requires support of formal methods tools.

On the other hand, quality of software and productivity of software developers (measured in debugged and tested lines of code per person-day) has developed quite differently. According to a recent study by Ed Yourdon and H. Rubin (see [Glass, 1999]) productivity has even declined by 13% and quality has moderately increased by 75% between 1985 and 1997. Therefore the question arises if the application of formal methods and formal methods tools cannot only increase reliability and safety of software, but also productivity of software development. Then, initially invested effort would pay back twofold. Currently, formal methods in the area of software engineering are primarily not applied to increase productivity, but only to guarantee specific properties of software in highly safety and security relevant areas. However, large projects carried out with formal methods (mainly on level 1 and 2) often revealed that productivity was the same or better than on comparable projects carried out using informal methods only. Additionally, a much higher quality of the software could be obtained.

Nevertheless, application of formal methods nowadays is usually restricted to highly sensitive areas as the following examples from the area of avionics, railway signaling, nuclear power control, and transaction processing show.

- The "Traffic alert and Collision Avoidance System" (TCAS) [Storey, 1996; Heimdahl and Leveson, 1996] is used to reduce the likelihood of midair collisions involving larger passenger aircraft. After detection of flaws in the original design, a formal specification on level 1 was developed (no tool support). The formalism was RSML (requirements state machine language) which is based on Harel's statecharts [Harel, 1987]. This specification was considered to be a big improvement over the original one in English. Later on, even consistency of that specification was checked automatically [Heimdahl and Leveson, 1996] and code was generated out of RSML specifications.
- The specification language VDM was used to specify another large avionics system, a "central control function information system" (CDIS) [Hall, 1996]). For CDIS which consists of approximately $80K$ lines of C-code, an abstract VDM model was specified in conjunction with concrete user interface definitions, definitions of the (concurrent) system behavior and a world model. During the design, this model was refined into more concrete module specifications. This product revealed a software defect rate (i.e., faults per thousand lines of code) twice as good as comparable projects without losing overall productivity.
- For the design of the flight warning computers for the Airbus 330 and 340, formal methods on level 2 have been used (for an overview of that system see, e.g., [Storey, 1996]). The software for these computers has been implemented in Ada, PL/M and assembler. A LOTOS specification for that program grew to about the same size as the implementation (around 27500 lines) and thus was too large to be handled by available formal methods tools. Furthermore, code generated from the LOTOS specification

was several orders of magnitudes too slow with respect to the requirements. Thus, using formal methods in this project was of little help.

- The avionics software of the Lockheed C130J was specified in a notation based on Parnas's tabular method [Heninger, 1980] and implemented in an annotated subset of Ada. This method led to software being close to "correct by construction" [Croxford and Sutton, 1996], thus reducing the amount of required software reworking considerably.
- The signaling system for the Paris Métro (SACEM) [Carnot *et al.*, 1992] was specified using the language B [Abrial *et al.*, 1991]. About 2/3 of its 21000 lines of Modula 2 code was safety critical and thus subjected to formal specification and verification.
- The automatic shutdown system of the Darlington nuclear power plant in Canada was developed using a formal specification method based on Parnas's method [Parnas, 1994]. This project is described in [Archinoff *et al.*, 1990] and surveyed in [Storey, 1996].
- The IBM transaction processing system CICS [Houstan and King, 1991] was one of the first, but nevertheless largest projects with consequent usage of formal methods (here: specification in Z [Spivey, 1988]). Of its approximately 250000 lines of code, 37000 were produced from Z specifications and further 11000 were partly specified in Z. The developers could measure a reduction of about 60% in faults with a overall cost reduction of about 9%.

Other industrial case studies in formal specification have been performed in the areas of databases (e.g., [Bear, 1991]), medical systems (e.g., a cyclotron controller [Jacky, 1995]), security systems (e.g., security policy model for NATO ACCS [Boswell, 1995]), and transportation systems. This list is by far not complete; for an overview see, for example [URL Formal Methods Archive, 2000].

When looking at industrial applications on the formality level 3, only very few projects on software systems can be found. According to surveys [Gerhart *et al.*, 1994; Clarke and Wing, 1996; Craigen *et al.*, 1993], most projects on level 3 have been carried out in the area of hardware development and verification. Typical examples are verification of a Cache coherence protocol [Clarke *et al.*, 1993], processor design for the AAMP5 processor [Miller and Srivas, 1995], for the T800 [Barrett, 1989], the Motorola 68020 [Boyer and Yu, 1996], the FM8502 [Hunt, 1986], or the Motorola CAP [Brock *et al.*, 1996]. In these projects and case studies, mostly interactive theorem provers have been used.

Most projects in highly formal software verification and design have been made using the theorem provers PVS [Crow *et al.*, 1995], HOL [Gordon, 1988], the Boyer-Moore theorem prover [Boyer and Moore, 1988; Boyer and Moore, 1990], KIV [Reif *et al.*, 1997], or others[2]. Here, always small, but

[2] The WWW pages of these systems list a large number of projects and case studies carried out with the respective theorem prover.

highly critical parts are investigated. Most of them stem from the following areas (as far as the author is aware of):

- communication protocols have gained much importance with widespread use of computer networks and distributed applications. Protocol verification with high computer support are reported e.g., in [Rajan *et al.*, 1995; Dahn and Schumann, 1998]. Automatic optimization of protocol stacks is subject of a project using NuPrl [Kreitz *et al.*, 1998]. Numerous attempts to protocol specification and verification can be found in the IFIP Transactions on Protocol Specification, Testing and Verification, e.g., [Jonsson *et al.*, 1991].
- process communication, interaction, and issues of synchronization are extremely important, in particular in real-time controlling systems (e.g., space-craft controllers, plant controllers). For example, errors in this area occurred in the Mars Rover of the Pathfinder mission (cf. [Lowry *et al.*, 1998]), or in the Deep-Space-One project [Lowry *et al.*, 1997]). Probably the largest project in the area was performed by SCC (Secure Computing Cooperation) on the LOCK operating system [URL Secure Computing, 2000]. Here, issues of non-interference and access control policy were verified based on a specification of about 102K lines of Gypsy [Ambler *et al.*, 1977; Good *et al.*, 1988].
- controllers (implemented in software and hardware) have been the topic of several case studies, e.g., an air-bag controller [Reif, 1998], or the outer space aid for extravehicular activities SAFER [Kelly, 1997]).
- authentication protocols are extremely important for ensuring safety of distributed applications (e.g., electronic commerce). Here, many formal methods have been developed. Some of them provide considerable computer support and a high degree of automatic processing. Many such protocols have been studied and various errors have been detected. [Paulson, 1997a; Paulson, 1997b] uses the interactive theorem prover Isabelle [Paulson, 1994]; [Kessler and Wedel, 1994] uses a special-purpose PROLOG program, and [Craigen and Saaltink, 1996] uses ZEVES. The system PIL/SETHEO has been developed by the author (see Section 6.1) and uses the automated theorem prover SETHEO. Several approaches to use model checking in this area are reported in [Kindred and Wing, 1996] or [Bolignano, 1998].
- Synthesis of high-performance scheduling algorithms for solving logistics problems is done with an automatic tool (PLANWARE, [Burstein and Smith, 1996]) using a category-based formal method.
- Synthesis of linear macro code for numerical calculations is done with the system AMPHION (see Section 4.7.1). AMPHION uses the automated theorem prover SNARK [Stickel *et al.*, 1994] to automatically synthesize FORTRAN programs in the area of astrodynamics and fluid dynamics. This tool has recently been extended to synthesize state-estimation code (C++) for avionics navigation.

All these projects share great overheads and considerable verification times limiting the scale-up to larger systems (or parts of software systems). A comprehensive overview over this field and numerous projects can be gained from the conference series "Formal Methods" [Bjørner *et al.*, 1990; Woodcock and Larsen, 1993; Naftalin *et al.*, 1994; Gaudel and Woodcock, 1996; Fitzgerald *et al.*, 1997], "Knowledge-Based Software Engineering" [Selfridge *et al.*, 1991; Johnson, 1992; KBSE-8, 1993; KBSE-9, 1994; KBSE-10, 1995; KBSE-11, 1996], "Automated Software Engineering" (e.g., [Welty *et al.*, 1997]), or "Computer Aided Verification" (e.g., [Dill, 1994; Alur and Henzinger, 1996; Hu and Vardi, 1998]). According to E. Clarke and J. Wing ([Clarke and Wing, 1996], p. 638), formal methods have "[...] already demonstrated success in specifying commercial and safety-critical software and in verifying protocol standards and hardware designs." They conclude that in the future "we expect that the role of formal methods in the entire system development process will increase especially as the tools and methods successfully in one domain carry over to others."

3. Processing of Logic

3.1 Automated Deduction

In this chapter, we will focus on methods and systems for automated deduction in first-order predicate logic. *Automated deduction* (or theorem proving with computer support) concerns the mechanization of deductive reasoning within a formal system. Of particular interest for us are methods for handling first-order predicate logic. Throughout this book we will refer to such an automatic system as an *automated theorem prover* (ATP). Already 1960, programs for automated theorem proving have been implemented [Gilmore, 1960; Wang, 1960; Prawitz *et al.*, 1960]. Since then, a large variety of automated theorem provers based on numerous proof methods have been developed. For the history of automated theorem proving and the most prominent milestones see, e.g., [Kelly, 1997], p. 84–86, or [Furbach, 1998] for tableaux calculi; for a history of logic in general see, e.g., [Kneale and Kneale, 1984].

In this chapter, we will introduce the basic notations for first-order logic and the most popular calculi for its automatic processing. Then, we will focus on a specific system, the automated theorem prover SETHEO. SETHEO is a high-performance automated theorem prover which has been used for the case studies in this book. We will also shortly discuss other automated theorem provers and the general characteristics of such systems.

For further reading, a number of textbooks and survey articles exist. [Bibel and Schmitt, 1998] is a three-volume book presenting a survey on the state of the art in automated deduction in Germany. Volume I focuses on theoretical foundations (calculi and methods), volume II describes systems and implementation techniques (interactive theorem proving, representation and optimization techniques, parallel theorem proving and cooperation), and Volume III its applications in mathematics, software development and hardware design. An overview of the currently most powerful implementations of automated first-order provers can be found in the documentation of the annual system competition CASC which is held during the conference on automated deduction (CADE). System descriptions and results appear in [Sutcliffe and Suttner, 1997; Suttner and Sutcliffe, 1998; Suttner and Sutcliffe, 1999].

Classical textbooks in automated deduction include [Loveland, 1978], [Chang and Lee, 1973], [Bibel, 1987], [Bibel and Eder, 1993], [Lloyd, 1987], or

[Robinson, 1979], just to name a few. Handbooks of logic (which also focus on automated deduction) are, e.g., [Gabbay, 1993; Abramsky *et al.*, 1992; Gabbay and Guenthner, 1983]. Books, describing single systems (often in form of a reference manual) and their applications are, e.g., [Boyer and Moore, 1988] for the Boyer-Moore prover, [Paulson, 1994] for the interactive prover Isabelle, [Constable *et al.*, 1986] for NuPrl, [Bündgen, 1998] for ReDuX (a term rewriting system), [Guttag *et al.*, 1993] for the interactive system Larch, or [Wos, 1988; Wos, 1992] for the OTTER theorem prover. An extensive overview of automated reasoning systems can be found in the Stanford university automated reasoning archive [URL ARS-Repository, 2000].

3.1.1 A First-Order Language

First-order logic, like any formal logic, is based on sentences of a formal language over a given alphabet. Specifically, a first-order logic can be defined over the following alphabet of (potentially infinite many) symbols:

$X_1, X_2, \ldots$	variables
$a_1, a_2, \ldots$	individual constants
$p_1, p_2, \ldots$	predicate symbols
$f_1, f_2, \ldots$	(syntactic) function symbols
$\wedge, \vee, \neg, \rightarrow, \leftrightarrow$	connectives
$\forall$, $\exists$	quantifiers
(,),“,” (Comma)	punctuation symbols

Then, a *term* can be constructed from elements of this alphabet in the following way:

1. variables and individual constants are terms.
2. if f_i is a function symbol and $t_1, \ldots, t_n$ are terms, then $f_i(t_1, \ldots, t_n)$ is a term.
3. the set of all terms is constructed by rules 1. and 2.

In first-order logic, terms usually are interpreted as objects, i.e., things which have properties, about which assertions are made and to which functions are applied. An *atomic formula* is defined as $p_i(t_1, \ldots, t_n)$ where p_i is a predicate symbol of arity n and $t_1, \ldots, t_n$ are terms. A *literal* L is is an atomic formula A or its negation $\neg A$. Atomic formulas (or literals) are the simplest expressions in our first-order language which are interpreted as assertions. With the help of logical connectives, more complex formulas can be constructed. Hence, a *well-formed formula* (wff) is defined as follows:

1. every atomic formula is a wff.
2. for wff's A and B, the formulas $\neg A$, $A \wedge B$, $A \vee B$, $A \rightarrow B$, $A \leftrightarrow B$, $\forall X_i \cdot A$, and $\exists Y_i \cdot A$ are well-formed, where X_i, Y_i are variables.
3. the set of all well-formed formulas is generated by 1. and 2.

Of course, this language is only a skeleton for a first-order language. For practical purposes, the names of the symbols are arbitrary strings. For example, a constant can be named *nil* or [ ]. Similarly, function symbols can be used in prefix and infix notation[1].

Example 3.1.1. The formula

$$\forall L \cdot (L = [\,] \vee (\exists H, T \cdot L = \mathtt{cons}(H, T))$$

is a well-formed formula. Here, the only predicate symbol is equality ("="), written in the usual infix notation. Variables are L, H, T. [ ] is an individual constant. `cons` is a function symbol with arity 2. One can imagine that [ ] denotes the empty list and `cons` represents the list construction operator *cons*.

3.1.2 Model Theory

Above we have described the formal language of first-order logic. Now, we discuss how formulas of such are logic are interpreted. An *interpretation* consists of a (nonempty, abstract or concrete) *domain of discourse* $\mathcal{D}$ and a mapping relative to $\mathcal{D}$ which assigns a semantic value or meaning to each well formed formula as well as to every well-formed constituent of that formula. In our case, an interpretation would assign a truth value (TRUE or FALSE) to predicate symbols and values to each function symbol and constant. If semantic rules are given that systematize an interpretation for each well-formed term or formula, the language is said to be *interpreted.* Of course, for a given alphabet, many interpretations can exist.

Example 3.1.2. For the symbols [ ] and `cons` from the previous Example 3.1.1 one can easily think of several different interpretations as shown in Table 3.1.

Interpretation	Symbol	meaning
I_1	$\mathcal{D}$ = lists	
	[ ]	empty list
	cons(H,T)	list constructor
I_2	$\mathcal{D}$ = traces	
	[ ]	empty trace
	cons(H,T)	prepend element to trace
I_3	$\mathcal{D} = \mathbf{N}$	
	[ ]	0
	cons(H,T)	add number to sum $H + \Sigma T$

Table 3.1. Different interpretations for symbols [ ] and `cons`

[1] The notation in this book is kept somewhat similar to PROLOG and SETHEO's input language. Thus, names of variables always start with an upper-case letter.

Given an interpretation I with domain $\mathcal{D}_I$, we want to determine the truth value of a well-formed formula. A *valuation function* v is a function from the set of terms the set $\mathcal{D}_I$ (with elements $\overline{a}_i, \overline{f}_i, \overline{p}_i$) which assigns inductively values to the constants and functions: $v(a_i) = \overline{a}_i \in \mathcal{D}_I$ for a constant a_i and $v(f_i(t_1, \ldots, t_n)) = \overline{f}_i(v(t_1), \ldots, v(t_n)) \in \mathcal{D}_I$ for a syntactical function symbol f_i of arity n and terms $t_1, \ldots, t_n$. A valuation v *satisfies* a formula $\mathcal{F}$, if

1. v satisfies $p_i(t_1, \ldots, t_n)$ if $\overline{p}_i(v(t_1), \ldots, v(t_n))$ is TRUE in $\mathcal{D}_I$.
2. v satisfies $\neg\mathcal{F}$, if v does not satisfy $\mathcal{F}$.
3. v satisfies $\mathcal{F} \wedge \mathcal{G}$ $(\mathcal{F} \vee \mathcal{G})$, if v satisfies $\mathcal{F}$ and (or) $\mathcal{G}$. Similar rules apply to the other logical connectives.
4. v satisfies $\forall X_i \cdot \mathcal{F}$ with variable X_i, if every valuation v' which is equivalent to v except for X_i, i.e., $v(X_j) = v'(X_j)$ for every $j \neq i$ satisfies $\mathcal{F}$.
5. v satisfies $\exists X_i \cdot \mathcal{F}$ with variable X_i, if there exists a valuation v' which is equivalent to v except for X_i satisfies $\mathcal{F}$.

With this definition it is possible to "calculate" the truth value for a given formula with respect to an interpretation. A formula $\mathcal{F}$ is *satisfiable* under interpretation I, if there exists at least one valuation v for which $v(\mathcal{F})$ evaluates to TRUE. $\mathcal{F}$ is *valid* ($\models \mathcal{F}$) if every valuation of $\mathcal{F}$ for every I yields TRUE. Then, $\mathcal{F}$ is called a *tautology.*

Example 3.1.3. For the domain of natural numbers, the usual interpretation of the syntactical constant `plus` would be the addition of natural numbers. Thus a valuation v of the term `plus`(*three*, *five*) would be $v(\texttt{plus}(\textit{three}, \textit{five})) = v(\textit{three}) + v(\textit{five}) = 3 + 5 = 8$. Here, $\overline{\texttt{plus}}$ is "+", $\overline{\textit{three}}$ is "3".

A formula $\texttt{plus}(X, \textit{three}) > \textit{five}$ is satisfiable, because the formula is TRUE for $X = \textit{five}$. However, it is not valid, because the valuation $v(X) = 1$ will not satisfy it.

Example 3.1.4. The validity of a formula always depends on the interpretation, as in the following formula $\mathcal{F} \equiv (\forall A, B \cdot A \times B = B \times A)$. When interpreted in the domain of natural numbers, $\mathcal{F}$ is valid ($\models_{\mathbf{N}} \mathcal{F}$). However, for arbitrary matrices, $\not\models_{\text{Matrix}} \mathcal{F}$. In this example, $\times$ denotes the respective multiplication operator in both cases.

Model-based deduction techniques (e.g., propositional satisfiability test, Model Checking) use algorithms which try to systematically test all valuations of a formula. This, of course, is only possible, if the domain is finite (or at least has finite model properties [Burkart, 1997]).

Example 3.1.5. Propositional logic is syntactically restricted in such a way that all predicate symbols have arity 0, i.e., there are no terms. Interpretations are made over the domain of boolean values, i.e., $v(p_i) \in \{\text{FALSE}, \text{TRUE}\}$. The validity of a formula can be checked by systematically evaluating the truth

value of the formula for all possible valuations of the occurring predicate symbols (2^n valuations for n predicate symbols).

Efficient propositional theorem provers however, use algorithms with a much lower complexity in the average case. These algorithms are mostly based on the Davis-Putnam procedure [Davis and Putnam, 1960].

3.1.3 Formal Systems

In contrast to model theory which investigates semantic values and interpretations, *proof theory* deals with the pure *syntactical* issues of formal systems. Proof theory therefore is the basis for automated theorem proving. A *formal system* consists of a formal language, axioms (i.e, statements of that language which are taken as given) and a set of inference rules or other means of deriving further statements (called theorems). The set of axioms and inference rules is called a *deductive system*, a set of axioms with all derivable theorems from it a *theory*. A *proof* then is a series of purely syntactic transformations according to the inference rules. $\vdash \mathcal{F}$ (not to be confused with the model-theoretic $\models \mathcal{F}$ as described above) means that $\mathcal{F}$ is a theorem in the given logic. This means that $\mathcal{F}$ is provable in that logic. When additional assumptions $\mathcal{A}_1, \ldots, \mathcal{A}_n$ are given, we write $\mathcal{A}_1, \ldots, \mathcal{A}_n \vdash \mathcal{F}$.

A formal system as described above usually is referred to as an (uninterpreted) *calculus*. Important properties of a calculus are consistency, completeness and decidability. Consistency is of fundamental importance for any useful application. A formal system is *consistent*, if it is not possible to derive from its axioms both a formula and its negation. If for some formula $\mathcal{F}$ both $\vdash \mathcal{F}$ and $\vdash \neg\mathcal{F}$ holds, the system is inconsistent and thus it is possible to derive any formula. *Completeness* (for different notions see, e.g., [Kelly, 1997], pp. 80) usually means that for every well-formed formula $\mathcal{F}$ either $\mathcal{F}$ or $\neg\mathcal{F}$ is a theorem.

Finally, a formal system S is *decidable* if there exists an algorithm for determining whether or not any well-formed formula is a theorem or not. S is *semi-decidable* if there exist algorithms which can recognize all theorems of S. This means that, when given a theorem, the algorithm eventually must terminate with a positive result. If given a non-theorem, the algorithm need not terminate, but if it does, it must return the result "non-theorem".

First-order predicate logic is semi-decidable[2]. This means that automated theorem provers for first-order logic terminate only when given a theorem to process.

Now we shall regard the question concerning the relation of the truth value of a formula under a given interpretation and its syntactical derivability. For a formal system S let Γ be a set of well-formed formulas from S, and $\mathcal{F}$ be a well-formed formula. Then, S is *sound*, if $\Gamma \models \mathcal{F}$ whenever $\Gamma \vdash \mathcal{F}$ (inference rules preserve truth). S is *semantically complete* with respect to

[2] Some weaker logics such as propositional logic are decidable.

an interpretation I if all formulas of S that are TRUE in I ($\models_I \mathcal{F}$) are theorems, i.e., $\vdash \mathcal{F}$ (*adequacy*). We call an interpretation I a *model* for S if every theorem of S is TRUE.

Based on this relationship, purely syntactic reasoning is sufficient for the determination of satisfiability or validity of a formula. Hence, theorem proving techniques which will be described in the following section are based on syntactical reasoning in calculi.

3.2 Theorem Proving

3.2.1 Introduction

We refer to a *theorem prover* as a system that, given a calculus (for some logic) and a formula, attempts to find a proof by repeatedly applying the inference rules of the calculus. There are many variations possible which concern extensibility (can the system be extended to handle a customized logic?), degree of automatization (which steps are performed automatically and where are user interactions required?), and closeness to underlying logic (how strong is the proof algorithm correlated with the calculus). A general observation is that provers which are close to the underlying logic and which are not extensible tend to be more automatic than extensible, generic theorem provers.

In this book, we focus on fully automatic theorem provers (ATPs) for first-order logic. ATPs usually accept formulas in first-order clausal normal form. This standardized form (each formula can be translated into this form, as will be described below) is easier to handle than arbitrary formulas. The provers' underlying calculi are specifically suited for first-order reasoning. There, these calculi are sound and complete. In this area, two major traditions of automated theorem proving are found: resolution-based theorem proving and proving with tableau-based calculi. Both kinds of calculi try to prove a formula by refutation. Instead of trying to prove $\mathcal{F}$ ($\vdash \mathcal{F}$), the prover tries to show that the negation $\neg\mathcal{F}$) is unsatisfiable.

Whereas the resolution-style calculi start from the axioms and generate new formulas (*synthetic calculus*), tableau-based methods create a specific data structure, a *tableau* out of the formula by breaking it down. Only when such a tableau meets certain properties, the formula is unsatisfiable. Hence, a calculus of that type is called an *analytic calculus*. In the following we will describe the basic resolution calculus and one kind of tableau-based calculus, namely model elimination. Model elimination is the basis for the automated theorem prover SETHEO which will be used throughout this book.

3.2.2 Clausal Normal Form

A normal form is a standardized form of formulas which facilitates manipulation of the formula. In automated theorem proving, *clausal normal form*

(CNF) is of particular importance. Almost all automated theorem provers for first-order logic work with formulas in CNF[3]. Each well-formed formula can be transformed into clausal normal form (or Skolem normal form) without affecting its refutability. However, a formula $\mathcal{F}$ and its transformation are not logically equivalent (in a strict sense of $\leftrightarrow$).

A *clause* c is a disjunction of literals $L_1, L_2, \ldots, L_n$. A *literal* is an atomic formula A or its negation $\neg A$. All variables occurring in a clause are implicitly universally quantified. A formula is in clausal normal form, if it is a conjunction of clauses $c_1, \ldots, c_m$. Thus a formula in CNF is of the form $\mathcal{F} \equiv \bigwedge_i \bigvee_j L_{ij}$ for literals L_{ij}. Hence, some authors (e.g., [Bibel, 1987; Andrews, 1986]) prefer the notion of a *matrix* for a formula in clausal normal form.

Example 3.2.1. In SETHEO clauses are written in a PROLOG-like notation, separating the positive (unnegated) atoms and the negated atoms by a <-. Each clause is furthermore terminated by a full-stop. Thus a clause

$$H_1 \vee \ldots \vee H_n \vee \neg T_1 \vee \ldots \vee \neg T_m$$

with atoms H_i, T_i is written as

$$H_1 ; \ldots ; H_n \text{ <- } T_1 , \ldots , T_m .$$

If the number of unnegated literals in a clause is less or equal to one, the clause is called a *Horn clause.* Formulas consisting only of Horn clauses play an important role in PROLOG and logic programming.

Any full first-order formula can be transformed into a corresponding set of clauses, using algorithms described e.g., in [Loveland, 1978; Nonnengart *et al.*, 1998; Chang and Lee, 1973]. Important transformation steps concern the removal of logical connectives other than $\wedge$ and $\vee$, moving of the quantifiers (*prenexing*), and *Skolemization.* Whereas universal quantifiers carry over to the clausal normal form, existential quantifiers must be removed. A formula

$$\forall X_1, \ldots, X_n \; \exists Y \cdot \mathcal{F}$$

is transformed into its Skolem normal form

$$\forall X_1, \ldots, X_n \cdot \mathcal{F}'$$

by replacing all occurrences of Y in $\mathcal{F}$ by $f(X_1, \ldots, X_n)$ with a new function symbol f. f is called a *Skolem function.* In the case of $n = 0$, we speak of a *Skolem constant* instead.

Example 3.2.2. The formula from Example 3.1.1

$$\forall L \cdot (L = [\,] \vee (\exists H, T \cdot L = \mathtt{cons}(H, T)$$

[3] ${}_3T^AP$ [Hähnle *et al.*, 1992] seems to be the only exception. It can handle arbitrary FOL formulas.

would be skolemized and written in SETHEO's clausal form notation as the following clause:

$$L = [\,]; L = \mathrm{cons}(f_1(L), f_2(L)) \text{ <- } .$$

where f_1 and f_2 are two Skolem function symbols.

3.2.3 Resolution

The resolution calculus was developed by A. Robinson in 1965 [Robinson, 1965]. It is a calculus for *refuting* formulas in clausal normal form. Since 1965 a large variety of different kinds of resolution calculi have been developed. Already in 1978, as pointed out in [MacKenzie, 1995], some 25 different variants are documented in [Loveland, 1978]. In order to demonstrate how resolution works, we focus on the most simple kind of resolution, binary resolution. This calculus has only one inference rule and one logical axiom, the empty clause, denoted by []. The resolution inference rule takes two clauses $L, K_1, \ldots, K_l$ and $\neg L', M_1, \ldots, M_n$ and produces a new clause:

$$\frac{L, K_1, \ldots, K_l \qquad \neg L', M_1, \ldots, M_n}{\sigma K_1, \ldots, \sigma K_n, \sigma M_1, \ldots, \sigma M_n}$$

where σ is a variable substitution such that $\sigma L = \sigma L'$. We say that L is *unifiable* with L'. Most machine-oriented calculi are based on unification. More precisely, a substitution σ is a mapping of variables to terms. When a substitution is applied, all variable occurrences are replaced by the corresponding terms. If two expressions (terms or literal) s, t are unifiable, then there exists a substitution which makes both terms equal. In general, there exist many such substitutions. Of particular interest are *most general unifiers* from which all other unifiers can be derived.

Example 3.2.3. When we try to unify the two terms $f(X)$ and $f(g(Y))$, we could find a substitution $\sigma = \{[X \backslash g(Y)], [Y \backslash a]\}$ which would make the terms equal (namely to $f(g(a))$). However, σ is not the most general unifier which is $\sigma_{mgu} = \{[X \backslash g(Y)]\}$.

The two terms $f(X)$ and a are not unifiable, as is $f(X)$ and X. In the first case, a *clash* occurs (different symbols cannot be made equal). In the second case, *occurs-check* prohibits unification, because variable X occurs in the term $f(X)$. In most PROLOG systems (which use unification without occurs-check), the attempt to unify both terms would yield endless loops or run-time errors.

A naïve algorithm for finding and applying a unifier has exponential complexity. However, by constructing auxiliary data structures, the complexity of the algorithm can be reduced considerably. E.g., the algorithm in [Corbin and Bidoit, 1983] has quadratic complexity with very little overhead. Even algorithms with almost linear complexity exist (e.g., [Martelli and Montanari,

1982]). These, however, exhibit a considerable overhead which makes them infeasible for unification of small terms.

A refutation in resolution can be found by repetitively applying the resolution inference rule, until the empty clause [] is obtained. Resolution is a complete refutation procedure for first-order logic. If a set of clauses is unsatisfiable (i.e., FALSE under all interpretations and all domains), then resolution will eventually terminate with the empty clause. Therefore, in order to prove a theorem, it is negated and a refutation for the negation is sought. Thus, if resolution terminates with the empty clause, the theorem is valid. If the original theorem is not true, resolution usually does not terminate.

Resolution is the basis for many efficient automated theorem provers. Most prominent examples are OTTER [McCune, 1994a; Kalman, 2001], Gandalf [Tammet, 1997], SNARK [Stickel *et al.*, 1994], SPASS [Weidenbach *et al.*, 1996] (which uses a superposition calculus closely related to resolution).

3.2.4 Model Elimination

Model elimination is also a complete and sound calculus for first-order predicate logic and is described in Loveland's textbook [Loveland, 1978]. Here, we present model elimination not in the linearized notation found in [Loveland, 1978], but in notation of connection tableaux [Bibel and Eder, 1993]. In contrast to the resolution-type calculi which generate new formulas from the old ones, tableau calculi work by building up a tree structure, a *tableau.* More precisely, a (clausal) tableau is a tree where the nodes are labeled by literals. Labels of siblings belong to the same instance of a clause.

Model elimination, as we describe it here uses a restricted form of tableaux, namely *connection tableaux.* Let S be a set of clauses $c_1, \ldots, c_n$. A connection tableau T for S is a tableau which is constructed using the following three inference rules.

1. The empty tableau only consists of a single (root) node, labeled ϵ. Then, a clause $c \in S$ (start-clause) is selected. Its literals comprise the child-nodes of the root. All leaf nodes are marked "open".
2. For an open leaf node labeled L from T select a clause $c \in S$, (all its variables replaced by new ones) such that one literal L' of c is unifiable with L and has the opposite sign (L and L' are *complementary*, i.e., $\sigma L = \neg\sigma L'$). Then, add the literals of c as new child nodes of L to the tableau. Mark the new leaf node, labeled L' as closed, the others as open. Apply the substitution σ to the entire tableau. This inference step is called *extension step.*
3. If the open leaf node N with label L has an ancestor node labeled L' such that L and L' are complementary with most general unifier σ, then apply σ to the entire tableau and mark N as closed (*model elimination reduction step*).

When a *closed tableau*, i.e., a tableau with each leaf node marked as closed can be constructed for S, then S is *unsatisfiable*. A proof of this completeness theorem can be found in most textbooks. Thus, model elimination finds a proof by refutation.

Example 3.2.4. The following simple example asserts that if for any pair of objects in a domain either a binary relation $p(X,Y)$ or the inverse relation $p(Y,X)$ holds, then for every object in the domain there exists an object such that both relations hold:

$$\forall X \forall Y \cdot (p(X,Y) \vee p(Y,X)) \rightarrow \forall V \exists Z \cdot (p(V,Z) \wedge p(Z,V))$$

After negation of the formula (necessary for refutation!) and conversion into clausal normal form, we obtain the following two clauses, where a is a Skolem constant: $(p(X,Y) \vee p(Y,X)) \wedge (\neg p(a,Z) \vee \neg p(Z,a))$. In (SETHEO's) clausal syntax we would obtain:

```
p(X,Y) ; p(Y,X) <-.
<- p(a,Z) , p(Z,a).
```

Figure 3.1 shows a closed tableau for the above set of clauses. The tableau has been constructed using the inference rules defined above. In the start step, we generate an empty tableau and select the first clause as the start clause. Its literals become the two (currently open) child nodes of the root. Then, we try to close the left node (labelled $p(X,Y)$) with an extension step into the second clause. This yields two new nodes, $\neg p(a,Z)$ and $\neg p(Z,a)$. The first node is closed by construction via the extension step and marked by a $*_E$. The substitution $X \backslash a$ is applied to the entire tableau. The second open leaf node $\neg p(Z,a)$ can now be closed using a reduction step: this literal and its direct parent $p(a,Y)$ are complementary. Thus they can be unified, the resulting variable substitution $Z \backslash a$ applied, and the node marked closed ($*_R$). Now, the left part of the tree is processed. The remaining open node can now be handled in the same way, finally yielding a closed tableau.

Besides SETHEO, which is described below, other prominent automated theorem provers are based on clausal-form model elimination like PTTP [Stickel, 1988; Stickel, 1989], Protein [Baumgartner and Furbach, 1994b], METEOR [Astrachan and Loveland, 1991], KOMET [Bibel *et al.*, 1994], or Parthenon [Bose *et al.*, 1989] .

3.3 The Automated Prover SETHEO

The prover SETHEO (SEquential THEOrem prover) is a high performance automated theorem prover for first-order logic in clausal normal form. It is based on the model elimination calculus described above and has been developed within the Automated Reasoning Group at the Technische Universität

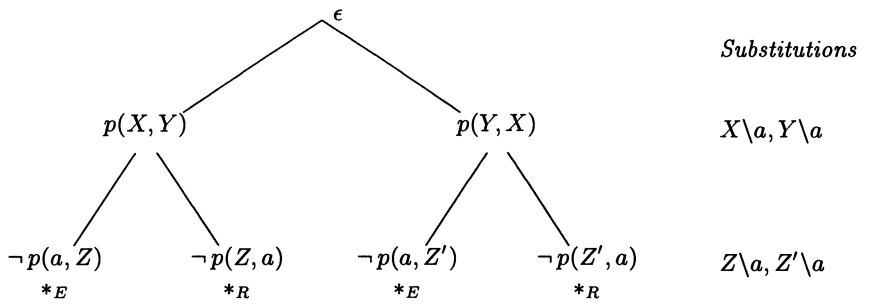

Fig. 3.1. A closed model elimination tableau. For better readability, variable substitutions are not applied to the literals.

München. The author is one of the principal designers of the system who has developed its architecture and implemented early versions of SETHEO. In the following we describe its calculus and proof procedure and various important extensions. Then we focus on the system's architecture which is built around an abstract machine.

3.3.1 Calculus and Proof Procedure

The calculus underlying SETHEO is model elimination in the form as described above. However, the design of SETHEO forced two minor changes:

- In the extension step described above, literal $\neg L'$ can be at an arbitrary position in clause c. Inspired by PROLOG's procedural reading for Horn clauses and from [Stickel, 1989], our implementation relies on a procedural reading: for the extension step the first literal of a clause must be selected. This requires to transform all clauses into sets of *contrapositives*. From a clause $L_1, \ldots, L_n$ with n literals n contrapositives are generated. In order to underline the procedural meaning, the <- is replaced by :-.

  $$\begin{aligned} L_1 &\text{ :- } \neg L_2, \neg L_3, \ldots, \neg L_n. \\ L_2 &\text{ :- } \neg L_1, \neg L_3, \ldots, \neg L_n. \\ &\ldots \\ L_n &\text{ :- } \neg L_1, \neg L_2, \ldots, \neg L_{n-1}. \end{aligned}$$

  Then, only the first literal, called *head*, can be used for the extension step.
- For the start step as described above, each clause can be used as a start clause. In SETHEO, we only can select a start clause from all clauses which only consist of negative literals[4].

[4] Again this is inspired from PROLOG where the *query* consists of one or more negative atoms, e.g., as in `?- loves(mary,X)` ("Who loves mary?").

The construction of a model elimination tableau involves *search*, since each open subgoal can induce the application of different inference rules with different clauses. Therefore, SETHEO (as all other model elimination provers) performs depth-first search. This means, the search tree spanned by the formula and the inference rules (OR-tree) is traversed in a left-to right depth-first manner with backtracking. In order to achieve completeness (i.e., to avoid endless loops), iterative deepening [Korf, 1985] is performed. SETHEO offers a wide variety of different completeness bounds (see, e.g., [Letz *et al.*, 1992; Letz *et al.*, 1994; Goller *et al.*, 1994; Moser *et al.*, 1997]). Most often, however, the depth of tableau (called A-literal depth) or the number of edges ("inference bound") is used.

3.3.2 Extensions

The basic calculus and proof procedure has been extended in various directions. The main goal was to increase the efficiency of the prover. Additional inference rules allow for an effective way of restricted lemma usage. Thus the proof length is reduced and we may get proofs at lower levels of iterative deepening. Pruning of the search space is accomplished by very powerful syntactic constraints which will be described below. Flexible dynamical control, like dynamic subgoal reordering or subgoal alternation adds to the system's performance. Finally, SETHEO has been extended to feature classical unit-lemmata and PROLOG-style built-ins for logic programming.

Additional Inference Rules. The calculus underlying SETHEO is a very weak calculus. This means, even for moderate simple formulas, the proof (measured in number of model elimination extension and reduction steps) can be extremely long. Therefore, the additional inference rule of *folding-up* has been developed and implemented [Letz *et al.*, 1994; Goller *et al.*, 1994]. This inference rule corresponds to Shostak's c-reduction [Shostak, 1976] and provides a restricted but highly efficient kind of lemma generation. This inference rule can be sketched as follows:

Folding up: Let N be an inner node with literal label L of the current tableau. Let us further assume that all descendents of N are closed (i.e., the subgoal N has been solved). The leaf nodes either have been closed by an extension step into a unit clause or by a reduction step into some node N' along the branch to the leaf. Let M denote the top-most node of these N's (i.e., that one which is closest to the root). If no reduction steps had been made to solve N, we set M equal to the root node of the tableau. Then, the solved node N can be moved up ("folded up") the branch just below M. Thereby, the node's label is negated into $\neg L$.

Later on, this already solved literal L can be used to close other open leaf nodes by a reduction step. Thus, L acts as a *lemma* as shown in the following example.

Example 3.3.1. Let us consider the following example. Given the formula $(p \vee q) \wedge (p \vee \neg q) \wedge (\neg p \vee q) \wedge (\neg p \vee \neg q)$ consisting of four clauses. Figure 3.2A shows a closed tableau for that formula. In Figure 3.2B, subgoal $\neg p$ just has been solved and has been subjected to folding up. Since the top-most node for the reduction steps in the solution is not above $\neg p$, this node can be moved up to the root (marked by a dashed arrow). When solving the subgoal $\neg q$ which is still open, our folded-up literal can be used as a lemma, yielding a much shorter proof as shown in Figure 3.2C. The node, solved by reduction into the lemma is marked by $*_{RF}$.

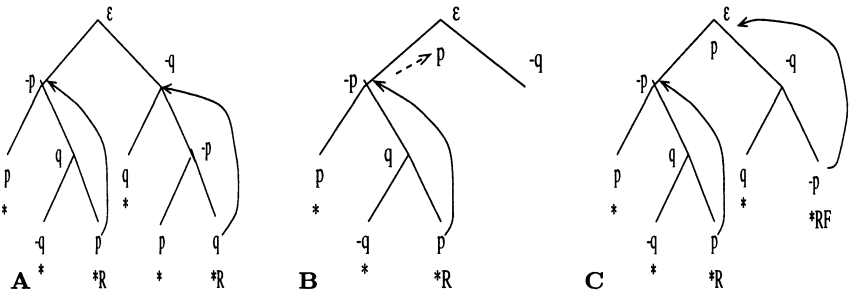

Fig. 3.2. Closed model elimination tableau (A). Folding up of $\neg p$ during tableau construction (B). Closed tableau with folded-up literal applied (C)

Similarly, an inference rule, called *folding down* is defined. Experiments showed that in many examples from applications, a considerable gain in performance can be achieved. On the other hand, folding-up increases the search space somewhat (there are more possibilities for performing a reduction step now) in case these lemmas are not useful.

Syntactical Constraints. Much of the search space results from the fact that during the search very often tableaux are generated which are not optimal, i.e., which contain redundant parts. So, for example, the same subgoal might occur over and over again in the tableau, or it is attempted to solve a subgoal again, for which it is already known to be unsolvable. Many of these redundancies can be removed if the calculus is restricted in such a way that certain kinds of tableaux are prohibited[5]. For example, the *regularity* condition forbids tableaux where two or more identical literals occur on one branch of the tableau. This situation just means that nothing new has been achieved and thus we can stop and backtrack. Other restrictions which are described in detail in [Letz *et al.*, 1994] concern tautologies or subsumable clauses.

[5] Of course, it must be and has been proven that these restriction do not affect completeness (cf. [Letz *et al.*, 1994]).

Example 3.3.2. Let us consider a subgoal $\neg p(a, b)$ and the symmetry rule for p: $p(X, Y) \rightarrow p(Y, X)$. When we perform an extension step into that clause, we obtain $\neg p(b, a)$ as the new goal. A second extension yields the subgoal $\neg p(a, b)$ which is identical to our initial subgoal. This means that no progress has been made. Therefore, this tableau which violates the regularity restriction can be discarded, saving further run-time.

In SETHEO, these restrictions are efficiently enforced using *syntactic constraints*. These constraints (which are generated during preprocessing) belong to the clauses and prohibit certain dynamic instantiations of the variables. For example, a clause

$$\ldots, p(X), \ldots : \texttt{[}X\texttt{]} \texttt{ != [a].}$$

is interpreted as follows. When this clause is used, all instantiations of variable X are monitored. If X gets instantiated to `a` at any given point, backtracking is initiated.

Further major redundancy during search is caused by the attempt to solve a subgoal over and over again. Subgoals which could not be solved can be remembered in an extra data structure and this information can be used to prohibit repeated proof attempts. This technique, called *failure caching* has been implemented, e.g., in METEOR [Astrachan and Loveland, 1991]. However, failure caching causes a huge overhead, in particular in the case of Non-Horn formulas. Therefore, SETHEO approximates failure caching by so-called *anti-lemmata* which are realized using SETHEO's constraint mechanism. Whenever backtracking occurs over a subgoal which already has been solved, an anti-lemma is generated, prohibiting the attempt to solve that (identical) subgoal again. These constraints are generated dynamically during the search.

Control. The size of the search space in model elimination strongly depends on the order in which the subgoals are selected and solved. Therefore, SETHEO does not only perform a static reordering of subgoals, but also a dynamic one during the search. This is done according to the general *first-fail* principle which prefers subgoals which are likely to fail. Additional powerful methods for subgoal alternation, delay of subgoals and look-ahead has been developed and implemented. For details see [Schumann and Ibens, 1998; Ibens and Letz, 1996].

Handling of Equality. The calculus of SETHEO as described above does not support reasoning with equality. The naïve approach to handle equations is to represent equality as a binary relation and to add the appropriate congruence axioms.

reflexivity	$\forall X \cdot X = X$
symmetry	$\forall X, Y \cdot X = Y \rightarrow Y = X$
transitivity	$\forall X, Y, Z \cdot X = Y \wedge Y = Z \rightarrow X = Z$
f-substitution	$\forall X_1, \ldots, X_n, Y_1, \ldots, Y_n \cdot X_1 = Y_1 \wedge \ldots X_n = Y_n \rightarrow f(X_1, \ldots, X_n) = f(Y_1, \ldots, Y_n)$
p-substitution	$\forall X_1, \ldots, X_n, Y_1, \ldots, Y_n \cdot X_1 = Y_1 \wedge \ldots X_n = Y_n \rightarrow p(X_1, \ldots, X_n) = p(Y_1, \ldots, Y_n)$

The first three axioms are obvious, the substitution axioms are used to carry over the equality of the LHS and RHS of an equation over function and predicate symbols (i.e., if $x = y$ then $f(x) = f(y)$)[6]. For function symbols and predicate symbols with an arity of greater than 1, the laws of substitution can be formulated in several different ways. For example, for an n-ary function symbol f, a different representation with n axioms might be ($1 \leq i \leq n$):

$$\forall X, X_1, \ldots, X_n, Y, Y_1, \ldots, Y_n \cdot X = Y \rightarrow f(X_1, \ldots, X_{i-1}, X, X_{i+1}, X_n) = f(Y_1, \ldots, Y_{i-1}, Y, Y_{i+1}, Y_n)$$

Whereas in the first representation, the equality of all sub-terms are tried to be shown "simultaneously", in the second case, we get a "step-by-step" approach, leading to a different shape of the search space. Although in the second case, we get n separate clauses, this representation is often preferred.

A second possibility of representing equations, together with the properties of the involved operators (e.g., a function symbols like"+") is the following *relational representation*: instead of introducing a predicate symbol for equality, each function symbol of arity n is transformed into a predicate symbol of arity $n + 1$, the last argument carrying the result of the operation. Thus, a $op(t_1, \ldots, t_n) = t$ becomes $p_{op}(t_1, \ldots, t_n, t)$. Here, no additional axioms have to be added to represent the properties of equality. Rather, axioms for representing properties of the operators (e.g., associativity and commutativity) have to be added.

This naïve approach of adding axioms, however, induces an extremely large search space, because specific redundancy elimination techniques of equality reasoning cannot be applied. This is in particular the case if there are deeply nested terms and if the proof requires many equational replacements to be made. Therefore, two specific techniques for handling equations have been integrated into SETHEO:

- *Lazy basic paramodulation and E*-SETHEO: The approach used here is to compile the properties of equality as a congruence relation into the terms and clauses by means of a variant of the STE-modification [Brand, 1975]. This transformation [Moser, 1996] usually yields a larger set of clauses, equational reasoning, on the other hand, can be performed more efficiently. In practice, this approach is relatively weak when used with large formulas

[6] These axioms should not be intermixed with those stating that a function is injective: if $f(x) = f(y)$, then $x = y$.

containing deep terms and only few steps of equality reasoning are required. Such proof tasks typically showed up in our case study in Section 6.3.

- *Linear completion*: Another approach, originally developed within Protein [Baumgartner and Furbach, 1994b] has been adapted by the author for the SETHEO system [Baumgartner and Schumann, 1995]. Here, all clauses belonging to the theory of equality are taken out of the formula and processed by linear completion (a method based on the Knuth-Bendix completion [Knuth and Bendix, 1970]). The result is a set of rules and clauses which can be applied within model elimination in a restricted way, taking much redundancy out of equational reasoning.

Handling of equations is an extremely complex area in automated deduction. Practical experiments with the naïve approach and the techniques described above showed that none of these methods is a clear winner for all kinds of proof tasks.

Unit Lemmata and Extensions for Logic Programming. Due to its relative closeness to PROLOG reasoning, SETHEO has been extended with several features for control of the prover and extended logic programming by procedural built-ins, global variables and unit-lemmas. SETHEO provides a variety of PROLOG-style built-in predicates (for a list see SETHEO's manual [Schumann and Ibens, 1998]). Most of them have procedural meanings (e.g., input/output) and can be used for extended logic programming (e.g., computation of answer substitutions, Section 7.7.1), combination of theorem proving and logic programming, control of the prover (e.g., setting/modifying completeness bounds), or planning, e.g., [Fronhöfer, 1996]. One of the core features is the concept of backtrackable global variables [Letz and Schumann, 1988; Schumann, 1991] which have a clean denotational semantics and allow for destructive variable assignment as known from procedural languages.

Finally, SETHEO has means for generating and using *unit-lemmata*, i.e., solved subgoals of length one. An efficient indexing data structure [Graf, 1996] allows for subsumption of generated lemmata. Thus, only a minimum of lemmata are kept. Nevertheless, unrestricted generation and use of lemmas usually lead to a search space explosion. Therefore, unit-lemmata are only available through a logic programming interface. This feature of SETHEO is for example used in the bottom-up preprocessor DELTA (Section 7.6.1).

3.3.3 System Architecture and Implementation

SETHEO is implemented as an abstract machine (SAM). This abstract machine is a variant of the Warren abstract machine [Warren, 1983] for the execution of PROLOG. For details on the SAM see [Schumann, 1991; Schumann and Ibens, 1998]. Single instructions of the SAM implement the basic operations of the calculus and proof procedure (unification, extension and reduction step, refinements, iterative deepening). Its tagged register architecture and multiple stacks allow for high performance and efficient memory

management (Figure 3.3). The abstract machine is implemented in C and running on various UNIX and Linux platforms.

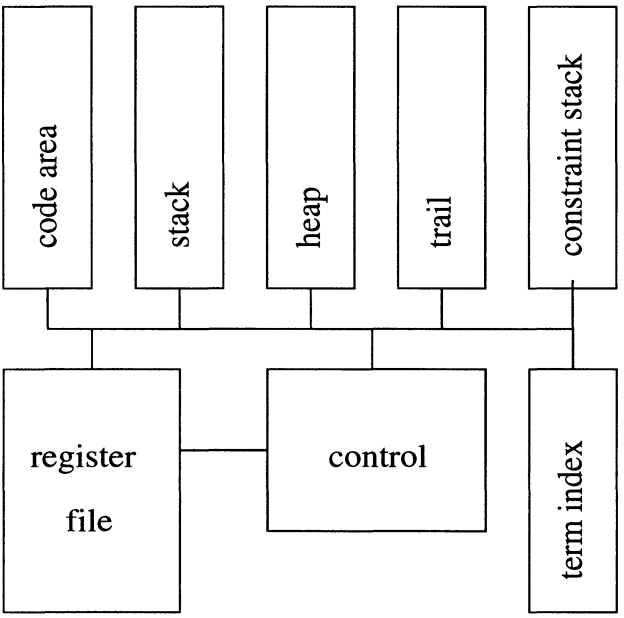

Fig. 3.3. Architecture of the SETHEO abstract machine SAM

This implementation approach, however, requires that each formula is first compiled into a sequence of machine instructions for the SAM. This compilation (including assembly) as well as preprocessing steps are performed by a separate program, called *inwasm*. The entire system architecture. is shown in Figure 3.4. When E-SETHEO's equality handling is used (optional), a preprocessing module *stexposu* converts the input into another set of clauses with compiled equality (as described above). Then, the preprocessor and compiler *inwasm* is invoked which produces a binary file for the SAM abstract machine. *Inwasm* performs the following steps: removal of pure (non-usable) parts of the formula, generation of static syntactic constraints, static reordering of subgoals and clauses, and generation/assembly of abstract machine code. The abstract machine SAM accepts a file, generated by the compiler and starts the search. When a proof is found, a machine representation of the closed tableau is produced. Furthermore, statistical data concerning the proof (e.g., number of inferences in the proof) and the search for the proof (e.g., number of unifications attempts, number of backtracking steps) is produced. This information, although not used in most applications, provides valuable feedback for tuning the prover. SETHEO is controlled (e.g., bounds, search mode,

parameters) by command-line parameters given to the individual modules (see [Schumann and Ibens, 1998] for a complete list and details). A variety of different parallel theorem provers have been implemented on top of this architecture. For a detailed discussion see Section 7.6.2.

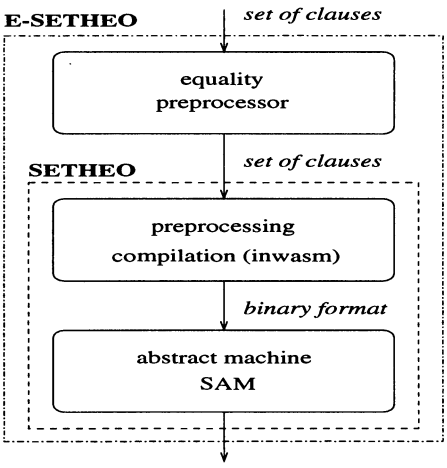

Fig. 3.4. System architecture of SETHEO and E-SETHEO

3.4 Common Characteristics of Automated Provers

In this chapter, we have presented the logical background of automated theorem proving. Although we have described only one system in detail, most first-order theorem provers exhibit similar capabilities. These common characteristics can be summarized as follows:

- Automated theorem provers for first-order logic are not flexible, i.e., they cannot easily be extended to other logics (except by translating the formulas).
- ATPs are working fully automatically. All parameters controlling the prover (usually very many) must be set in advance. Once the prover has been started, no reasonable user interaction (except aborting the run) can be given. All provers have a time-out facility to ensure that the prover stops if a proof could not be found within the given limits.
- ATPs usually very weak in detecting non-theorems, i.e., false formulas. Due to the semi-decidability of first-order logic, never all non-theorems can be

detected, but most existing provers[7] fail to recognize even the most trivial non-theorems as such. Rather, they do not terminate.

- ATPs are highly efficient implementations of search algorithms, nothing less or more. This means that they do not have any user environment (e.g., formula and proof management) as it is known from interactive theorem provers.
- ATPs are capable of producing a proof (as a sequence of applied inference steps and according substitutions) in contrast to other deductive techniques like model checking.
- ATPs are very powerful and exhibit a high inference rate (i.e., number of performed inference steps per second during the search). The individual inference steps, however, are very small (e.g., one resolution step or one model elimination extension or reduction step).

These basic capabilities should be kept in mind throughout the next chapter where we will focus on the characteristics of proof tasks which arise from applications in the domain of software engineering.

[7] Theorem provers based on model generation, like Satchmo [Manthey and Bry, 1988; Schütz and Geisler, 1996] or MGTP [Fujita *et al.*, 1992; Hasegawa *et al.*, 1992] are much better at this. However, they only work efficiently on range-restricted input formulas (e.g., on finite domains).

4. Characteristics of Proof Tasks

In this chapter, we will have a close look at the essential characteristics of the proof obligations which arise from the application and which are supposed to be processed by an automated inference system. These characteristics can be classified into different categories: *syntactic-* and *size-related issues*, *logic-related issues and validity*, and *system-related issues.*

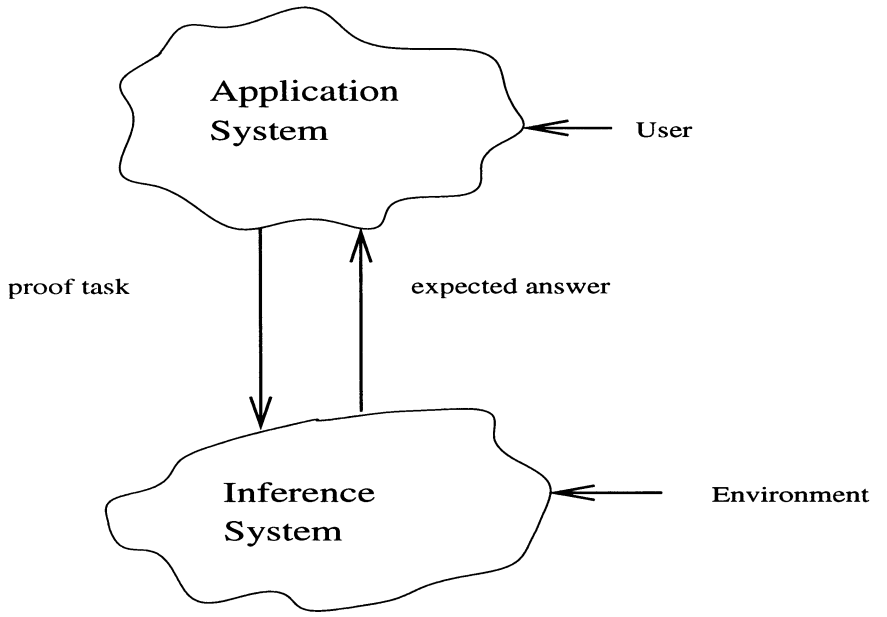

Fig. 4.1. Architecture of a general combination of an application system with an inference system

To this end, we consider a — at the moment hypothetical — system which combines a formal methods tool (or even a formalism only) with an automated inference system. Its "raw architecture" is depicted in Figure 4.1. Proof obligations are issued by the application system and processed by the infer-

ence component(s). Then, the application expects some kind of answer (e.g., a proof or a counterexample). The the proof obligations are also influenced by the environment of the combined system (e.g., users, resources).

All characteristics (which can be noted in a simple tabular form, Section 4.6) are essential for the decision whether an automated theorem prover can be used or not, and which requirements the ATP must fulfill for its successful application — topics which will be central in Chapter 5. The classification form and its entries will be illustrated by some well-known examples of existing tools and applications.

4.1 Seen from the Outside

4.1.1 Number of Proof Tasks

Probably the most important characteristic for the use of an automated theorem prover from the application systems' point of view is the *number* of proof tasks which are generated by the application system and which must be processed by the inference component. Of central importance is the number of proof obligations per "typical" session or action performed with the application system. There are applications where a typical session only generates a few proof obligations (e.g., verification of protocols, program synthesis). With other applications, one single action on the user level causes hundreds or even thousands of proof obligations to be processed (e.g., in software reuse where one specification must be checked against an entire library of component specifications).

The number of proof tasks per session is also a major factor in determining which resources to allocate for each proof task. For applications which are interactive and where the user is waiting for results ("results-while-u-wait"), only a short (wall-clock) time can be spent for processing proof tasks. If, however, during this time many proof obligations must be processed, specific techniques like preprocessing, filtering and parallel execution must be considered. These techniques will be discussed in detail in Chapters 5 and 7.

Furthermore, the *total* number of processed proof obligations during the entire life cycle of the application system is of interest. Only from results of processing many proof tasks, statistical data can be obtained or domain-specific benchmark data bases can be developed. These data are important for tuning the theorem provers and for the (automatic) development of heuristics to increase the application's performance.

4.1.2 Frequency

The *frequency* of proof tasks is strongly related to the absolute number of proof tasks as described above. It reflects the number of proof tasks per time unit which are generated by the application and which are expected to be

processed for smooth operation. In contrast to batch systems which may run overnight (e.g., generation of test data) and may process in the order of one proof task per hour, a more interactive system must handle the given proof tasks in a much shorter time to be acceptable for the user. For this kind of systems (e.g., AMPHION, NORA/HAMMR, ILF), answer times will be limited to approximately one minute. Typical values for frequencies are

- $\approx 50/min$ for automated online verification systems (e.g., verification of down-loadable code),
- $\approx 1/min$ for interactive systems, and
- $0.1/min - 0.01/min$ for non-interactive (e.g., batch-like) verification systems, or systems to systematically generate test cases, etc.

For comparison, authors of case studies with interactive theorem provers or manual verification systems report times to prove major proof obligations of up to three month [Schellhorn and Arendt, 1998; Havelund and Shankar, 1996].

4.1.3 Size and Syntactic Richness

An important criterion for a proof task is its size and syntactic structure. Its size can be measured as the number of tokens (i.e., the number of symbols and logical operators). If a formula already consists of a set of clauses, the number of literals can be used as a measure for the size of the proof task instead. The syntactic structure can be characterized by a number of different parameters.

Complexity of the Propositional Skeleton. For formulas in clausal normal form, we have the distribution of clause lengths (i.e., number of literals per clause), and the number of non-Horn clauses. The latter figure is of interest, because for many theorem provers, non-Horn clauses require specific handling and can substantially increase the size of the search space (induced e.g., by the need for case splitting or by additional inference rules, e.g., the model elimination reduction step). For other formulas, the kind, number, and nesting of logical connectives is of interest. For example, proof tasks with many biconditionals "$\leftrightarrow$" are rather complex, because they can lead to large sets of clauses (see Section 3.2.2). Formulas which have many nested logical operators also tend to increase the size and complexity of a formula.

Complexity of Terms and Syntactic Richness. The number of different symbols and the size and nesting structure of terms in a proof task are in general an important characteristic for it. Proof tasks with many different symbols often contain internal structures (e.g., hierarchies of defined operators, clusters) which can be helpful for the automatic processing of the proof obligation, e.g., for preselection of axioms, or for imposing orders on the terms. Typical knowledge-base applications often generate proof tasks with such a characteristic.

On the other hand, proof obligations with few different symbols tend to be more "homogeneous", but it might be more difficult to extract information from such a formula. Many classical theorem proving examples, as e.g., can be found in the TPTP problem library [Sutcliffe *et al.*, 1994], are of this kind. However, the syntactic richness gives no hint how difficult it is to find a proof as we will see below.

Arity of Function Symbols. The higher the arity of a function symbol, the more possibilities for instantiating the arguments exist. Therefore, function symbols with many parameters are in general quite hard to process. In most cases, function symbols of high arity come in when a formula is converted into clausal normal form. There, all quantifiers are removed by Skolemization (Section 3.2.2). Existentially quantified variables are replaced by new constants or function symbols, called Skolem functions. The arity of a Skolem function depends on the number of universal quantifiers which have a scope surrounding the existential quantifier.

Example 4.1.1. A first-order formula

$$\forall X \forall Y \exists Z \cdot p(X, Y, Z)$$

is converted to the literal $p(X, Y, f(Y, Z))$ with a new Skolem function symbol f with arity 2. On the other hand,

$$\exists X \forall Y \forall Z \cdot p(X, Y, Z)$$

is converted to $p(a, Y)$ where a is a Skolem constant (or Skolem function of arity 0).

Therefore, it is important to reduce the number of outer universal quantifiers during the conversion as much as possible. Such methods are described, for instance, in [Loveland, 1978], or [Boy de la Tour, 1992] and should be part of all transformation systems. One method is illustrated by the following example.

Example 4.1.2. With two simple rules (written as inference rules), quantifiers can be distributed over logical connectors. Thus, the scope of the quantifiers can be reduced which helps to avoid long Skolem functions during subsequent Skolemization.

$$\frac{\forall X \cdot (p(X) \wedge q(X))}{\forall X_1 \cdot p(X_1) \wedge \forall X_2 \cdot q(X_2))} \qquad \frac{\exists X \cdot (p(X) \vee q(X))}{\exists X_1 \cdot p(X_1) \vee \exists X_2 \cdot q(X_2))}$$

Let us now consider the formula

$$\forall X \forall Y \cdot (\exists Z \cdot p(X, Z) \wedge q(X, Y))$$

When this formula is translated into clausal normal form using the standard Skolemization, Z will be replaced by a Skolem function of arity 2, e.g., $f(X, Y)$. Using the first of the two above transformation rules, we obtain:

$$(\forall X \forall Y \exists Z \cdot p(X, Z)) \wedge (\forall X \forall Y \exists Z \cdot q(X, Y))$$

After removing the empty quantifier in the second conjunction, we finally get: $p(X, f'(X)) \wedge q(X, Y)$ where the Skolem function f' is only of arity 1.

Nesting of Function Symbols. Much information can be extracted from the way, how symbols are nested in the terms indexnesting of function symbols of the formula: when there is no nesting of symbols, the formula belongs to data logic which is simple to process and even decidable. Information about nested function symbols (in particular, if additional semantic information is available, cf. Section 4.3.5) can be very helpful for controlling the prover. For instance, certain function symbols must not be nested recursively. E.g., $\mathsf{hd}(\mathsf{hd}(X))$ is not a valid expression, if hd denotes the head of a list. On the contrary, $\mathsf{tail}(\mathsf{tail}(X))$ is perfectly right. Such pieces of information can be very helpful to rule out certain terms and to prune the search space considerably. E.g., in SETHEO, the first case could be forbidden using syntactic constraints: $\mathsf{hd}(Y) : [Y] \neq [\mathsf{hd}(\#X)]$ means that variable Y must never be instantiated with a term of the form $\mathsf{hd}(\ldots)$[1]. Furthermore, restrictions on the depth of the terms (= maximal nesting of symbols) can be applied to reduce the search space. As we will see in Section 7.4, sorts can be used to describe restrictions on nesting of function symbols in a formal and efficient way.

Summary. As a rule of thumb, large formulas with a rich structure are difficult to process by an ATP. However, as the following examples will show size and syntactic richness of a proof task is not always directly related to the difficulty of finding a proof for it.

Example 4.1.3. The following formula encodes a proof problem from mathematical logic. The predicate $\mathit{theorem}(X)$ is TRUE, if and only if X is a valid propositional formula; $i(X, Y)$ denotes the implication $(X \rightarrow Y)$[2]. Given one single axiom (4th Łukasiewicz axiom, L4) and the rule of modus ponens (MP, also called condensed detachment), the proof obligation is to show that a typical tautology, e.g., $a \rightarrow a$ (LC-1) can be deduced from this axiom. Such Hilbert-style calculi with minimal numbers of axioms have been investigated by Łukasiewicz [Łukasiewicz, 1970]; hence the name of the formulas. A transformation into first-order logic (T-encoding) yields the following formula:

$$\forall X, Y \cdot \mathit{theorem}(i(X, Y)) \wedge \mathit{theorem}(X) \rightarrow \mathit{theorem}(Y) \quad \text{(MP)}$$
$$\forall P, Q, R, S, T \cdot$$
$$\mathit{theorem}(i(i(i(P, Q), i(R, S)), i(T, i(i(S, P), i(R, P))))) \quad \text{(L4)}$$
$$\forall X \cdot \mathit{theorem}(i(X, X)) \quad \text{(LC-1)}$$

[1] However, this does not guarantee soundness in general, because SETHEO's constraints are turned off automatically in certain cases. For more details see [Schumann and Ibens, 1998].

[2] This technique to encode other logics in first-order logic (T-Encoding) is described in detail in Section 7.2.2.

These formulas (our formula is just an example of a whole class of formulas) are, despite their small size and simple structure, very hard to solve. Even powerful automated provers like OTTER or SETHEO need up to several minutes of run-time to find a (very long) proof. There are several research papers on using automated theorem provers for these problems, e.g., [Wos and McCune, 1988; Pfenning, 1988; Veroff and McCune, 2000].

Example 4.1.4. Consider the following formula:

$$(\mathcal{F} \wedge p) \rightarrow p$$

where p is a single literal and $\mathcal{F}$ a large formula (e.g., a long conjunction of axioms). Although the above formula can get arbitrarily large, the proof for it is trivial. Nevertheless, such proof tasks often occur in real-world applications. A typical example is the retrieval of a component (see Section 6.3) where the query specification and the specification of the library module is identical. In that case, $\mathcal{F}$ would be a (large) set of axioms and hypotheses. Another example would be the verification of a refinement step which has no influence on the component under consideration.

In general, however, the syntactic structure of a proof obligation will give valuable hints for the requirements imposed upon the automated deduction system. Typical implications are:

- very large formulas need extensive preprocessing and simplification (Section 7.3) in order to remove redundant and unnecessary parts. Otherwise, the resources required by the ATP to process the formula (often even before the search itself can start) would surpass available limits.
- small formulas are either rather trivial or require logically deep and complex reasoning (e.g., complex transformation of proofs). Their handling thus requires specific techniques and strategies (Section 7.6).
- in many applications, large formulas with many different symbols contain "internal structure" which might be helpful to control the prover. This is in particular true, if additional (semantic) information is available. One technique for selection of axioms during preprocessing is described in Section 5.7.

4.2 Logic-related Characteristics

4.2.1 Logic and Language

In general, the language and logic of the application ("source logic") is not first-order predicate logic. For example, an application might use VDM/SL [Dawes, 1991] as its specification language. The logic underlying VDM is a three-valued logic [Barringer *et al.*, 1984]. Therefore, the proof tasks must be

translated from the source logic (and language) into a logic suitable for the automated theorem prover.

This translation can be made already within the application system. In that case, the proof tasks that are handed to the ATP are already in first-order logic. Otherwise, the interface between the application system and the prover is in charge of transforming the proof obligations (see Section 7.2 for details). Such a transformation can be very simple or extremely complex, depending on the "distance" between source and target logic. In any case, the source logic and its specific characteristics (e.g., restrictions specific to the given application) are valuable sources of information which facilitate the application of an automated theorem prover. Thus, first-order formulas generated by translating formulas from some other logic often exhibit certain structural properties which can be used for pruning the search space. Translation can also impose additional requirements upon the automated theorem prover by the need to handle specific extensions to first-order logic. Two common extensions, equational logic and sorted logic will be discussed in more detail below.

4.2.2 Simplification

Often a proof task contains redundant parts. This is particular the case, if the proof tasks are automatically generated by the application system. The transformation from source to target logic can introduce formulas which then turn out to be trivially TRUE or FALSE. Also arithmetic expressions can contain redundancies (e.g., $Y = 1 \times (Z + 0) + 0$).

Example 4.2.1. Simplification should remove logically trivial parts of a formula, e.g., in $p \wedge \text{TRUE}$ or $p \wedge (q \vee \neg q)$.

For instance, the following formula which is the base-case of a simple induction over lists

$$X = [\,] \rightarrow ((X \neq [\,] \rightarrow \mathcal{Q}) \vee (X = [\,] \rightarrow \mathcal{R}))$$

with arbitrary formulas $\mathcal{Q}$ and $\mathcal{R}$, can be simplified to $X = [\,] \rightarrow \mathcal{R}$.

If these redundant parts remain in a proof task, processing time can increase substantially, because current automated provers do not simplify their input formulas. Hence, simplification needs to be done at the application system or the prover's interface.

4.2.3 Sorts and Types

Most specification methods are based on a typed or sorted logic. Inspired from programming languages where data-types facilitate the development of software, (abstract) data-types and other sort information have been incorporated in most formal methods and specification languages. Use of sorts

and types can range from simple sort specifications (e.g., X : nat), to most complex types, e.g., P : correct_program which denotes that P is a "correct program"[3].

Sorts and types can facilitate the search for proofs, or can be cause for new proof tasks. A typical example for the latter case would be our expression P : correct_program. Type-checking this expression implies the full verification of P. The interactive theorem prover PVS [Crow *et al.*, 1995] works with such an approach. Due to its complexity, handling of such typed proof tasks are well beyond the capabilities of today's first-order ATPs and must be handled by the appropriate (interactive) tools.

When looking at the proof task under consideration, it is thus an important characteristic whether this proof task contains types and sorts or not. Proof tasks based on a sorted or typed logic usually require specific handling. It is always possible to express sort information as a unary predicate. Then an atom $p(t : \mathsf{s})$ with term t of sort s and predicate symbol p would become $p(t) \wedge \mathit{is\_of\_sort}(t, \mathsf{s})$. However, such a transformation in general creates extremely large search spaces and is in practice infeasible. There are efficient ways of handling sorted formulas which will be discussed Section 7.4. However, they only work for restricted kinds of sorts. Therefore, it is important to know which kinds of sorts occur in the proof tasks. A typical classification with respect to their handling with automated theorem provers is (see, e.g., [Semle, 1989; Bürckert, 1985; Dahn, 1996; Bahlke and Snelting, 1985]):

- are the sorts *static*, or does the sort information depend on variable substitution, e.g., $X > 0 \rightarrow X$: nat ? Dynamic sorts in general are more difficult to handle than static ones.
- does polymorphism, e.g., overloading of function symbols, occur in the formulas?

  Example 4.2.2. A function symbol is overloaded when there are different semantic interpretations for it, depending on the data types of the arguments. In programming languages and most specification languages, arithmetic operators are overloaded. For example $+ : \mathsf{nat} \times \mathsf{nat} \longrightarrow \mathsf{nat}$ evaluates the sum of two natural numbers and returns a natural number whereas $+ : \mathsf{float} \times \mathsf{float} \longrightarrow \mathsf{float}$ does this for float-point numbers. Thus, when used, axiomatization and processing (e.g., implicit data-type conversion) has to make sure that the correct operation is used. For example, $X +_{\mathsf{nat}} Y > 0$ is only valid for variables carrying natural numbers.

  Except for restricted cases (which in practical cases are well sufficient), handling of polymorphic types can be extremely complex.
- what is the structure of the sort hierarchy? Quite often, sorts can be arranged in a hierarchical manner under the relation "is sub-sort of" ($\sqsubseteq$). For

[3] For a definition of sorted logic and its properties and further details see Section 5.6.

example, nat $\sqsubseteq$ int $\sqsubseteq$ real. If such a hierarchy forms a tree or directed acyclic graph (DAG), sorts can be processed more efficiently (Section 7.4.3). In case, the hierarchy is flat, the problem is called *many-sorted.*

4.2.4 Equality and Other Theories

For the efficient processing of a proof task, it is important to know, whether it contains equations (and/or inequalities), or formulas from specific algebraic theories. Many automated theorem provers use special-purpose methods for dealing with such proof tasks. In particular, for handling equations, powerful methods have been developed. Thus, in many cases, proof tasks can be processed considerably more efficient by provers with such methods than without them. In the latter case, all properties of equality (or other theories) must be added as axioms to the formula (Section 3.3.2). Thereby, the size of the search space increases tremendously. For formulas consisting solely of equations (and/or inequalities), special-purpose equational theorem provers (e.g., Discount [Denzinger and Pitz, 1992], or E [Schulz, 1999]) have been developed and are applied with great success.

Hence, an important characteristic for a proof task is the amount of equations (conditional or unconditional) within the entire formula. This figure has a strong influence on the choice of an appropriate theorem prover and is important for tuning the system.

Of similar importance is *arithmetic.* Proof tasks from all domains in software engineering usually contain arithmetic expressions. Quite often, they are based on integer arithmetic and use only quite simple operations, e.g., increment/decrement or addition/subtraction. Typical examples would be counters within loops or array access. Other proof obligations, however, might also contain complex arithmetic expressions over real numbers. A classical example would be the verification of the well-known steam-boiler problem [Abrial *et al.*, 1996], a hybrid control system. Here expressions concerning temperature, pressure and physical constants occur.

In contrast to programming languages, where arithmetic expressions can be evaluated, in logic, arithmetic expressions can give rise to complex equations which must be solved, e.g., $6 = X \times 2$. Because solving arithmetic equations is a very complex task, it is important to have a look at which operations actually occur in the proof tasks of the application. For certain subsets, efficient methods to solve equations can be incorporated into automated inference systems (e.g., Pressburger Arithmetic [Pressburger, 1929]). Typical restrictions are:

- arithmetic ranging over natural numbers or finite subranges,
- the only operation is $+1$ or -1 (increment, decrement),
- only addition and subtraction occur.

The more restricted the type of arithmetic expressions is that occurs, the better they can be handled by the automated prover. More complex equations

sometimes can be handled by a symbolic algebra system, e.g., Mathematica [Wolfram, 1991]. Such systems, however, are usually not guaranteed to be sound, because they perform simplifications without checking for additional assumptions , (e.g., division by zero). For a detailed discussion on incorporating symbolic algebra systems into automated theorem provers see [Armando and Jebelean, 1999].

4.3 Validity and Related Characteristics

4.3.1 Theorems vs. Non-theorems

The ratio between theorems (= valid formulas) and non-theorems is an important criterion for any application. Whereas a successful verification should consist of provable theorems only, attempts to verify specifications with bugs lead to non-provable proof obligations, or *non-theorems*. In other domains, e.g., logic-based component retrieval, most of the proof tasks are in fact non-provable, because they correspond to non-matching modules in the database (Section 6.3).

Most automated theorem provers can detect theorems, but they are in general not able to detect even very simple non-theorems as such. This is due to the calculus/proof procedure and the general undecidability of FOL. Trying to prove $\neg\mathcal{F}$ for a given proof task $\mathcal{F}$ does not help much, because in most cases neither $\mathcal{F}$ nor $\neg\mathcal{F}$ are universally true.

In cases, where many non-theorems are expected, the application of classical automated theorem provers might fail. Here, other techniques have to be used which will be discussed in Section 5.8 and Section 7.5. Furthermore, the application system might expect feedback on *why* the conjecture could not be shown.

4.3.2 Complexity

The dynamic complexity of a proof task is much more difficult to describe and assess than a static measure like size or syntactic richness (Section 4.1.3). With "complexity" we mean an estimation, how "hard" it is to solve a given proof task, not complexity of the calculus or proof procedure in general. The dynamic complexity concerns the average size of the proof, e.g., measured in number of inference steps in the final proof and the resources consumed to find the proof. Since these figures heavily depend on the calculus, the proof procedure and its implementation, only a thumb-rule value can be given: "easy" for a proof with few, simple inference steps which can be found with few resources, and "hard" otherwise.

As demonstrated before (Section 4.1.3), the static complexity usually gives no clear indication for the dynamic complexity. Hence, it is important to

consider both measures together, in particular, since current technology automated theorem provers are very sensitive with respect to these issues. So, many theorem provers (e.g., SETHEO) are overwhelmed by large formulas ($> 100 - 300$ clauses), regardless of how complex they are. Due to historic reasons, most automated provers have been developed to handle small formulas (i.e., mostly formulas which only contain exactly the clauses needed for the proof), which have comparatively complex and deep proofs.

When we relate the static and dynamic complexity of proof obligations, we obtain the situation depicted in Figure 4.2. Both dimensions of complexity can be subdivided further: for static complexity, we can distinguish between "*proving in the small*" and "*proving in the large*". Whereas proving in the small deals with formulas of small size and little syntactic richness, large formulas (with many axioms and hypotheses) belong to the second category. A distinction for the dynamic measure can be made between *shallow* (i.e., easy) proofs and *deep* (or complex) proofs.

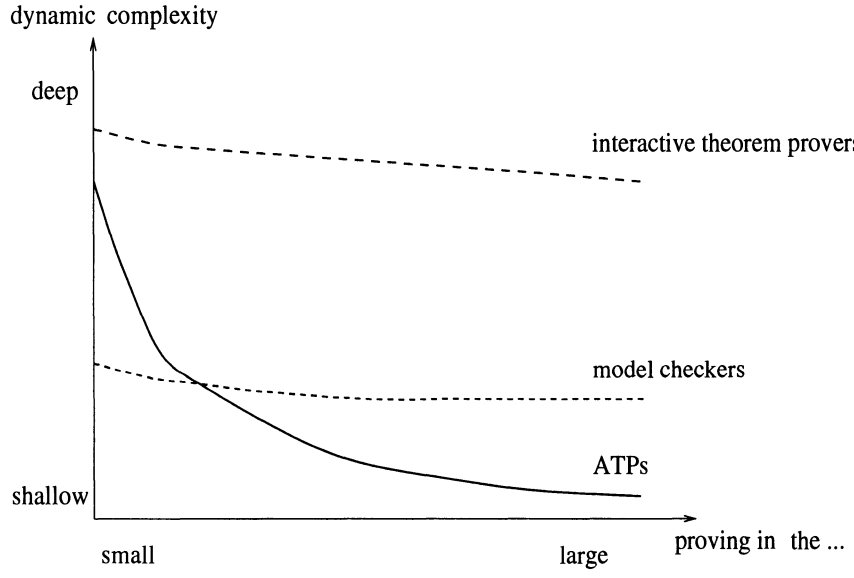

Fig. 4.2. Applicability of current theorem proving technology w.r.t. the dimensions of dynamic complexity ("deep/shallow proofs") and "proving in the small/large"

The situation for current technology interactive theorem provers (ITP), automated theorem provers (ATP), and Model Checkers (MC) with respect to both dimensions is depicted in Figure 4.2. Interactive provers are capable of handling complex proofs, relatively regardless of the proof obligation's size. Finding the right parts of a formula for the inference steps is left to user.

Model checkers are also capable of handling large formulas. They are, however, restricted to much shallower proofs, mainly because the expressiveness of their input language (propositional logic) is rather limited. Automated provers, finally, are very sensitive with respect to the size and structure of the formulas. For small formulas, ATPs are capable of finding even rather complex proofs, but for larger formulas, they often fail. This is why additional techniques of reducing the size of the formula (e.g., simplification or preselection of axioms) is so important.

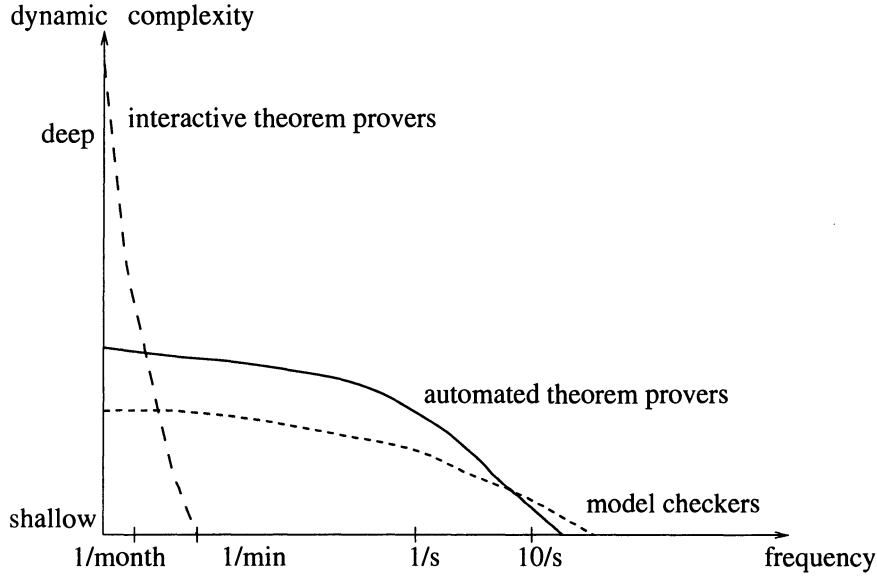

Fig. 4.3. Applicability of current theorem proving technology w.r.t. the dimensions of dynamic complexity ("deep/shallow proofs") and "maximal frequency"

Figure 4.3 shows the relation between dynamic complexity and the maximal frequency (Section 4.1.2) with which the proof tasks can be processed. Interactive provers can — due to the required user interactions — handle only very low frequencies (up to about 1/min). Obviously, the more complex a proof, the longer it takes to construct it with such a system. Automated theorem provers and model checkers can accept proof obligations with much higher frequencies. Here, more complex proofs can be found with increasing resources (run-time). Because of the search-space explosion, increased run-time does unfortunately not mean proportionally more solved proof tasks.

4.3.3 Expected Answer

When a proof task has been processed by the inference system, the application system expects an *answer*. In the simplest case, a boolean value is sufficient. Of course, due to the always imposed limits, it can be the case that a time-out occurs. Time-outs and other errors must the be mapped to a logical value (in general FALSE), or, better, directly returned to the application system. Hence, we would get (TRUE/FALSE/TIMEOUT/ERROR). Additional pieces of information which the application system might expect are:

- statistical data (run-time, number of inferences, consumed memory, etc).
- used formulas, i.e., which parts of the formula (e.g., which hypotheses) are actually required for the proof.
- (disjunctive) answer substitution. If a theorem contains existentially quantified variables (e.g., $\exists X \cdot p(X)$), a substitution $X \backslash t$, indicating which X exists, can be of interest. In the general case, such an answer substitution is a disjunctive set of values, e.g., $X = a \vee X = b \vee X = c$. The calculation of answer substitutions is of importance for many applications, in particular knowledge-based applications, where information is extracted from a data base.
- variable substitutions: an extension of the above information is the list of variable substitutions for *all* variables throughout the proof. For example, in AMPHION, this information is used to synthesize the required output program from the proof (see Section 4.7.1 for details). Whereas this information is readily available for tableau-based provers (e.g., SETHEO, METEOR, Protein), resolution-style provers (e.g., OTTER, SNARK) need additional techniques and data structures to obtain this information.
- a proof: of course, the entire information discussed above is contained in a full proof which the inference system might return. Important issues about the expectations of the application system are: is a proof in the ATP's logic sufficient, or must the proof be transformed into the application system's source logic? Can the proof be returned as a data structure, or must the proof be human-readable and understandable? (cf. Section 4.4.5)
- counterexample(s): if a theorem cannot be proven, information about this failure might be expected. One answer could be to return a counterexample (i.e., variable substitution), under which the theorem is FALSE.

4.3.4 What Is the Answer Worth?

Besides the information provided with the answer, as discussed above, an important issue is: "what is the answer worth?" If the application system always expects a correct answer, the inference system must be sound and complete. Soundness means that always correct proofs are returned, and completeness means that the prover eventually finds a proof for a valid proof task. In practical applications, however, certain cut-backs with respect to these

requirements must be made. For completeness, this is due to the theoretical semidecidability of first-order logic and the extreme high complexity of proof procedures.

Hence, it is important to realize what an application might need, given certain resource limits. These expectations can reach from "the answer must *always* be correct", or "no answer is better than a wrong one" to "the obvious theorems should be proven, and the obvious non-theorems marked as such". In particular, in non-critical applications or during development phases it makes sense to consider acceptable degradations of soundness and completeness. This can influence the choice of inference system and its control. Furthermore, for applications with limited resources (e.g., time limits, limits in computational power), such restrictions can be vital for using an ATP at all.

Of course, such a restriction must be meaningful and acceptable. For example, PROLOG's pure depth-first search would be not acceptable for Horn clause reasoning, because even obvious and trivial proofs would be lost. Other techniques, like abstraction and limitation to small domains (e.g., check a conjecture with 0,1,2 instead of all natural numbers) might be helpful in certain cases. In Chapter 6 and Section 7.5, these issues will be discussed in more detail.

Example 4.3.1. The following PROLOG program together with its query (starting with ?-) is equivalent to a valid Horn-formula. However, a PROLOG system will not find a proof, because it will recursively try the second clause. This unrestricted depth-first search leads into an endless loop.

```
?-  married(john,X).
    married(X,Y) :- married(Y,X).
    married(john,mary).
```

4.3.5 Semantic Information

Theorem proving, by definition, is an exclusively syntactic processing of terms. For the logic calculus and proof procedure, the semantic interpretation of function symbols, predicate symbols and constants is never used. For instance, if we use $add(X, Y)$, meaning $X + Y$, all properties defining the addition of two numbers must be given as additional axioms to the formula.

In practice, such knowledge about the semantics of the symbols can be of great help. It can be used, e.g., to select efficient methods for handling these proof obligations. So, for common theories, like equality or basic arithmetic, specific and efficient techniques have been implemented (see previous Section 4.2.4). Although it is possible to *recognize* the equality predicate, even if its name[4] isn't *equal* or $=$, much run-time overhead can be saved, if

[4] In order to recognize a theory, all axioms must be scanned to find those with the appropriate properties. In case of equality, we have to search for reflexivity, symmetry, transitivity, and substitution axioms.

it is known beforehand that *equal* really means equality. Similarly, semantic information can be used to set up a strong sort system (see Section 4.2.3 and 5.6).

4.4 System-related Characteristics

4.4.1 Flow of Data and Control

This item concerns the architecture of the system, as seen from the point of view of the application. A general description highlights the major components and their type and shows the flow of data and control. This description will also locate the place and the tasks of the inference component(s) (e.g., automated provers).

If more than one inference component exists (or is envisioned), a variety of different architectures are possible as depicted in Figure 4.4. The provers may work independent from each other (i.e., they work on different proof tasks, or they may compete (Fig. 4.4a). For example, in the ILF system [Dahn and Wolf, 1996], different theorem provers (SETHEO, OTTER, SPASS, DISCOUNT) compete to solve the same proof task within a given time limit. The prover which is the first to find a proof, wins. If more than one inference component works on a problem by splitting it up into subproblems, we may find an architecture as depicted in Fig 4.4b. Such an architecture can be expected when an automated theorem prover is supported by decision procedures or a symbolic algebra system, like Mathematica [Wolfram, 1991].

On the other hand, provers may be placed in a pipeline, working as filters (Fig. 4.4c). For example, in NORA/HAMMR (Section 6.3), components of increasing deductive power rule out as many non-matching components as possible. Finally, subcomponents of the inference system can cooperate, e.g., by sharing intermediate results (Fig. 4.4d). Such a system is the parallel theorem prover CPTHEO [Fuchs and Wolf, 1998].

4.4.2 Relation Between Proof Tasks

If during one session, more than one proof task is generated by the application system, it is important to know whether the proof tasks are related to each other, or not. If they are related, information could be exchanged between the proof obligations, concerning heuristics, useful lemmata, proof plans, etc.

4.4.3 Automatic vs. Manual Generation

Are the proof obligations (or parts thereof) generated automatically by the application system, or can parts of the proof tasks be entered manually? Furthermore, are parts of the proof tasks fixed (e.g., a domain theory which is

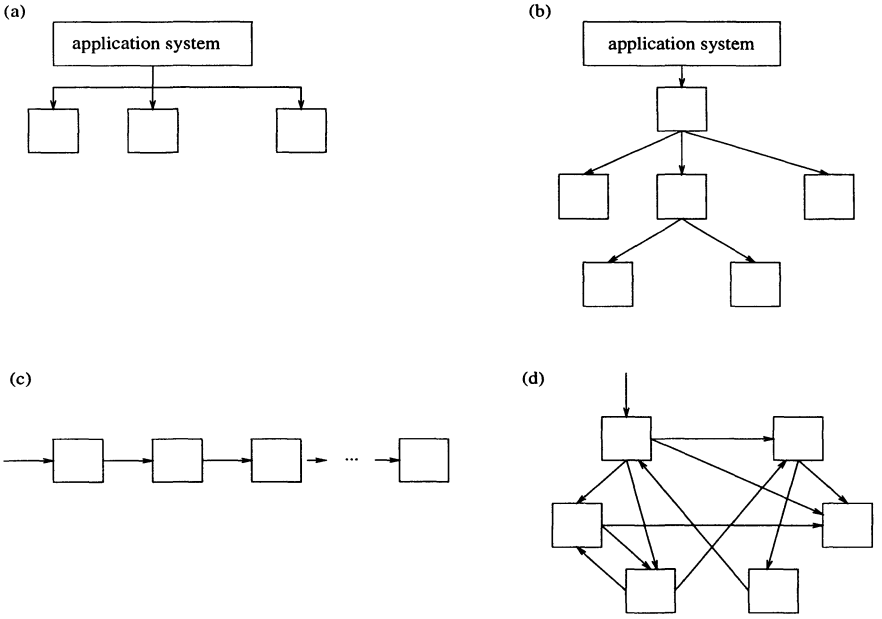

Fig. 4.4. Ways to combine different provers or deductive components with the application system

the same for all proof tasks from that domain)? The answer to these questions is of importance for the entire system since many design decisions depend thereupon.

If the entire proof task is generated automatically, it is always assured that the proof tasks are syntactically and semantically correct. Otherwise, the prover (or better, the application system) must check for syntactic correctness and must react accordingly. This means, that, e.g., the sort of each function symbol or constant must be checked in advance[5].

Specific ways of entering data (e.g., input of specifications using a graphical user interface as it is done in AMPHION or AUTOFOCUS) restricts the ways of generating source (and target) specifications. Thus, proof tasks are much more homogeneous and of "one style". This facilitates fine-tuning of the automated prover and allows to develop better strategies for automatic processing of proof tasks.

Example 4.4.1. Application systems which use automated theorem provers can be classified with respect to their generation of proof obligations into:

[5] This simple technique allows to catch many misspellings of symbols. If, e.g., cond is typed instead of cons, the system can immediately tell that there is no symbol cond of sort item × list → list. Otherwise, the prover would simply accept cond as a new symbol and would not find a proof.

fully automatic with restricted input. Examples for formal methods tools which have a restricted way of (usually graphically) entered inputs are AMPHION (see Section 4.7.1 below), AUTOFOCUS [URL Autofocus, 2000], SADT, or State-mate (based on Harel's statecharts [Harel, 1987]). Here the logic specification is generated out of the given figures. Restricted expressiveness (e.g., only certain nesting level of objects, or fixed number of slots in an object) can largely facilitate automatic processing.

semi-automatic. Verification systems like KIV [Reif *et al.*, 1998], KIDS [Smith, 1990] or tools for a specific specification language result in proof tasks with many common characteristics which are imposed by the input language and its translation (e.g., structure of the formula, representation of operators, or restricted sorts).

arbitrary input. Interactive theorem provers usually allow to enter arbitrarily structured formulas.

Example 4.4.2. In the system NORA/HAMMR (see Section 6.3 for details), only the domain-specific axioms are fixed. All specifications can be entered in a (pretty) arbitrary way, albeit we have a uniform translation from VDM/SL into first-order logic. Checking that the specifications are equivalent then can lead to rather complex proof obligations. As an example, consider the following three specifications in Table 4.1. They define a function which returns the first element of a list (the head) as a singleton list. For example, $head_1([a, b]) = [a]$.

SPEC: $\text{head}_1(l)$: list m : list PRE: $l \neq []$ POST: $m = [hd\ l]$;
SPEC: $\text{head}_2(l)$: list m : list PRE: $l \neq []$ POST: $l = m \hat{}\, tl\ l$;
SPEC: $\text{head}_3(l)$: list m : list PRE: $l \neq []$ POST: $\exists i$: item, $\exists l_1$: list & $l = [i] \hat{}\, l_1 \wedge m = [i]$;

Table 4.1. Three specifications of a function which returns the first element of a list (the head) as a singleton list

The run-times needed to solve the proof tasks (in their translation as used in NORA/HAMMR) are shown in Table 4.2. They have been obtained by SETHEO on a SUN ultra-sparc. Here, $\text{head}_i \rightarrow \text{head}_j$ means that the proof task was generated by connecting the appropriate postconditions via implication. The almost trivial task of showing this implication for identical specifications ($\text{head}_i \rightarrow \text{head}_i$) could be solved by SETHEO very quickly.

proof task	run-time[s]
$head_1 \rightarrow head_2$	22.1
$head_2 \rightarrow head_1$	6.8
$head_2 \rightarrow head_3$	11.6
$head_3 \rightarrow head_2$	11.8
$head_3 \rightarrow head_1$	2.2
$head_1 \rightarrow head_3$	2.5
$head_i \rightarrow head_i$	< 0.1

Table 4.2. Run-time for proof tasks showing the equivalence of different specifications of the same function

4.4.4 Guidance and Resource Limits

Additional information provided with the proof obligations can be helpful to *guide* the automated theorem prover during the search of a proof. Typical examples are:

- "such a proof task is often shown by induction over n": this piece of information can be a tactic (as it is known from interactive theorem provers), or can be a general heuristics. Whereas in the first case, one would normally expect to be this an integral part of the proof task, the second case could be used to control the strategies of the prover(s).
  A generalization of this would be to have *proof plans* [Bundy, 1996] for the entire proof task or parts thereof. In the extreme case, the search for a proof in such a proof obligation would degenerate to a check (or replay) of the proof.
- known resource limits (e.g., run-time). If the maximal run-time is known a priori (in contrast to "run until I stop you"), automated provers can use much better ways of distributing the resources for good search strategies. An example is the prover Gandalf [Tammet, 1997] which takes the entire available run-time and distributes it in a weighted way among its different strategies.

4.4.5 Human-understandable Proof Tasks

Does the user see the proof tasks, their processing and the answers of the automated prover, or are they hidden from the outside? In the second case — which should be preferred where possible — the user has no need to be able to read or understand the low-level details of proof tasks (e.g., target logic, proof calculus, details of the ATP). Then, we can have an arbitrary representation of the proof task (e.g., names of symbols). This means that a term `p_01(f_027(a_BhS30,X_99))` is acceptable. If, however, the user (or even the "application administrator"[6]) has to have access to the process-

[6] Some applications can require an application administrator who is in charge of keeping the tools in a consistent state, updating the knowledge-base (e.g., for

ing of proof tasks (e.g., for debugging, ATP tuning), he/she must be able to understand the proof obligations. This at least requires problem-oriented (readable) symbols and in some cases even human-readable ATP proofs (see also Section 7.7).

4.5 Discussion

For the evaluation of a particular system or application, the actual experience made with that system is of great importance. If, e.g., a system architecture and the implementation works great, but the answer time (for the user) between interactions is in the range of hours even on fast hardware, the system is not usable in practice. Furthermore, it is important to know, how much training a user must have in order to operate the system. Reports on case studies carried out with the formal method or tool under consideration largely facilitates the assessment of the proof tasks' characteristics and to set up requirements, the targeted automated prover must fulfill.

4.6 Summary and Evaluation Form

Before attempting any application of an automated theorem prover, it is strongly advisable to have a close look at the proof tasks to be expected. As we have seen, syntactic and semantic criteria of the proof tasks strongly influence the prover's behavior. Furthermore, we can get a quick estimate whether an automated theorem prover is suited for the given application or not. For example, proof tasks which are too large cannot be handled by a state-of-the-art automated theorem prover; simple proof obligations for which an answer is expected almost immediately are also critical for an automated prover (see Figures 4.2 and 4.3). The most important criteria of a proof task can be represented in a compact way shown in Table 4.3. Qualitative values (like low, medium, high) are usually sufficient.

- *Deep/shallow*: Proof tasks can be rather simple or extremely difficult. As discussed in Section 4.3.2 above, shallow proof tasks are those for which the proofs are short (e.g., up to approx. 10-20 model elimination or resolution inference steps) or straightforward. Usually such proofs can be found relatively easy. Deep proofs, on the other hand are lengthy and complex in their internal structure.
- *Size & richness*: Size of the formula and its syntactical richness give indications about the difficulty to process a formula. For example (see Section 4.1.3 for details), large proof tasks generally need longer preprocessing

knowledge-based synthesis or reuse), or tuning the ATP. He/she has to have access to low level details and must of course have some knowledge about automated theorem proving.

category	value		
deep/shallow	Shallow	Medium	Deep
size & richness	Small	Medium	Large
answer-time	Short	Medium	Long
distance	Short	Medium	Large
extensions	Y/N	which?	
validity	% non-theorems		
answer	TRUE/FALSE	proof	other
semantic info	Y	some	N

Table 4.3. Compact evaluation table for proof tasks of an application

time than small ones (even making the application of an automated prover impossible if short answer times are required). Formulas with ample syntactic richness often exhibit internal structure which can make theorem proving easier (e.g., allow for good pre-selection of axioms).

- *Answer time* is the maximal time the automated theorem prover is given to solve a proof task. Run-times below around 1 second are certainly short (note that transformation of the proof task and preprocessing can take considerable time). Usual run-times in the "medium" category are several seconds to around one minute. Only few applications will allow run-times which are considerable longer than a few minutes.
- *Distance*: This values describes the effort necessary to convert and translate a given proof task into the representation of the prover. A proof task which is already in first-order logic and only needs syntactic transformation (e.g., infix into prefix conversion) exhibits a small distance between source and target logic. Medium or large distances may require complex translations (e.g., from a modal logic), induction, or splitting up into several individual proof obligations.
- *Extensions*: Often, a translation of a proof task can be accomplished into a first-order logic (FOL) with extensions more efficiently than into pure FOL. Typical examples are: sorted first-order logic (Section 4.2.3), FOL with equality and arithmetic (Section 4.2.4).
- *Validity*: Since non-valid proof tasks ("non-theorems") require specific techniques, it is important to know whether many proof tasks comprise non-theorems or not (Section 4.3.1).
- *Expected answer*: Of course, the application system expects some answer from the automated theorem prover. This can be a boolean value, an answer substitution, or a proof (machine-oriented or human-readable), as discussed in Sections 4.3.3 and 4.4.5.
- *Semantic information*: Any additional information provided with the proof tasks can be helpful for the automated theorem prover (adaptation, tuning, heuristics). Therefore, this issue should be noted carefully.

If more details are needed, a form as shown in Table 4.7 (p. 69) can be used. As an example, this form has been filled out with data from our case study on verification of authentication protocols in Section 6.1. As in the text above, the items are arranged according to the topics *syntax*, *logic*, *validity*, and *system-related* issues. Where appropriate, mean values and ranges (e.g., size of the proof tasks) are given. When filled out, this form contains enough information for a thorough analysis of the requirements, an automated theorem prover must fulfill for a successful application.

4.7 Examples

In this section, we look at three different application systems which use automated theorem provers to process (some of) their proof tasks: AMPHION is a system for automatic synthesis of FORTRAN programs, KIV an interactive verification system, and ILF an interactive theorem prover which is being applied to hardware and software verification and mathematics. These examples have been selected, because they have been designed for practical usability and they are being applied for real-world problems. All of them use automated theorem provers for first-order logic to process proof tasks. The description of the systems will highlight their system architecture, main features and characteristics of the proof tasks.

4.7.1 Amphion

The main application area of AMPHION [Pressburger and Lowry, 1995; Lowry *et al.*, 1994a; Lowry *et al.*, 1994b; Stickel *et al.*, 1994] is the automatic synthesis of astrodynamic FORTRAN programs out of a given subroutine library (NAIF). Specifications are entered in a graphical way as shown in Figure 4.5 (taken from [URL Amphion, 2000]). All bodies (here Jupiter, Sun, space-craft) and their relationship as well as the desired function (here to calculate the boresight angle between the Space-craft Galileo and the Sun) are entered using graphical elements. From this, an internal formal specification of the problem is automatically generated and processed by the automated prover SNARK. Its result corresponds to the sequence of library calls necessary to calculate the desired function. Finally, a post-processor converts this data structure into the desired FORTRAN program (Figure 4.6). This tool is widely used within NASA and has been extended to handle several other domains (e.g., fluid dynamics and state estimation). In this application, hiding the prover and its logic is important. Only then, this tool can be used by non-specialists in the area of theorem proving: "[...] users are able to develop their own specifications after only an hour's tutorial" ([Lowry *et al.*, 1994b], p. 49).

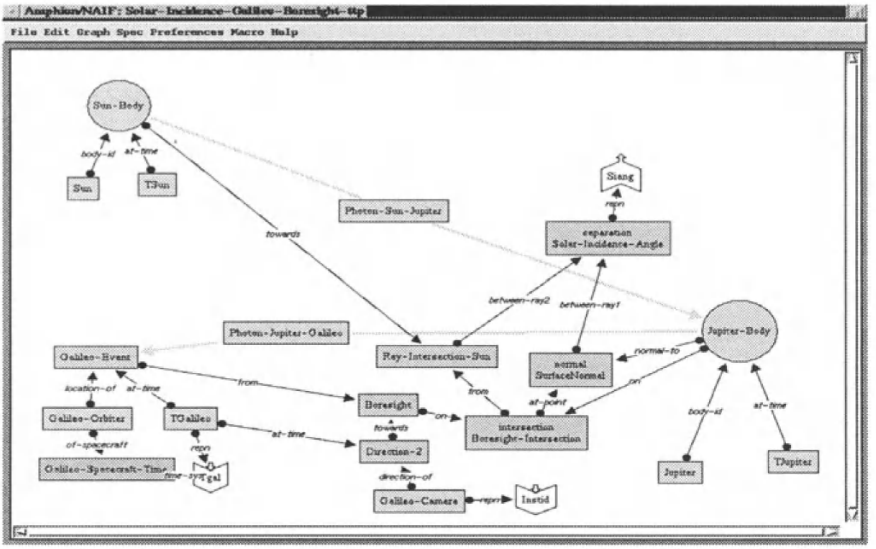

Fig. 4.5. Example of graphical input specification for AMPHION

System Architecture. AMPHION's architecture is shown in Figure 4.7. It consists of three basic components: the graphical user interface for constructing the specification, together with a specification checker which will assure that only correct specifications can be entered and processed (specification acquisition subsystem). The internal form of a specification is a first-order formula augmented with lambda terms (for an example see [Lowry *et al.*, 1994b]).

The automated theorem prover SNARK then takes this formula, together with a set of (domain-dependent) axioms and theorem-proving tactics and tries to find a constructive proof. The variable substitutions for the existentially quantified output variables of the problems form the desired applicative program which then is converted into the target language using techniques of program transformation (program synthesis subsystem). Axioms and tactics for SNARK as well as control information for the graphical user interface are extracted from the defined domain theory (domain-specific subsystem). This theory can be changed, if the system is to be applied to other domains. Therefore, only the developer/administrator has access to the formulas and the prover. The user communicates with the system only via the graphical user interface on the problem-oriented level.

The Theorem Prover SNARK. The prover built into AMPHION is SNARK [Stickel *et al.*, 1994; URL SNARK, 2000]. It is a bottom-up, resolution style prover for first-order logic. It features a variety of efficiently implemented resolution principles (like UR-, Hyper-resolution), and the set of support

```
      SUBROUTINE SOLARO (TGAL,INSTID,SIANG)
C ...
C       Input variables
      CHARACTER*(*) TGAL
      INTEGER INSTID
C       Output variables
      DOUBLE PRECISION SIANG
C ...
      CALL SCS2E ( GALILE, TGAL, ETGALI )
      CALL BODVAR ( JUPITE, 'RADII', DMY1, RADJUP )
      CALL SPKSSB ( GALILE, ETGALI, 'J2000', PVGALI )
      CALL SCE2T ( INSTID, ETGALI, TKINST )
      TJUPIT = SENT ( JUPITE, GALILE, ETGALI )
      CALL BODMAT ( JUPITE, TJUPIT, MJUPIT )
      CALL ST2POS ( PVGALI, PPVGAL )
      CALL SPKSSB ( JUPITE, TJUPIT, 'J2000', PVJUPI )
C ...
      CALL SURFNM ( RADJUP(1), RADJUP(2), RADJUP(3), P, PP )
      CALL MTXV ( MJUPIT, P, XP )
      CALL MTXV ( MJUPIT, PP, XPP )
      CALL VADD ( PPVJUP, XP, VO )
      CALL VSUB ( PPVSUN, VO, DVOPPV )
      SIANG = VSEP ( XPP, DVOPPV )
      RETURN
      END
```

Fig. 4.6. Example output of AMPHION

strategy. A flexible agenda ordering allows tactics to select the next clause(s) to be processed. In addition, SNARK incorporates efficient techniques for handling equality, using paramodulation and demodulation together with recursive path-orderings (RPOs). Many of the axioms of the domain theory are unconditional equations which are tried to be oriented using a Knuth-Bendix completion procedure. Recursive path orderings (which are domain-specific) restrict the use of paramodulation and guide the application of demodulation. Domain-specific knowledge is coded into RPOs which guide the search (i.e., the direction of application of the equations) from the abstract input specification towards the specification of the applicative program.

The Proof Tasks. The proof tasks processed by AMPHION (see Table 4.4) are certainly not trivial. Since the length of the proof corresponds to the length of the generated program (about 10–20 subroutine calls), we have medium deep problems. The size of the problems are with around 100 clauses of medium size. Usually one proof task is generated per session and has to be solved within a few minutes. The source logic of AMPHION is extended first-order logic; thus the distance for logic translation is small. SNARK

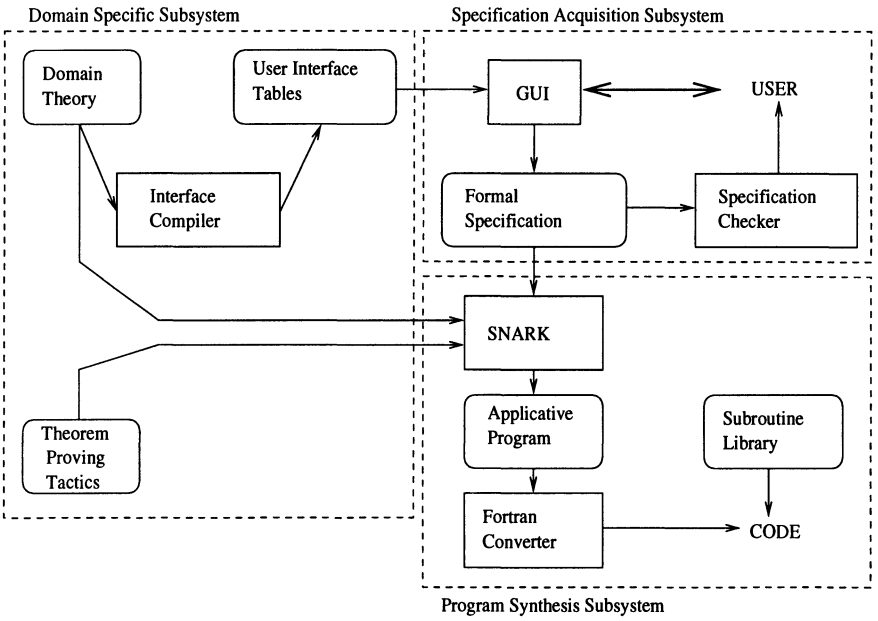

Fig. 4.7. AMPHION's architecture (taken from [Lowry *et al.*, 1994b])

uses equations and lambda terms as extensions to first-order logic. Due to the specification checker, all proof tasks comprise theorems. The expected answer is the assembled lambda-term, representing the applicative program. Semantic information is provided by the domain-specific subsystem in form of proof tactics, processing priorities (literal weights) and term orderings.

category	value		
deep/shallow	Shallow	**Medium**	Deep
size & richness	Small	**Medium**	**Large**
answer-time	Short	**Medium**	Long
distance	**Short**	Medium	Long
extensions	equations,λ-terms		
validity	100 % theorems		
answer	variable substitutions		
semantic info	**Y**	some	N

Table 4.4. Proof tasks of AMPHION: important characteristics

4.7.2 ILF

ILF [Dahn *et al.*, 1994] is an interactive proof environment ("Proof-Pad") which allows the interactive construction of complex proofs by using tactics. The system has been successfully applied to problems from mathematics (Mizar [Dahn and Wernhard, 1997]), hardware verification (verification of a microprocessor [Wolf and Kmoch, 1997]) and the verification of communication protocols [Dahn and Schumann, 1998], proving its usability in the area of high-quality software and hardware design.

For the verification of software protocols, an interface to the specification language Z [Spivey, 1988] has been defined and implemented [Dahn and Schumann, 1998]. Z specifications are directly translated into ILF's sorted first-order predicate logic. With the help of a graphical user interface and a tactics-based language, the proof tasks can be broken down into individual subgoals. Each subgoal is first tried by one of the connected automated provers (OTTER [McCune, 1994a], SETHEO, and DISCOUNT [Denzinger, 1995; Denzinger and Pitz, 1992][7]). The time-limit for the automatic tries — which are performed in parallel (see Section 7.6.2) — is set to several seconds. If during this time no proof could be found, the user has to further break down the proof obligation.

category	value		
deep/shallow	Shallow	**Medium**	**Deep**
size & richness	**Small**	**Medium**	Large
answer-time	**Short**	**Medium**	Long
distance	Short	**Medium**	Long
extensions	sorts, equality		
validity	depends on application		
answer	proof		
semantic info	Y	**some**	N

Table 4.5. ILF: characteristics of proof tasks

A unique feature of ILF is its capability to present the user a problem-oriented, human readable proof. Such a proof consists of a "patchwork" of small proofs, found by the automated provers and by the interactive application of tactics. By using a common (natural deduction style) calculus, the block calculus ([Dahn and Wolf, 1994] and Section 7.7), all proofs can be represented in a uniform style. This proof is then automatically postprocessed and typeset to obtain a LaTeX document (see Section 7.7 or 6.1 for an exam-

[7] Whereas the first two provers can handle arbitrary formulas in first-order logic with equality, DISCOUNT is restricted to problems that consist of equations only. In this domain, however, it is extremely powerful.

ple). The important characteristics of ILF's proof tasks are summarized in Table 4.5[8].

4.7.3 KIV and Automated Theorem Provers

KIV [Reif *et al.*, 1997; Reif *et al.*, 1998] is an interactive verifier, specifically suited for the verification of high quality software. Many industrial applications of KIV, e.g., the verification of an airbag controller [Reif, 1998] demonstrates the practical usability of this system. Based on dynamic logic, reasoning can be performed manually and by tactics. As with any interactive system, the time to find a proof can be considerably. For example, for the verification of certain refinement steps for a PROLOG Abstract Machine implementation [Börger and Rosenzweig, 1995], proof times of up to two months have been reported in [Schellhorn and Arendt, 1998][9].

category	value		
deep/shallow	**Shallow**	**Medium**	Deep
size & richness	Small	**Medium**	**Large**
answer-time	**Short**	**Medium**	Long
distance	**Short**	Medium	Long
extensions	sorts, equality		
validity	depends on application		
answer	TRUE/FALSE		
semantic info	Y	some	**N**

Table 4.6. KIV: characteristics of proof tasks

Hence, one aim of KIV's developers is to use automated theorem provers to process simple proof tasks without user interaction. To this end, the prover $_3T^AP$ [Hähnle *et al.*, 1992; Hähnle, 1993; Beckert and Hähnle, 1992] was integrated into KIV, and experiments on the expected performance of SETHEO, SPASS, OTTER, and Protein [Baumgartner and Furbach, 1994b] have been performed for selected domains [Reif and Schellhorn, 1998]. Characteristics of these proof tasks are shown in Table 4.6. One key-problem which was identified is the preprocessing of axioms: when working with the interactive system, one usually loads all theories (e.g., theory of natural numbers and arithmetic, lists, trees) which might be needed for the ongoing verification. When interactive steps are performed, the human user usually "knows"

[8] Here, we assume that the proof tasks are in first-order logic extended by sorts. A proof in the block calculus is expected as an answer. Although the corresponding transformation modules are a part of ILF, these modules can be used in a standalone manner, augmenting the ATP itself.

[9] Although this case study might seem a little academic, its complexity and size resembles a typical demanding industrial application.

which of the axioms to apply. Most automated theorem provers, however, are overwhelmed by the sheer number of axioms (often several hundreds) in the formula. Therefore, a powerful method has been integrated into KIV [Reif and Schellhorn, 1998] which is able to reduce the size of the proof tasks considerably (see Section 5.7).

Category	Range			Value(s)
Syntax				
number	**L**	M	H	$N = 3 - 8/session$
frequency	L	**M**	H	$f \geq 1/min$
size	**S**	**M**	L	$N = 40$ [30 . . . 60] clauses
prop. complexity	**L**	**M**	H	$l_{cl} = 2$ [1 . . . 3] lit/clause
richness	**L**	**M**	H	$N_{symb} = 15$ symbols
arity	**L**	M	H	$arity \leq 2$
nesting	nesting			**Y** N
	recursive			**Y** N (limited)
Logic				
source logic/language				BAN, AUTLOG
target logic				FOL (CNF,SETHEO)
$d_{\text{source-target}}$	S	**M**	L	
simplification	**N**	some	Y	
sorts & types			static	**Y** N (not used)
			polymorphism	Y **N**
			sort hierarchy	**flat** tree DAG other
equality	0%	. . .	100%	0% (no equality)
arithmetic	0%	. . .	100%	$\approx 5\%$
operator "+1"	**Y**	N		
other theories		**Y**	N	BAN, finite sets
		decidable (BAN)		**Y** N
	decidable (AUTLOG)			**Y**? N
Validity				
theorems/nonthms	0%	. . .	100%	80 – 90% theorems
complexity	L	**M**	H	$l_{BAN} \approx 3 - 10$
				$l_{FOL} \approx 3 - 30$
Answer	**proof** (domain-oriented notation)			
soundness	reasonable		**Y**	
completeness	**reasonable**		Y	
semantic info	**N**	some	Y	
System				
relation proof tasks	N	**some**	Y	
automatic generation	**Y**	N		out of PIL-specification
guidance	Y	**N**		
limits known	**Y**	some	N	fixed
human understandable	**Y**	some	N	

Table 4.7. Characteristics of proof tasks in an extended table format. The example is about verification of authentication protocols with the PIL/SETHEO system (Section 6.1).

According to [Reif, 1998], proof tasks which have been solved by the automated theorem provers have been relatively simple (corresponding to about 1-3 manual interactions). Ideally about 25-30% of the proof tasks eventually could be solved automatically within reasonable run-times.

5. Requirements

As illustrated in Chapter 3, current high-performance theorem provers essentially are carefully designed and powerful search algorithms. However, they lack many features required to successfully handle proof obligations, arising from real-world applications. In a way, automated theorem provers are like racing cars: they are very fast and powerful on racing tracks, but cannot be used reasonably in everyday traffic, because they don't have turn-indicators, break lights, etc.

In this chapter, we will set up requirements which an ATP should fulfill for its successful application. We describe each requirement and illustrate its importance. Selected requirements will be covered in Chapter 7 in detail, where techniques and methods will be developed or adapted.

These requirements are tightly coupled to the structure and characteristics of the proof tasks as discussed in the previous chapter. Furthermore, the underlying formal method is somewhat important: if, for example, a rather weak formalism (or logic) is used, one can expect to show only properties which can be expressed within that logic — regardless of the ATP's properties. This means that a prover cannot prove more than can be expressed in the underlying logic of the formal method.

5.1 General Issues

5.1.1 Expressiveness

In general, the proof obligations are formulated in the logic of the employed formal method. In most cases, this logic is different from a logic which directly can be understood by an automated theorem prover (here, we are thinking at first-order predicate logic). Hence, we will call the original logic of the proof task *source logic*, and that of the prover *target logic.*

Usual types of source logics, found in the area of software engineering are higher-order logic(s), modal logic(s), (extensions of) first-order logic, intuitionistic logic, or propositional logic.

If a proof task is to be processed automatically, one must either develop and implement a specific proof procedure for its source logic, or formulas of that logic must be transformed into a logic (in general a "lower" one) for

which a system exists. Due to the complexity of logic processing, there is no guarantee that a specific proof procedure can be implemented at all, or that a transformation exists. Luckily, however, proof tasks of a given application domain quite often (about 90 – 95% according to many researchers) do not use all "specialties" of the source logic. Often, a source logic has been selected which essentially is much too "powerful" for the required application. Reasons for this can be that such a logic allows to use simple notations which otherwise would be hard to read. A typical example is the abbreviation of classes of formulas, e.g.,

$$\forall P \in \{send, receive\} \cdot \forall Data \cdot correct(Data) \rightarrow P(Data)$$

This formula actually is higher-order (there is a quantifier over predicates). However, this formula can easily be transformed into FOL by simply generating all required instances.

In general, the more powerful the logic is the more complex (and possibly more inefficient) the proof procedures are for that logic. Therefore, it is essential to have a close look at the characteristics of the proof tasks (see Chapter 4) before selecting an automated prover. As mentioned before, major characteristics which might be able to restrict the required logic are:

- does the application have a finite state space only? Typically, descriptions of finite automata, protocols and other similar areas of application are good candidates for such a restriction. In such a case, the formula gets decidable and specific procedures (e.g., propositional solvers, model checkers) might be the appropriate choice.
- does the application have finite domains only? Finite domains typically occur in situations where an enumeration domain occurs (e.g., $x \in \{1, \ldots, 8\}$ for a chess board), or all elements (e.g., strings, messages) are have a maximal (a priori known) length. Here, a similar approach as above can be of interest.
- where are the quantifiers and what is their structure? When transforming higher-order logic or modal logic into classical logic, structure and scope of the quantifiers play a crucial role. Certain (but often occurring) structures (e.g., all higher-order variables are all-quantified and the quantifiers occur on the outermost level only) can easily be transformed without loss of generality. Other cases cannot be transformed at all (because HOL is strictly more powerful than FOL). For a more detailed discussion of this issue see Chapter 7.2.

Even when the application uses a logic for which a simple transformation into a weaker logic exists, additional considerations are necessary: the development of automated provers for classical first-order logic, propositional logic and modal propositional logics has a long history and much effort has been spent to squeeze out the last bit of efficiency. Therefore, it is advisable to use an "off-the-shelf" prover rather than a special-purpose system which has to be newly implemented in the first place. However, transformations from one

logic to another (in general weaker one) has also its pitfalls: some transformations tend to blow up the input formula (often exponentially), so that the time needed to preprocess such a huge formula already might be larger than searching for the proof with a low-efficiency special-purpose system. Other transformations can destroy the structure of the formula. Then, for example, it is extremely difficult if impossible to "re-translate" the proof back into the source logic. Finally, the transformation itself can be highly complex with respect to resources and run-time.

Hence, the following key questions (see Chapter 7.2 have to be addressed: If I am using a general purpose theorem prover and I have to transform my logic, do I loose expressiveness and information during the transformation? And, is the transformation effective, i.e., do I gain efficiency over a special-purpose prover?

Figure 5.1 shows the expressiveness of different logics and (some) possible transformations between them. Please note that this diagram does not show all possible transformations. In Section 7.2, we will discuss transformations of one important kind of higher-order construct, namely induction, and the translation of modal logics into first-order logic. A transformation of first-order logic into Horn clause logic is the basis of the PTTP theorem prover [Stickel, 1988].

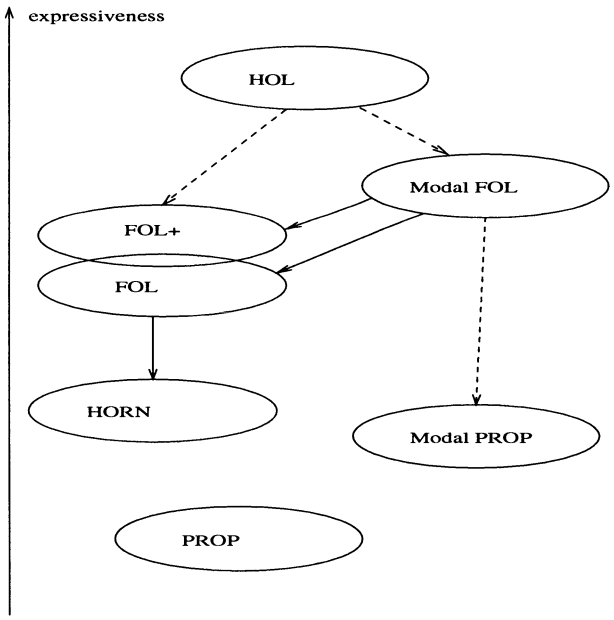

Fig. 5.1. Expressiveness of different classes of logics. Solid arrows mark possible transformation between the different logics (FOL+ denotes first-order logic with extensions).

5.1.2 Soundness and Completeness

In general, a theorem prover (and its underlying calculus) should be complete and sound. This means that one can assure that we are not losing any proofs and each proof found by the system is indeed a proof.

Due to the undecidability, completeness of a theorem proving procedure only guarantees that a proof (if there exists one) is eventually found. However, the length of a run of the system is always limited. Thus, all practical implementations of ATP's are incomplete. In practical applications, run-time limits are often very short. Then, requirement of the prover always being complete, must be considered carefully.

In this context, we can distinguish between benign and malevolent incompleteness. In the first case, we are not able to find all proofs, but at least, we can obtain most of the proofs, we are interested in (e.g., short ones, proofs with low complexity). "Bad" incompleteness will cause us to loose many important proofs. As an example, we can take PROLOG's unbounded depth-first search which very often encounters endless loops although a simple proof (or solution) exists (see Example 4.3.1 on page 56). Hence, for any applications, our aim is to find the obvious proofs always, and to find as many complex proofs as possible. Possible ways of introducing "controlled" incompleteness (discussed in the appropriate sections in more detail) could be leaving out certain axioms, limiting the size of terms, or changing weight and usability of parts of the formula.

Soundness on the other hand can be ensured for most theorem provers and applications. An unsound prover would be rather embarrassing since in that case, the user does not know whether the proof returned by the system is a proof or not. This situation, however, should not be intermixed with that of the inconsistency of a formula: if a formula (in most cases, a set of axioms) is inconsistent, a prover will always return a proof regardless of the formula's validity.

For specific purposes, some applications might even allow to sacrifice soundness: for example, to filter out "promising" proof obligations, we might use a prover which simplifies the problem, e.g., by abstraction (converting all terms of the predicates into empty strings). When the prover finds a solution for that simplified problem, then there might exist a proof for the original problem. On the other hand, however, we know that if *no* proof can be found for the simplified problem, we know that there cannot exist a proof for the original one. Thus we can reject this proof obligation. Such techniques are of interest in many applications and will be discussed in more detail in Section 5.7.

5.1.3 Can You Trust an ATP?

If a deductive system returns a result, how can we check if the result is correct — even if the prover is sound and correct? If a deduction system

finds a model, this model normally can easily be trusted. One can always check out this model by applying the variable assignments to the formula and then checking its validity. This is straightforward and can be done easily by hand if the model is not too large.

Automated theorem provers and model checkers, on the other hand, operate with exhaustive search. Their result that there is no model within the given limits (or there is no model at all) is much more questionable, because the user has to rely on the correct implementation of the search algorithm. Even in case, the prover returns "proof found", it is difficult to check the correctness of the proof[1], since most machine-generated proof tasks are extremely hard to read. Hence, it would be helpful to have a reliable system which can detect, if a proof-structure is indeed a proof or not (with respect to the given formula and calculus). Such a *proof checker* is, for most calculi (e.g., model elimination, resolution), rather simple to implement. It takes the generated proof and the formula as its input and checks that all inference rules have been applied in a correct way and that all terms in the proof are instances of terms of the formula. Since a proof checker does not perform any search, the resulting algorithm is rather simple and efficient. Therefore, it might be feasible to even verify this piece of code[2]. Techniques like proof-carrying code (PCC, [Necula and Lee, 1998]) specifically use proof checkers to automatically ensure the correctness of the verified code even in a hostile environment.

Although even the automated prover may work correctly, many errors can occur in all parts of the application: there might be errors in the formalization, its transformation into (first-order) logic, preprocessing of the resulting formulas, or in the axioms. In order to be sure about the correctness all parts of the system should be observed carefully.

5.1.4 Proving and Other AI Techniques

Depending on the application's needs, one has to carefully consider, if a theorem prover is the right choice for processing the proof tasks. Other AI techniques (e.g., case based reasoning, neural networks, taxonomies) are often much more powerful in certain applications. In that case, such methods should be used instead, or a combination of automated deductive methods with other AI techniques should be sought. For instance, AI methods could be used to help to locate a solution while the automated prover is used to ensure correctness.

Example 5.1.1. The following examples show possible combinations of AI techniques with automated deduction.

[1] Things are getting much more complicated, if the prover does not provide a proof.

[2] For higher safety, one might even thinking of using a verified compiler (see, e.g., [Gaul, 1995]).

- Techniques of proof planning [Bundy, 1996] are used to automatically find the structure of a complex proof. A combination with a theorem prover (e.g., ΩMEGA [Benzmüller *et al.*, 1997], or the interface between HOL and Clam [Slind *et al.*, 1998]) provides techniques for checking and adapting proof plans towards the current proof obligation.
- Logic-based component retrieval of software components (see Section 6.3) tries to find matching modules in a library, given a query specification. The naïve approach (checking against each library component) would result in extremely many proof tasks, leading to unacceptable answer times. Here, other techniques developed in this area like taxonomies, fulltext retrieval can be combined with logic-based retrieval, providing effective preselection of promising module candidates.
- Other reasoning techniques (like case-based reasoning, planning methods, fuzzy reasoning, neural networks) can be combined with automated deduction. Here, the major role of the automated prover is to ensure correctness while being guided by the results of the AI technique.

5.2 Connecting the ATP

A major requirement for an automated theorem prover concerns its possibility to be "hooked up" to an application system (e.g., interactive theorem prover, verification system). Only, if a clearly defined interface between the two systems exists, proof tasks can be exported to the ATP and results can be expected in a correct way. This interface influences many design decisions. In the rest of this section, we will discuss the major characteristics of typical interfaces and describe the requirements arising from such a combined system.

When an automated theorem prover is connected to an application system, it must conform to such an interface (protocol), i.e., the way data and control information is passed to the prover, results are transferred, the prover is started and stopped, and parameters are transferred. In the following, we will discuss the important issues (from the side of the ATP) of

1. *reading in and preparing the proof task*,
2. *starting the prover(s)*,
3. *assembling and analyzing the results*, in case a proof could be found,
4. *stopping the ATP*, if no proof can be found and in other situations (e.g., errors or a user interaction), and
5. a final and careful *cleaning up* of the ATP's files.

A generic system architectures will be discussed in more detail in Chapter 7. Since the application system is in charge of controlling the entire application, these issues also contain requirements for the application system itself. Probably the most important point is when to decide to actually try

an automated prover to solve the proof obligation at hand. In particular in combination with interactive theorem provers, this problem can be arbitrarily hard, since a proof task in general provides no information, whether it can be solved automatically with reasonable resources or not. Thus, the ATP can be activated in a user-driven way (i.e., the user issues a command or tactic which exports the proof obligation to the ATP), or a heuristics-driven way, where heuristics try to decide which proof obligation to try automatically. A further approach could be the background operation of the prover: each occurring proof task is automatically sent to the ATP which runs in the background (or even on a different processor). This approach allows to meaningfully use the time between the occurrence of the goal and the user's reaction. If the proof task is easy to solve, the ATP will come back with a solution before the user has decided which action to performs with the current proof obligation. If the ATP does not find a solution, the user just will proceed in the usual way. Such a feature is implemented into the system *veracity* [Williams, 1994]. If, however, such a background operation is ergonomic or just a nuisance for the user is questionable and probably strongly depends on the application.

5.2.1 Reading and Preparing the Proof Tasks

In general, a proof task consists of the theorem to be shown (the "goal"), hypotheses, additional assumptions and axioms of the background theory (usually called *domain theory*). Additional information, e.g., lemmata, definitions of sorts, can augment the proof task. Such a proof obligation is represented in the application system's logic and syntax. Now, the ATP has to perform two different tasks: *reading in* the proof task, and *preparing* it in such a way that it can be processed by the prover[3].

Reading in a proof task requires a fully defined language for exchanging proof tasks between the application and the prover. The ATP (and/or its preprocessing modules) have to reliably reject any syntactically incorrect proof task.

Although this task may seem obvious and trivial, its importance is often underestimated. Errors, left undetected can easily result in unsound or incomplete behavior of the ATP.

Example 5.2.1. The following examples (which actually happened within our case studies) illustrate errors occurring during preprocessing of formulas: an application system produced a formula, consisting only of *one* single line. Although the ATP's preprocessing module accepted formulas in a format-free way, it could process one-line formulas only up to a given (but undocumented) size. The rest of the formula was silently skipped by the module, leading to

[3] It depends on the application, whether logic and syntactical representation of the proof task is already transformed in the application system to meet the ATP's input language, or whether the transformation is left to the ATP.

an incomplete system. Thus, often large, but almost trivial proof tasks could not be solved.

In another case, a proof task of the form $(Ax_1 \wedge \ldots \wedge Ax_n) \rightarrow Thm$ with axioms Ax_i was generated by the application system in the following way: first print an open parenthesis, then call the function to generate $Ax_1 \wedge \ldots \wedge Ax_n$, then print a closing parenthesis and the rest of the formula. However, there existed cases with $n = 0$. Then a formula $() \rightarrow Thm$ was generated which was read by the ATP's input module as the empty clause. This resulted in an unsound behavior such that all given proof tasks with $n = 0$ where trivially true.

An early version of SETHEO did not recognize the token TRUE as such. Thus generated formulas with subformulas of the form TRUE $\wedge \mathcal{F}$ could not be solved, because $\mathcal{F}$ was discarded from the formula. The reason was that TRUE was handled as an ordinary predicate symbol which, however, had no axioms attached. Therefore it was regarded *pure* and set to false.

Other, problems which are often encountered concern the length of symbols, or specific infix operators (which may not be defined or can have a different binding power). Therefore, an interface language which is as simple as possible and which is exactly defined is a prerequisite for any successful application. Furthermore, good ways of error handling must be provided.

Depending on the capabilities of the application system and the ATP, it can be the best (or indeed the only) way to generate all data separately for each proof task. Otherwise, a proof task can be generated incrementally, i.e., common parts (e.g., axioms) are generated and prepared only once. If each proof tasks contains all data, considerable run-time for reading in (and preprocessing) the proof task can sum up, making the application of the ATP infeasible. An incremental approach, however, requires that the ATP is capable to process partial formulas (e.g., in a way a linking-loader does). Most current high-performance provers lack such a feature.

When the proof task has been read in by the ATP, it must be *prepared* in such a way that it can be executed by the ATP's proof procedure. This preparation phase usually consists of a number of individual steps which typically concern syntactical transformations, transformations of the formula, preselection of axioms, incorporation of sorts, etc. The most important (and complex) preparation modules are in charge of transforming the application's logic into first-order predicate logic, the transformation of arbitrary first-order formulas into clausal normal form, and the transformation of ATP proofs into proofs of the application (if required).

Figure 5.2 shows an example, used in one of our case studies. Although many of these steps seem to be trivial syntactical transformations, care must be taken that the entire chain of transformation is robust and implemented correctly. Otherwise, hard to detect errors might occur, possibly sacrificing soundness and completeness of the ATP-system. Important requirements will be discussed in Section 5.3 (transformation of logic), Section 5.5 (simplifica-

tion), Section 5.6 (correct handling of sorts), and Section 5.7 (preselection of axioms).

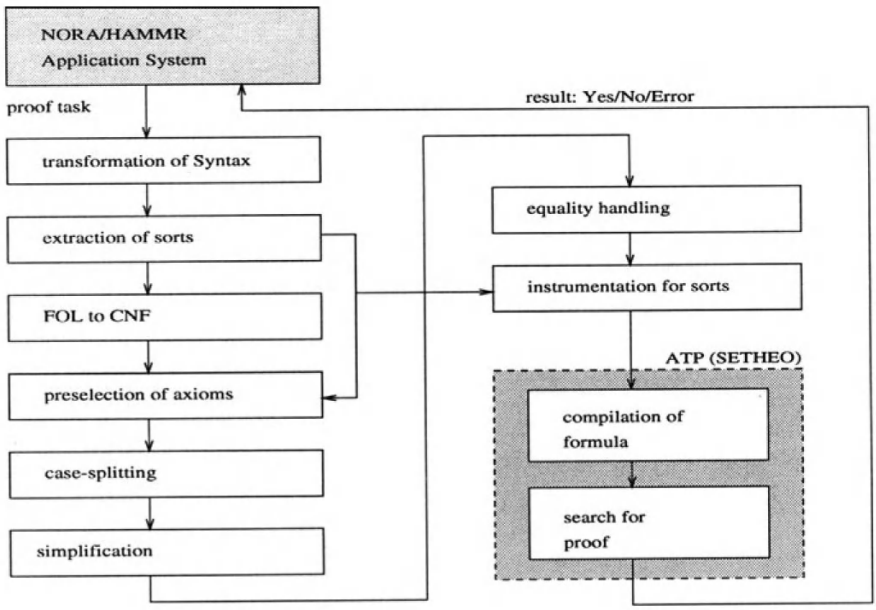

Fig. 5.2. Typical modular structure of an ATP-application-system: this figure shows the ATP (here: SETHEO) and its preprocessing modules for handling proof-tasks in a software-reuse environment (for details see Section 6.3).

5.2.2 Starting the Prover

Once the proof task has been fully prepared, the prover must be started with the appropriate parameters. The question which parameters should be set for the given proof task is extremely difficult and will be discussed in Section 5.9 and Section 7.6.

When an ATP is embedded, its start-up time is of interest, because it adds to the overall run-time, the ATP can use to process a proof task. However, a number of high-performance provers (mostly those, implemented in PROLOG or Lisp) exhibit considerable start-up times (in the order of 10s of seconds). This, of course, poses a severe problem for applications, where the ATP is to be started separately for each proof task. One way to overcome this problem is to start the ATP once and "pipe" the proof tasks into the prover as they show up.

5.2.3 Stopping the Prover

Since virtually all theorem proving procedures have an extremely high time-complexity, there always exist proof tasks which cannot successfully be processed with the resources available for that proof task (even if the procedure is decidable).

Of particular importance with this respect is the run-time allotted for the ATP. Especially interactive application systems (e.g., interactive theorem provers) expect that the automatic subsystem to return an answer within a couple of seconds (to normally up to several minutes) in order not to keep the user waiting too long. Therefore, the ATP must be stopped, if its available resources have been exhausted. Besides returning the appropriate information to the AS, the following items must be considered carefully:

- The ATP must stop completely and reliably. In particular processes which have gone astray after the main proof task has been aborted, can pose severe problems (in particular in distributed environments). Furthermore, one must take into account the time needed to stop the system. Especially parallel provers can take up to a few minutes to locate and kill all processes[4]. In such cases, asynchronous operation should be considered (i.e., the ATP is considered to have stopped as soon as the stop-command has been issued).
- The resource limit should be set in a *reproducible* way. If, for example, a wall-clock time limit is set (e.g., 1 minute), it will depend on the load of the workstation, how many proof tasks can be handled successfully. This, e.g., can mean that a successful run of the application system during the weekend cannot reproduced during the week!

5.2.4 Analyzing the Results and Cleaning Up

Whenever the ATP has stopped, the results must be analyzed carefully. If a proof has been found, it might be necessary to return information about the proof or the proof itself to the application system. If the application system's logic is different from the ATP's, a complex transformation of the proof might be required. However, there are cases where information is lost during the formula preparation, such that a back-transformation of the proof is not possible. These issues will be discussed in more detail in Section 7.7. Quite often, information about the axioms and assumptions which actually been used for the proof is of importance for the application. This information can usually be extracted without much overhead from the results the prover provides. Furthermore, variable substitutions can be important for the application. E.g., in the synthesis system AMPHION (Section 4.7.1), substitutions of certain variables comprise the synthesized program (proof as program).

[4] Even sequential provers can sometimes take quite a while before quitting, e.g., if memory management has to be performed or temporary files (often many and large files) have to be deleted.

Also in the case, no proof could be found, feedback to the application system might be required. In the ideal case, the application system would get information on why the current proof task could not be solved (e.g., because an axiom was missing). Current automated theorem provers, however, do not have this feature. In Section 5.8 we will discuss what has to be done, if many of the proof obligations are not provable, i.e., they are non-theorems.

In all cases, assembly of statistical data (e.g., on run-times, characteristics of proof tasks and proofs) can be important, in particular during the development and tuning of the system.

5.3 Induction

Whenever recursive data structures, or recursively defined functions occur in a proof task, it is likely that such a proof task can be shown only by induction. Although the proof task itself (or its translation) is classical FOL, an automated theorem prover might not be able to process such a proof task successfully.

Typically, the handling of induction was restricted to interactive theorem provers and special-purpose inductive theorem provers. Given higher-order recursive definitions of data structures, induction schemata and lemmata can be extracted semi-automatically (Section 5.7). Automated provers for handling induction typically work with a detailed inspection of the term structure of the proof tasks. With this technique, called *rippling*, information can be obtained, on which variable(s) induction must be carried out, and which induction scheme is to be used. Several implementations (e.g., Oyster/Clam [Bundy *et al.*, 1990]) are able to automatically process quite complex inductive proof tasks. These inductive theorem provers, however, consume a large amounts of computing resources and are relatively slow. Thus, they are not feasible for many applications investigated in this book.

When looking at typical applications, however, one will see that in many cases, much additional information is available for handling inductive proof tasks. With this information, a proof task can be augmented in such a way that it can be processed with an ordinary automated theorem prover. Of course, some intricate inductive problems which e.g., would require specific generalizations of the induction hypothesis, or specific lemmata, cannot be solved. However, many of the simpler obligations (and thus interesting for automatic processing) can be handled efficiently. With information provided by the application, an automated theorem prover should be able to provide the techniques for handling standard inductive problems. These techniques will be discussed and assessed in detail in Chapter 7.2.1.

5.4 Equality and Arithmetic

5.4.1 Handling of Equality

The notion of equations is central to almost all formal methods. Statements, properties, descriptions are very often written in an equational form with with a separate left-hand side and right-hand side. Of course, both sides are not only set equal (by "="), but also other relational operators (e.g., $\leq, \geq, \neq$) are used. It seems to be the case that equations are in particular suited to visualize complex formal expressions and to perform operations with them. Thus, even in the somewhat artificial benchmark library TPTP [Sutcliffe *et al.*, 1994], about 54% of the problems contain equations. Thus, an automated theorem prover must be able to efficiently handle equations.

In general, each prover for first-order predicate logic can handle equations. As described in more detail in Section 3.3.2, one can always add the standard congruence axioms for equality (reflexivity, symmetry, transitivity, substitution axioms). However, this naïve approach in general results in huge search spaces to be explored. Therefore, specific techniques must be present for efficient handling of equational proof obligations.

In general, we have to distinguish between unconditional equations (e.g., $a = b$) and conditional ones: here, equations occur in the formula together with other predicates, e.g., $p(a) \wedge X > 5 \rightarrow a = b$. For formulas which only contain unconditional equations (or inequalities), there exist powerful automated theorem provers. These theorem provers (e.g., Discount [Denzinger and Pitz, 1992], E [Schulz, 1999], or Waldmeister [Hillenbrand *et al.*, 1999]) usually work with Knuth-Bendix completion methods [Knuth and Bendix, 1970]. They are capable of solving large sets of equations in very short times.

Also many automated theorem provers have built-in techniques for handling equations. In particular for bottom-up resolution provers like OTTER, Gandalf, SPASS, or SNARK, there exist many standard methods (e.g., paramodulation). E-SETHEO preprocesses the formula using a variant of Brand's STE method (Section 3.3). All special-purpose methods, however, introduce some overhead during the search for the proof. Therefore, in cases where there is only little equational reasoning, it can be better to use the axiomatic method.

5.4.2 Arithmetic

Besides equations, most application proof tasks contain arithmetic expressions. Except for few application domains (e.g., numeric mathematics, modeling of physical laws[5]), mostly rather simple integer-arithmetic is occurring.

[5] In typical examples, like the Steam-boiler specification [Abrial *et al.*, 1996] or modeling of an EVA Rescue unit [Kelly, 1997], floating-point numbers and arbitrary expressions are used to express physical laws. In the course of modeling for verification, however, quite often, abstractions can be found to circumvent such symbolic and numerical calculations.

For many applications, it is even possible to restrict the domain to natural numbers and addition. Typical examples in this area are expressions for accessing array elements.

Nevertheless, an automated theorem prover must be capable of handling such expressions efficiently in order to be applicable. A problem, often underestimated is that such expressions can contain (logic) variables. This means that often expressions cannot be simply evaluated, but must be "solved". PROLOG's built-in arithmetic (operator `is`), for example, can only evaluate an expression, if all its variables are instantiated. In general, each expression over natural numbers can be written using the successor function s. For each natural number $n \in \mathbf{N}$, we define:

$$n \equiv s^n(0) = \underbrace{s(s(\ldots s(0)\ldots)}_{n}$$

Then we can make recursive definitions for all required operations. For addition, we yield the two rules: $0 + n = n$, and $s(m) + n = s(n + m)$.

Such a transformation, however, results in extremely large terms and complex proofs which span huge search spaces. The following simple example illustrates this.

Example 5.4.1. Let us consider the standard recursive definition for the Fibonacci numbers:

$$\begin{aligned} &\text{fib}(0) = 1 \\ &\text{fib}(1) = 1 \\ &\forall N : \; N > 1 \; \cdot \; \text{fib}(N) = \text{fib}(N-1) + \text{fib}(N-2) \end{aligned}$$

When written as a logic program with a query $\text{fib}(n) = X$ for some natural number n, we obtain results (using SETHEO) as shown in the first part of Table 5.1, labeled "nat-arith". For each experiment we show the number of inference steps (*inf*) in the proof and the run-time in seconds (T), obtained on a SUN ultra-sparc. If we use $s(s(\ldots s(0)\ldots)$ as the representation of a natural number n, the proofs and the corresponding search spaces get much larger (Table 5.1, line marked "succ-arith").

This problem has the particular feature that operations in the same data must be performed repeatedly. If we keep the intermediate results ("caching", or lemmatizing), we are able to cut down the size of the proof substantially (last row of the table).

Fortunately, powerful solving methods for such simple kinds of arithmetic as described above (e.g., Pressburger Arithmetic) exist. However, it is far from trivial to incorporate such solvers into a general purpose theorem prover. For all formulas containing arithmetic expressions, it is essential that these expressions are *simplified.* Only then, unnecessary and time-consuming proofs can be avoided. Although such simplifications are state-of-the-art in compiler design and symbolic algebra systems (like Mathematica [Wolfram, 1991]

query:		fib(5)	fib(10)	fib(15)	fib(20)
result:		8	89	987	10946
nat-arith	*inf*	15	177	1973	21891
	T	< 0.01	0.14	1.79	4.81
succ-arith	*inf*	32	500	6929	> 50000
	T	< 0.01	0.19	11.96	> 220
nat-arith	*inf*	9	19	29	39
w/caching	*T*	< 0.01	< 0.01	< 0.01	< 0.01

Table 5.1. Calculation of Fibonacci numbers as a logic program: different representation of natural numbers (SETHEO)

or Maple [Char *et al.*, 1988]), current automated theorem provers lack this important feature.

Example 5.4.2. When automatically generated by an application system, arithmetic expressions are often not simplified. As an example, let us consider an array-declaration `integer a[21,21]`. If, in a proof task, the offset for `a[0,20]` is needed, an automatic system might generate the offset as $((s^{20}(0) + 0) + s^{21}(0) \times 0) \times s(0)$. A typical prover, like SETHEO needs 139 inferences (about 40ms), to evaluate this expression. A simplification during preprocessing would just yield the term $s^{20}(0)$. In particular, high overhead results, if expressions of this kind must be processed over and over again during the search for a proof.

5.4.3 Constraints

Many formal methods use a logic with *constraints* as a basis for representation and reasoning. In that case, clauses c of a formula are augmented by sets of constraints C, such that $\mathcal{F} = \bigwedge c : C$. This means that $\mathcal{F}$ is valid, if for each variable substitution which yields TRUE, all constraints must hold. Typical kinds of constraints are: *symbolic* constraints (e.g., [apple $\neq$ pear] or [X = *apple* $\vee$ *pear*]), or *numeric* constraints (e.g., [$X > 5$]). Although a first-order language with constraints is obviously only as powerful as pure first-order logic, such an extension is convenient in many cases.

The distinction between ordinary clauses and constraints in principle allow efficient methods for constraint solving to be applied. In the area of logic programming, this approach has been applied quite successfully (constraint logic programming, [van Hentenryck, 1989; Estenfeld, 1986]). Its integration into a first-order automated theorem prover, however, requires much effort and has been attempted only in a few cases (e.g., [Stolzenburg and Thomas, 1998; Caferra and Peltier, 1997]). Often, however, selected subclasses of constraints (e.g., symbolic inequality constraints) are available.

5.5 Logic Simplification

Most automated theorem provers have been designed to handle formulas for which rather complex and lengthy proofs exist. The formulas themselves only contain the clauses necessary to find a proof. Due to this historic reason, most automated theorem provers are not capable of efficiently handling large formulas. The time to preprocess a large formula is in general too high and the size of the search space is too large to be acceptable. This is in particular true, since most theorem provers do not detect obvious unusable parts of the formula. In most applications, where proof obligations are generated automatically formulas are not minimal. This means that they contain unnecessary parts (e.g., $\mathcal{F} \vee \text{TRUE}$).

Thus, an automated theorem prover must be capable to (logically) *simplify* a formula before processing it. This involves in particular: throwing away obvious true or false parts of the formula, tautologies, quantifiers for variables which are not used, and pure literals. If additionally, semantic information is available, it also should be used for simplification. For instance, a literal $\text{cons}(a, b) = [\,]$ can instantly be simplified to FALSE.

5.6 Sorts

Most formal methods in the area of software engineering use a logic which is augmented by sorts or types. With such an extension, all symbols and variables are marked to which kind of object they belong. Like the data types in most programming languages, these constructs are used to facilitate user-friendly expression of complex data structures and to enhance readability, because for each symbol, it is immediately evident to which sort it belongs. Moreover, a system which uses a sorted logic is able to easily detect many incorrectly sorted formulas (e.g., $5 + cons(1, nil)$), a feature especially helpful for manual input of formulas.

In their most general form, typed languages (e.g., typed lambda-calculus) are very hard to handle. The determination of the correct sort of an expression can be arbitrarily hard — in general it is undecidable. Just consider the following small example:

$$\forall P : \mathsf{program} \cdot (pre_P \wedge impl_P \rightarrow post_P) \rightarrow P : \mathsf{correct\_program}$$

which means that a program is correct, if you can prove that its implementation ($impl_P$) obeys the postcondition ($post_P$) on its legal input domain (pre_P). Thus the determination of the sort for some P can involve complex proofs. Such an approach is implemented into the PVS system [Crow *et al.*, 1995]. There, a proof task is broken up into smaller ones by generating new proof tasks which have to ensure that the correct types are instantiated.

In many cases, however, *sorts* are much more simple and directly reflect the data-type structure of the underlying programming or specification language. Here, typically, the determination of the sort of a symbol is much easier. Before we focus on the requirements for an ATP's system for handling sorts, we give a more formal definition of essential notions.

Sorts are used for the classification of entities. Therefore, a non-empty set S of sorts (usually constant symbols) is given. Then, all syntactic objects of a first-order language (i.e., variables, constants, function symbols, and predicate symbols) are assigned specific sorts. For a constant c of sort $\mathsf{s} \in S$ we write $c : \mathsf{s}$. An n-ary function symbol f is of the sort

$$f : \mathsf{s}_1 \times \ldots \times \mathsf{s}_n \longrightarrow \mathsf{s}_0$$

with sorts $\mathsf{s}_0, \ldots, \mathsf{s}_n$. An n-ary predicate symbol p (denoting an n-ary relation) has the sort declaration

$$p : [\mathsf{s}_1 \times \ldots \times \mathsf{s}_n]$$

Variables of sort s are written as $X : \mathsf{s}$. A sorted logic of this structure is called *many-sorted*. If the set S of sorts entirely consists of constants, we have *monomorphic* sorts. In that case, each object of the language is of a given, distinctive sort. For many practical applications, it is important to have sorts with parameters, i.e., *polymorphic* sorts. Here, sort definitions can be parameterized, representing entire classes of sorted structures. If the same function or predicate symbol is used for different sorts, we call it *overloaded.*

Example 5.6.1. Let us assume, we have the sorts $S = \{\mathsf{color}, \mathsf{nat}, \mathsf{list}\}$. Then, well-formed term and predicate definitions in a sorted logic would be (cons denotes the list constructor for lists of natural numbers, the predicate *isempty* checks, if a list is empty):

$$\begin{array}{ll} \mathtt{cons} & : \mathsf{nat} \times \mathsf{list} \longrightarrow \mathsf{list} \\ [\,] & : \mathsf{list} \\ \mathit{isempty} & : [\mathsf{list}] \end{array}$$

Then, $\mathit{isempty}(\mathit{red})$ with $\mathit{red} : \mathsf{color}$ would not be a well-formed formula.

If generic lists are required, a polymorphic sorted logic can be used. Then, the declaration of the list construction operator for a list of items of sort T might look as follows:

$$\mathtt{cons} : \mathsf{T} \times \mathsf{list\ of\ T} \longrightarrow \mathsf{list\ of\ T}$$

Then, a variable denoting a list of natural numbers would be written as $X : \mathsf{list\ of\ nat}$.

Up to now, the sorts represent disjunct classes of entities. However, sorts can also be ordered with respect to a specific relation "subsort of", denoted as $\sqsubseteq$. With this relation, *hierarchies* of sorts can be constructed. E.g., $\mathsf{nat} \sqsubseteq \mathsf{int} \sqsubseteq \mathsf{float}$ which means that an object, belonging to a subsort (e.g., 0 : int)

automatically belongs to its super-sort, but not vice versa. If all sorts are disjunct to each other and if there is no hierarchy, we are talking of a *many-sorted* problem, or a *flat* sort hierarchy.

Due to the importance of sorts in most applications, an automated theorem prover should be able to handle formulas from sorted first-order logic efficiently. ATPs must process sort information in order to be correct. Otherwise, incorrect proofs, based on illegally-sorted terms (e.g., $cons(nil, nil)$) can occur. The general transformation of a formula in sorted logic into pure first-order logic is always possible: the sort information is compiled into a specific predicate of arity one. Then, the determination of the correct sort is a classical deduction problem. For example, a literal $p(X : \text{NAT})$ would be transformed into $p(X) \wedge \text{sort\_of}(X, nat)$. This approach, however, induces huge search spaces and is thus not feasible for most applications.

In order to determine which technique of handling sorts is most appropriate, one must have a look at the given characteristics of sorts and their hierarchy. For certain restrictions, the generally undecidable problem of determination of the correct sort becomes tractable. For example, if the sort hierarchy is a directed acyclic graph (DAG) (or an upper semi-lattice)[6], sorted unification is always unitary, i.e., it produces one most general unifier. This definitely is an important prerequisite for sorts to be handled efficiently by automated provers. In the other case, each sorted unification would open up its own search space. In Section 7.4, we will present several approaches for efficient handling of sorted formulas, obeying the above restrictions.

5.7 Generation and Selection of Axioms

A formula comprising a proof task consists of several parts: the theorem to be shown (the "goal"), hypotheses and assumptions. Furthermore, the theory (or theories) defining all operators must be present. In case, no special-purpose handling of a theory exists (e.g., for handling the theory of equality, see Section 5.4), the theories are specified as sets of *axioms*. Table 5.2 gives a typical example: the construction of lists with *cons* and *append*.

Proof tasks, originating from a rich domain with many operators can thus have up to several hundred axioms. Automated theorem provers, however, have severe problems with handling proof tasks with large sets of axioms. The increased time to preprocess and read in such large formulas alone can be a problem. Much more severe, however, is the fact that additional axioms tremendously increase the search space to be traversed. The size of the search space increases, even if the axiom cannot be used for the proof. This can lead to the situation that a trivial proof task (with only a few proof steps) cannot be processed successfully with the automated theorem prover. Such a proof

[6] For a proof see, e.g., [Bahlke and Snelting, 1986].

(1) $\forall X, Y : \mathsf{list}\ \forall I, J : \mathsf{item} : cons(I, X) = cons(J, Y) \rightarrow I = J \wedge X = Y$
(2) $\forall X : \mathsf{list}\ \exists Y : \mathsf{list}\ \exists Z : \mathsf{item} : X = cons(Z, Y) \vee X = [\,]$
(3) $\forall L : \mathsf{list}\ \forall X : \mathsf{item} : cons(X, L) \neq [\,]$
(4) $\forall X : \mathsf{list}\ \forall Y : \mathsf{list}\ \forall I : \mathsf{item} : app(cons(I, L), X) = cons(I, app(L, X))$
(5) $\forall L : \mathsf{list} : app([\,], L) = L$
(6) $\forall X : \mathsf{list}\ \forall Y : \mathsf{list}\ \forall Z : \mathsf{list} : app(app(X, Y), Z) = app(X, app(Y, Z))$
(7) $\forall X : \mathsf{list}\ \forall Y : \mathsf{list} : app(X, Y) = [\,] \leftrightarrow X = [\,] \wedge Y = [\,]$
(8) $\forall L : \mathsf{list} : app(L, [\,]) = L$

Table 5.2. Typical first-order axioms for constructions of lists ("cons") and concatenation of lists ("append")

task only would require few axioms to be added in order to be provable. Table 5.3 shows a typical behavior (an example from Section 6.3).

Exp	Axioms								T[s]
	1	2	3	4	5	6	7	8	
1	•	•	•	•	•	•	•	•	> 100
2	•	•	•	•	•	o	•	•	87.5
3	•	•	o	o	o	o	•	•	2.2
4	o	•	o	o	o	o	•	•	0.2
5	o	o	o	o	o	o	o	o	no proof

Table 5.3. Influence of selection of axioms (• = axiom present) on run time (SETHEO). Axioms are taken from Table 5.2.

Hence, a selection of such axioms which will be needed for the proof is required for many applications of automated theorem provers. However, the selection of such a subset itself is undecidable. Therefore, a good *preselection* of axioms is needed which throws away obviously unnecessary axioms without sacrificing completeness by eliminating too many (see Figure 5.3). Some applications might even allow for slightly incomplete preselection of axioms, as long as not too many valuable proofs are lost.

For a further discussion of this topic, we also have to have a look at where the axioms are coming from and what their internal structure is. In cases with a rich domain, theories are typically structured in a hierarchical manner.

Example 5.7.1. An example from [Reif and Schellhorn, 1998] on a proof task

$$\#en = \mathrm{succ}(0) \rightarrow \emptyset \oplus \mathrm{el\_of}(0, en) = en$$

about finite enumerations illustrates that the theory of finite enumerations with operation $\ominus$ is based on that of enumeration, elements, natural numbers with addition and subtraction, and basic natural numbers (Figure 5.4).

Thus, the entire theory consists of a structured set of small theories with a few axioms only. A selection method which is presented in [Reif and Schellhorn, 1998] can take advantage of such a hierarchical structure. By selecting

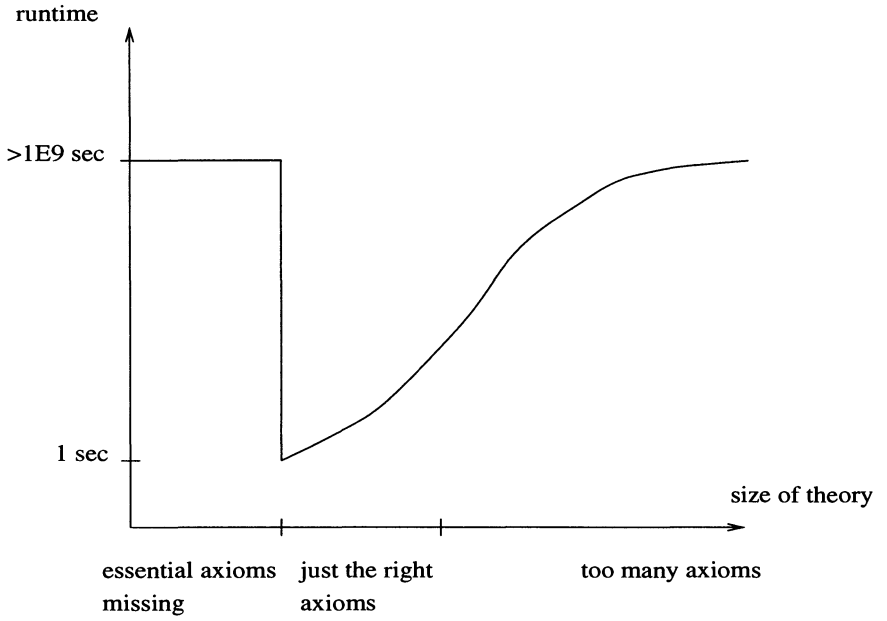

Fig. 5.3. Run-time over number of axioms

only those axioms from subtheories which are dependent from the theory of the theorem, a practically good preselection can be accomplished. Much

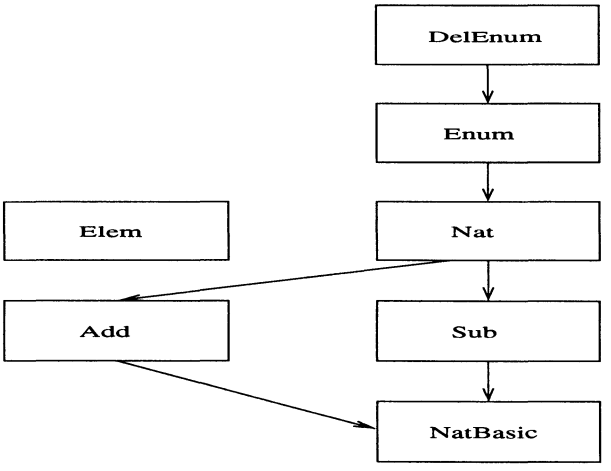

Fig. 5.4. Hierarchy of theories for enumeration types with $\ominus$ (from [Reif and Schellhorn, 1998], p. 227)

more complicated to handle is a large flat ("homogeneous") theory with many axioms. Here, no simple information is provided which axioms might be necessary for the current proof task.

A further issue concerns the *structure* of the axioms and the *minimality* of the set of axioms. For each theory, there exists a minimal set of axioms which is sufficient to completely and consistently describe the theory. If, however, such a minimal set of axioms is optimal for the automated theorem prover very much depends on the application and the prover. In many cases, it might be feasible to add additional axioms to the theory. These can be regarded as lemmata, because they can be derived from the minimal set. However, they can abbreviate a proof substantially and thus lead to shorter proof times.

5.8 Handling of Non-Theorems

A typical task for an automated theorem prover for first-order logic is to *find* a proof for the given formula. If a complete proof procedure has been selected, the prover will eventually stop, if the formula is valid (or its negation unsatisfiable). However, no assumptions can be made if the formula is *not valid.* Due to the semi-decidability of first-order logic, the prover may or may not terminate on the given formula. In the classical field of automated theorem proving, this behavior is implicitly accepted — most benchmarks are in fact valid formulas. However, when we look at applications, things are quite different. Just imagine that you want to verify a piece of code against its specification. If either of them contains errors, the entire proof task $Impl \Leftrightarrow Spec$ (or subtasks thereof) is not valid. In particular during the early stages of a verification or refinement process, such situations are likely to occur.

An even higher probability of the occurrence of formulas which are not valid occurs during logic-based component retrieval: if a library component does not match the given query, the corresponding proof task cannot be valid. In the general case, one will end up with very few (or only one) valid proof tasks and many (size of the entire library, often 10s of thousands) non-valid proof tasks. Here, automatic detection of invalid formulas which we will in the following call *non-theorems* is of great importance.

For practical applications, the user requires that an automated theorem prover recognizes at least trivial and obvious non-theorems as such. Whereas for some applications (like logic-based component retrieval) the answer FALSE is sufficient, in other cases, the user expects some sort of *feedback* about what went wrong. Invalidity of a formula could be caused by errors in the formula, missing axioms or assumptions. In Section 7.5 we will describe several practical approaches for handling non-theorems (counterexample generation, meta-programming, and simplification) and will discuss their advantages and shortcomings.

5.9 Control

Finally, when the entire proof task is prepared for the automated prover, the prover itself is started to search for a proof (or a counterexample). Because of severe time and resource restrictions, the prover should find a proof (if there exists one, of course) as fast as possible. The control of the prover, however is a very hard problem which we will discuss in more detail below. One one hand, the run-time behavior of a prover strongly depends on the proof obligation itself. Furthermore, most automated theorem provers provide a *large* numbers of parameters which influence the search of the system. This often leads to the situation that a user of a prover has to actually ask the developer of the system to select a good parameter setting for a given problem.

When applying an automated theorem prover, it must conform to the following important requirements:

1. *Usability*: the control of the prover itself must either be hidden from the user, or it must be configured in such a way that the user needs not have detailed knowledge about the automated theorem prover itself (e.g., calculus, proof procedure).
2. The prover should behave *smooth* with respect to somewhat similar proof tasks. This is in particular important, if the application system expects some "reliability" in the answers of the ATP. A smooth behavior could e.g., characterized by: small proof obligations should be solved with little resources.
3. For N given proof tasks and a (time) limit T, the prover should try to solve the proof tasks as early as possible. This means that we are more interested in short answer times than in finding as many proofs as possible.
4. For N given proof tasks and a (time) limit T, the prover should solve as many proof tasks as possible. Here, the run-times to find a proof (t_p) can be higher. ("We are slow, but we get more tasks solved")

In general, the last two requirements contradict each other. Figure 5.5 depicts typical idealized behaviors of a theorem prover. Ideally (curve marked (a)), some proof tasks can be solved with little resources. Then the total number of solved proof tasks with $t_p < T$ increases. The more resources you spend, less new proof obligations can be solved. This is due to the fact that the search space in general grows much faster than the number of proofs in that search space. Curve (b) marks our requirement 3 which aims at short answer times. In particular, in combination with interactive systems where the user actually has to wait until the ATP finishes, this behavior is desirable. Finally curve (c) reflects requirement 4. Here, for short run-times the performance can be lower than with (a) or (b), but for longer run-times, (c) outperforms (b).

Figure 5.6 shows the actual behavior for a number of well-known provers with a fixed parameter setting. The figure shows the number of proof tasks

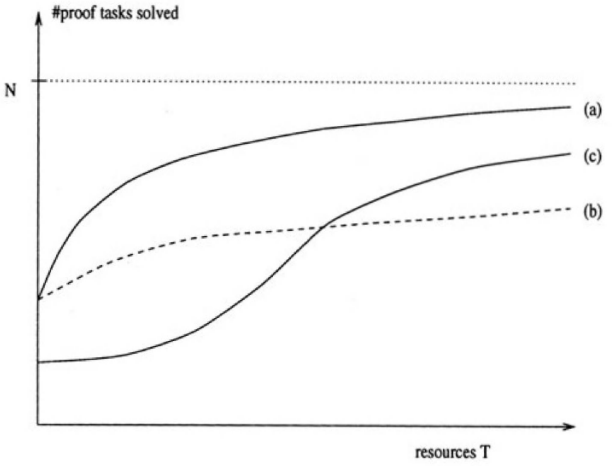

Fig. 5.5. The typical number of proof tasks solved with resources $t_p < T$. (a) ideal case, (b) aims at short answer times, (c) at general high performance.

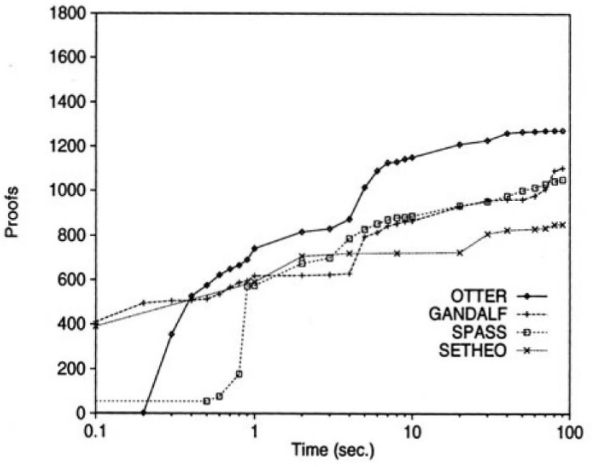

Fig. 5.6. Number of solved proof tasks with runtime $t < T$ for various well-known theorem provers (from [Fischer, 2001])

solved over run-time for each theorem prover on a large number of proof tasks from software reuse (Section 6.3). In general, however, there does not exist a uniform way of controlling an automated prover such that it fulfills all requirements set up above. In the following two subsections, we will have a closer look at the influence, the proof obligation and the parameter setting of the prover has on the run-time behavior. In Section 7.6, we will develop a method which is able to fulfill these requirements in a reasonably good way.

5.9.1 Size of the Search Space

The search space to be traversed for a proof obligation is spanned by the formula and the proof calculus. The ATP's proof procedure then determines the way it is explored (e.g., depth-first-search, breadth-first-search, min-max-search). For first-order predicate logic the size of the search space is highly exponential and very non-deterministic. Therefore, it is not possible for the search algorithm to determine the "distance" from the proof (in the kind of "5 more minutes to go before the proof is found"). This is in contrast to many other search and optimization problems, where the distance of the current state to the final state can be estimated in a reasonable way.

An a-priori estimation of the entire size of the search space, given resource limits (e.g., maximal depth of search tree) is much to inaccurate, because of the ATP's optimizations for pruning the search space. These become active only during the search and require knowledge about previous states (e.g., an optimization to avoid the same failing goals).

Example 5.9.1. The following example illustrates this behavior. Table 5.4 shows the size of the search space over the limiting bounds. The figures of the example have been obtained with SETHEO. The example is MSC001-1 (originally known as "Nonobviousness" [Pelletier and Rudnicki, 1986]). The bound is the maximal depth of the tableau (A-literal depth), the size of the search space is measured in attempted model elimination inference steps. A similar behavior can be observed with any theorem prover.

Measure	Search space (# of attempted inference steps)			
depth-bound =	3	4	5	6
full search	41	533	9.8×10^4	$> 10^8$
unification	30	127	3149	1.3×10^6
opt 1	30	108	703	5037
opt 2	30	127	359	1321

Table 5.4. Size of the search space for a proof task w.r.t. several values of the depth bound and various optimization levels (SETHEO)

The first line reflects the full size of the search space. Its size could be determined analytically, given the characteristics of the formula. In the next row (labeled "unification"), we explore the search space, performing unification eagerly, i.e., as soon as an inference step is tried, we perform unification. In case, the unification fails, this part of the search tree is chopped off. Subsequent rows shows the size of the search space, spanned by the formula and calculus, using various levels of optimization[7]. The huge difference between

[7] Optimization "opt1" enforces regular model elimination tableaux (for details see Section 3.3), and "opt2" additionally avoids repeated attempts to solve the same (failed) subgoal over and over again (anti-lemmata).

the analytic estimate and useful figures (last row) indicates that an a-priori estimation of the size of the search space, spanned by a given proof task does not yield reasonable results.

5.9.2 Influence of Parameters

Most modern theorem provers feature a large variety of parameters to control their behavior. They can be entered as command-line parameters (e.g., SETHEO), or as specific control statements in the formula-file (e.g., OTTER). Their high number makes it hard to determine good settings, even for specialists in the field. Whereas binary settings can be checked systematically, parameters with continuous values (e.g., [0...1]) are much more complicated to handle. Typical parameters belonging to this group are assigned weights to symbols of formulas or numerical parameters for the calculation of resource bounds or orderings. The following example shows a contour-plot of the run-time over two independent continuous parameters for one example. These figures have been obtained by the parallel system SiCoTHEO-CBC (see Section 7.6.2) and TPTP example GRP001-1. Contour-lines connect points with equal run-times. Short run-times are dark-colored, areas of long run-times are in light color.

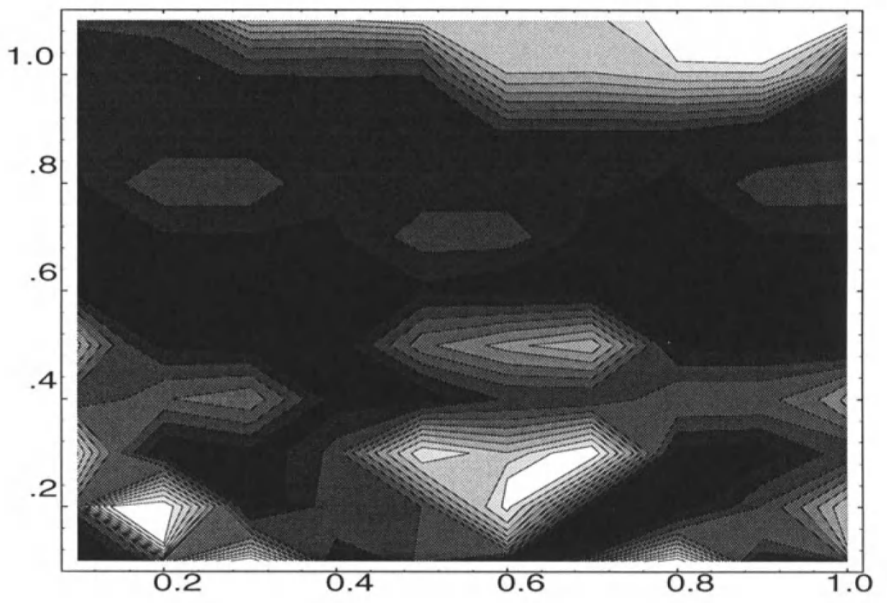

Fig. 5.7. Contour-plot of run-time (dark = short run-time) over two parameters (stepsize=0.1) for one example obtained by the parallel prover SiCoTHEO

This figure shows a "swiss-cheese" like pattern: small changes in parameter values can lead to drastic changes in the behavior of the system ("falling into holes"), and one cannot detect regular patterns of this behavior. These parameters, as most others, do not exhibit a linear (or even monotonous) behavior. However, the behavior is not fully chaotic, as the enlargement in Figure 5.8 shows. Here, some regular patterns can be detected, but they are hard to interpret.

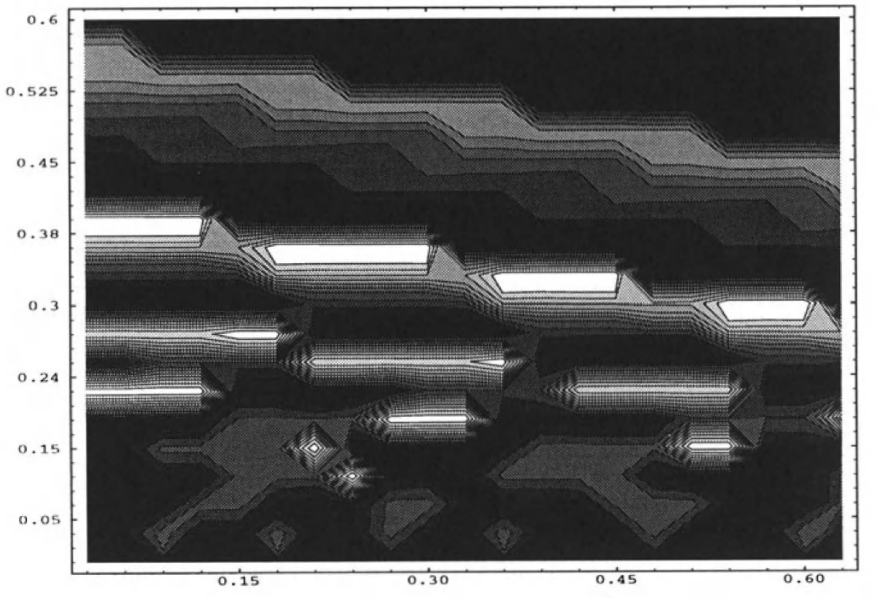

Fig. 5.8. Enlarged contour-plot of run-time (dark = short run-time) over two parameters (step-size=0.03) for one example

On a higher level, similar experiments could be made with the parameter "prover": this results in the question which automated theorem prover should be used for which kinds of proof obligations. Several competitions between automated theorem provers (e.g., CASC-13 [Sutcliffe and Suttner, 1997], CASC-14 [Suttner and Sutcliffe, 1998], or CASC-15 [Suttner and Sutcliffe,

1999]) and results obtained from our software reuse case-study (Section 6.3.5) illustrates this problem.

5.9.3 Influence of the Formula

When dealing with a number of proof obligations, one tends to attribute "similar" formulas with a similar behavior in search. This, however, is not true. Even the slightest change in a formula can result in totally different behaviors. The most striking example probably is adding a single $\neg$ to a formula and thus changing its validity (e.g., from provable to invalid).

Example 5.9.2. We illustrate this behavior with large numbers of proof obligations from various domains: Figure 5.9 relates the syntactical distance (a Hamming distance) of formulas from an experiment in software reuse (see Section 6.3) to their run-time ratio (obtained by SETHEO). Ideally, small changes in the formula should be reflected by similar run-times (logarithm of run-time ratio is close to 0). Then, their syntactical structure could be used to predict their run-time behavior. Unfortunately, however, even rather homogeneous sets of proof tasks (as in our case) do not exhibit such properties.

A second set of data was from the theory of ordered fields [Dräger, 1993]. Figure 5.10 shows a similar situation: even small syntactical differences in the formulas lead to large changes in their run-time behavior. The two separated "clouds" are coming in from two different ways of specifying the proof obligations. By using a different set of predicates and representation of terms, the syntactical distance between two formulas from each class is rather high.

5.10 Additional Requirements

If an automated theorem prover is to be applied successfully, it also must fulfill requirements concerning pragmatic issues like availability, support, stability, etc. In the following, we will shortly discuss these requirements.

5.10.1 Software Engineering Requirements

Important requirements for practical applicability are not specific to automated theorem provers or even inference systems. They concern features of the automated theorem prover as a large and complicated piece of software. One of the key requirements concerns *stability* and *robustness* of the system. A system should be stable, i.e., it should exhibit the same behavior for the same input, regardless of the situation where it is invoked. Typical problems with stability are errors and weaknesses in memory management, handling of files, and "nondeterministic errors". The latter errors only occur once in

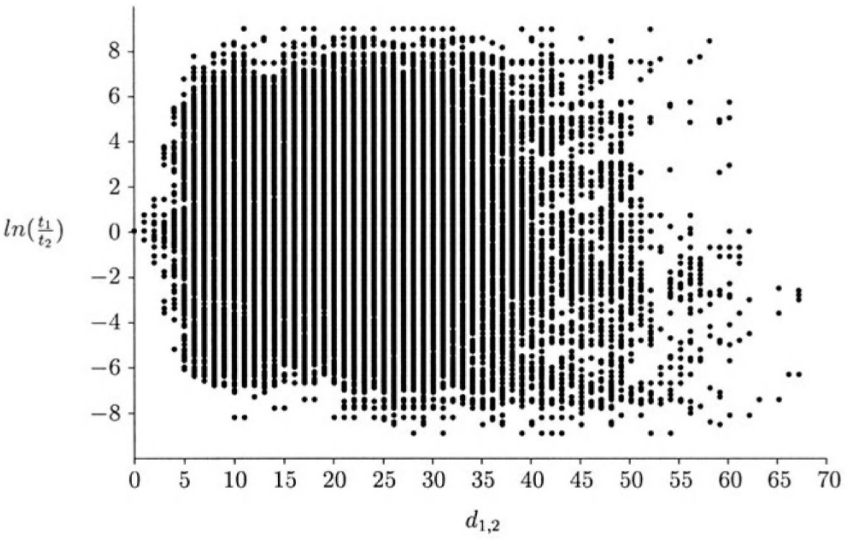

Fig. 5.9. Ratio of run-times over the Hamming distance $(0, \ldots, 65)$ between two proof tasks (SETHEO, example proof tasks from case study in software reuse)

a while and are in particular hard to detect. They can be removed only by extensive testing.

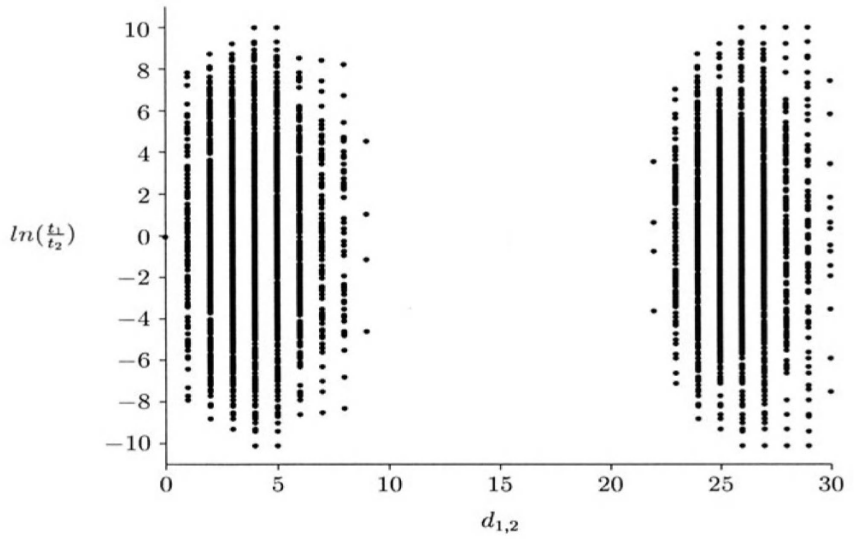

Fig. 5.10. Ratio of run-times over the Hamming distance between two examples (SETHEO, theory of fields)

A robust system is able to correctly cope with proof obligations, even if they contain errors. In particular in applications where manual input is possible, a high amount of robustness is required. Typical issues are:

- handling of erroneous proof tasks: if a formula is satisfiable or contains errors (like typing errors, invalid symbols, etc), the prover is in charge of making an elaborate error handling. Errors must be detected and reported appropriately. Furthermore, the system should not crash on any given proof task. E.g., for several years, early versions of SETHEO crashed when it was given a formula without an entry-clause.
- names and syntax: most theorem provers use certain reserved names which cannot be used as symbols within a proof task. This is acceptable, as long as these are clearly documented. The same holds for implementation-specific restrictions, like the maximal length of a symbol.
- handling of excessively large proof tasks: when presented an extremely large proof task, most theorem provers react in a rather unacceptable way. They try to process this task and, in most cases, use more system resources than available. This can lead to the case that so much memory is allocated that a workstation essentially gets inoperable, or they generate huge intermediate files which can cause problems with over-full file systems. Therefore, all theorem provers should have means to reject a proof task, if it is, compared to the given resources, too large. This can be realized by limiting file sizes, computation time, amount of memory, etc.

5.10.2 Documentation and Support

A complex piece of software like an automated theorem prover can be applied in practice only if there is enough and continuous support for it. Support does not only concern availability of the system (e.g., via ftp or WWW), but also documentation. This documentation (e.g., a user's manual) should cover all important details of the system and its implementation. Unfortunately, due to restricted human resources, quite often such documentation does not exist or is out of date.

Releases of a new version of a theorem prover must be accompanied by information about the changes and enhancements. In particular, modifications concerning the interface (like input or output language, command-line parameters) must be documented clearly. Otherwise, it is likely that old versions of the prover are used in applications, simply because the new one might (and often does) exhibit unexpected behavior.

6. Case Studies

In this chapter, we present three case studies concerning the application of the automated theorem prover SETHEO. The case studies have been selected from application domains within the area of Software Engineering. Their purpose is to show, how the specific techniques and methods adapted and developed around the "naked" automated prover work, and how they influence the applicability of such a system.

We will take case studies from three different areas in the Software Engineering process, namely: verification, refinement of specifications, and software reuse. From these large areas, "niches" have been determined, where we will show that the application of automated theorem provers is in the scope of current technology and helpful for making tools more attractive for the user.

This requirement, however, poses severe restrictions on where to look. On one hand, the proof tasks must not be too complicated, otherwise, they cannot be handled by any current or near-future automated system. On the other hand, areas, where a larger number of (similar) proof tasks show up, are of interest. Furthermore, the topic should be of widespread interest. Thus the following three topics have been selected to be of particular interest: verification of authentication protocols, refinement of protocol specifications (also a specific task within verification), and logic-based component retrieval.

With the huge growth of distributed software applications (like local area networks, distributed programming (e.g., PVM), the internet, and so on), communication and security protocols are becoming more and more important. For widespread applications like tele-banking, e-commerce, or teleconferencing, a proper identification of the communication partners and secure transfer of data is of great importance. Using an *authentication protocol*, two (or more) partners are able to identify themselves to the other(s), and can establish a private communication. Such a protocol must ensure that no intruder can listen to that communication, or provide fake data. Since such protocols are often being used in highly sensitive areas, their correct working must be guaranteed. However, most security protocols used in practice initially contained errors. Thus, formal proof of their properties is an important requirement. In order to accomplish this, a variety of approaches and (mostly interactive) systems have been developed (cf. [Meadows, 1995;

Geiger, 1995] for an overview). In this application, a high level of abstraction reduces the complexity of the proof tasks and makes automatic processing feasible. In Section 6.1, we will present an approach based upon an automated theorem prover.

A good *communication protocol* must allow for fast and reliable communication even when using non-reliable media. Formal methods are quite common-place in this area, as e.g., the standard OSI/ISO layer model shows. All layers and the interfaces between them set up a huge "playground" for application of formal methods. A major reason, why protocols seem to be well suited for automatic processing is their relatively simple internal structure. Whereas normal programs use a variety of complicated, user-defined, often recursive data structures (requiring induction), the data structures within a protocol specification are usually rather simple. For example, a data package to be transferred consists of opaque user data (which can be abstracted into a single piece of data), and additional control information, normally just a couple of integer values. Therefore, proof tasks tend to have a more uniform and structured form than those, originating from arbitrary verification tasks. This opens up a chance for successful application of automated proving techniques (Section 6.2).

Logic-based component retrieval is a particular technique used within the area of software reuse. Its goal is to retrieve one or more components from a database of components, given a target specification (i.e., a specification of what the user wants to have). This specification is then matched against specifications of the components in the data base, giving rise to large numbers of proof tasks. Time constraints and the required level of automatic processing for such proof tasks are high; only a few seconds are available before the user expects the answer. Therefore, reuse systems have to apply elaborate heuristics and filters to find an optimum between the two parameters: recall and precision. Ideally, all stored components which are exactly matching are retrieved, yielding a high recall and good precision. All components which cannot be found by the system decrease the recall, whereas components which are retrieved, but do not match, decrease the precision. In contrast to handling of proof tasks in other areas (e.g., verification), where soundness and completeness is *required*, here, a good compromise must be found, leaving ample space for developing techniques for estimation and filtering. This case study will be described in detail in Section 6.3.

6.1 Verification of Authentication Protocols

6.1.1 Introduction

Authentication Protocols. *Authentication protocols*[1] are used when secure services are to be provided between two or more partners. In particular they are used to establish the correct identity of the communication partners and to exchange secret information (e.g., encryption keys). Such protocols have to guarantee that other persons (usually called "intruders") cannot gain access to secret data, obtain information from transmitted data, and are not able to disturb the flow of information.

Usually, an authentication protocol is running between two or more communication partners called *principals*. The protocol itself consists of a number of *messages*, each of which can contain a variety of information in encrypted or plain form: e.g., names of principals, time-stamps (*nonces*) to ensure freshness, secret or public keys.

Example 6.1.1. The *Kerberos* protocol (e.g., [Burrows *et al.*, 1989]) establishes a shared key K_{ab} between two principals pA "Bob", pB "Alice" with the help of an authentication server pS. This protocol has been developed at MIT within the Athena project [Moormans and Belville, 1990]. The Kerberos protocol consists of 4 messages. A successful protocol run is shown in Figure 6.1. With the first message, pA contacts the server pS, sending its own name and that of the proposed communication partner pB. In the second message server pS provides pA with the session key K_{ab}, and a certificate *Cert* conveying the session key and pA's identity (encrypted by the key K_{bs}). The entire message is encrypted by pA's key K_{as} in such a way that only pA can decode and read that message. Then, the latter part of the message is sent from pA to pB, along with a time-stamp (nonce T_a), issued by pA. Finally, pB acknowledges the establishment of secure communication by incrementing the time-stamp T_a and sending it back to pA (encrypted now by the mutual session key K_{ab}). After that step, a secure communication with the session key K_{ab} is established. This session key now can be used between pA and pB for arbitrary purposes, e.g., exchange of secret data.

This protocol assumes that pS is a trusted principal which is capable of generating secret and reliable session keys K_{ab}. pS is usually called a *certificate authority*(CA). Such authorities can be specific (and government-controlled) companies or specifically set-up computers in a trusted environment (e.g., an office with restricted access).

Analysis of Authentication Protocols. In the last few years, cryptographic protocols have gained more and more importance. For many distributed applications, large amounts of money (e.g., tele-banking) or personal data (e.g., cellular phones) are at stake. Thus, *safety* and *security* have

[1] Sometimes, these protocols are also called cryptographic protocols.

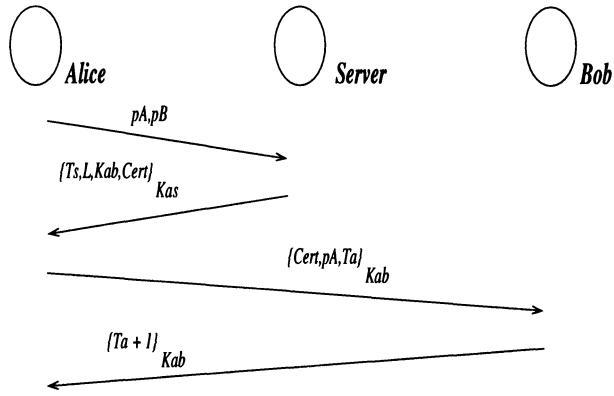

Fig. 6.1. The Kerberos protocol. The certificate is $Cert = \{T_s, L, K_{ab}, pA\}_{K_{bs}}$.

become very critical issues (cf. [Neumann, 1995]) and actually have to be *guaranteed* by vendors. The following components are of central importance:

1. positive identification of the communication partners (e.g., by smart-cards, finger prints, scanning of eye-background),
2. encryption algorithms (e.g., DES, RC4, IDEA),
3. the authentication protocol itself (i.e., which messages are exchanged in which order between the principals), and
4. the environment (e.g., criminal access to keys, reliability of the communication partners).

In order to guarantee security, all 4 components must be working correctly and reliably. However, as recent incidents show (see Example 6.1.2) this is not always the case, causing large losses in money and customer's confidence. Each of the above components can be analyzed separately. Therefore, we are able to restrict ourselves to the analysis of the authentication protocols (point 3). During the development of an authentication protocol many and hard to detect errors occur which require an exact analysis.

Example 6.1.2. Recent incidents show the vulnerability of security protocols on all four components:

1. The actual user's identity is (not yet) checked at ATM machines. So, any criminal with a (stolen or copied) card and the PIN is able to retrieve money. Checking can be done using fingerprints, optical face recognition, or recognition of eye-background.
2. Weak or buggy encryption algorithms are a great risk, because they can be broken with little effort. Examples are the erroneous algorithm in D2-Smart cards [Der Spiegel, 1998] used in cellular phones, and small

encryption keys as forced by the US export restriction (40 bits vs 128 bits)[2].

3. Most authentication protocols initially had bugs [Burrows *et al.*, 1989; Burrows *et al.*, 1990].
4. Security easily can be compromised if passwords or keys are given away or stored on the hard disk of a computer where they might be read remotely by intruders (as happened with German users of an ISP (t-online) [c't98, 1998]).

A *provable* guarantee of security and safety properties can only be given if *formal methods* are used during the design and analysis of the protocol. Therefore, many methods and tools for the analysis and verification of authentication protocols have been studied and developed. For an overview see, e.g. [Geiger, 1995; Meadows, 1995]. In order to find weak points (or rather their absence) in such a protocol, typical intruder scenarios can be simulated (cf. the interrogator system [Millen *et al.*, 1987]). Methods for formal verification have been developed since about 1989. The most well-known approach in that area is presented in [Burrows *et al.*, 1989]. The authors have proposed a modal logic of *belief*, called BAN logic (named after the initials of the authors). Several problems with the BAN logic (e.g., with respect to modeling intruders and specifying confidentiality properties) and a missing denotational semantics has led to many extensions and modifications of this logic, e.g., GNY [Gong *et al.*, 1990], SVO [Syverson and van Oorschot, 1994], or AUTLOG [Kessler and Wedel, 1994].

Several approaches with computer-supported verification of authentication protocols using one of these logics have been reported: AUTLOG [Kessler and Wedel, 1994] uses a PROLOG system with forward reasoning, whereas Brackin [Brackin, 1996] uses the higher-order theorem prover HOL to perform reasoning with the BAN logic. Craigen and Saaltink [Craigen and Saaltink, 1996] transform the BAN logic into first-order logic using an approach similar to our own. The prover EVES is then used to perform reasoning. In order to improve the level of automatic processing, a specific forward reasoner has been added.

In [Kindred and Wing, 1996] protocol runs and intruder scenarios are modeled as finite-state systems. These are explored by symbolic model checking. However, this approach cannot reflect — due to its finiteness — all the possible ways an intrusion can occur. This difficulty can be overcome with an approach based on communicating sequential processes (CSP). It is capable of modeling a potentially infinite number of interleaving protocol runs. [Paulson, 1997a; Paulson, 1997b] uses Isabelle to interactively show security and safety properties of authentication protocols.

[2] Recently, even 128 bit encryption could be cracked in less than 3 days with special parallel hardware [Electronic Frontier Foundation, 1998].

6.1.2 The Application

We apply the automated theorem prover SETHEO to *automatically* verify theorems formulated in the BAN and AUTLOG logic.

The BAN Logic. The BAN logic [Burrows *et al.*, 1989] is a multi-sorted modal logic of belief with the basic sorts: *principal*, *encryption key*, and *formula*. Principals are denoted by A, B, P, Q, S and pA, pB, pS; formulas as X, Y. Encryption keys (shared between two principals) are written as K_{ab}, K_{as}, K_{ba}. Public keys are K_a, K_b, K_c, whereas $K_a^{-1}, \ldots$ denote the corresponding secret keys. Nonces (e.g., time-stamps) are T, T_a, T_b. The main operator of that logic is belief: P **believes** X (or $P \models X$) means that principal P believes a statement X. This also means that P acts as if X was true. This operator and other BAN operators (see Table 6.1) may be connected via "$\wedge$" to construct new formulas. For a complete list of operators and a detailed description see [Burrows *et al.*, 1989] or [Burrows *et al.*, 1990].

Shorthand	Description	Shorthand	Description
$P \models X$	P **believes** X	$\{X\}_K$	X encrypted with key K
$P \triangleleft X$	P **sees** X	$P \overset{K}{\leftrightarrow} Q$	comm. via shared key K
$P \mid\sim X$	P **said** X	$P \overset{K}{\rightleftharpoons} Q$	comm. via secret key K
$P \Rightarrow X$	P **controls** X	$\overset{K}{\mapsto} Q$	comm. via public key K
$\#X$	**fresh**(X)	$\langle X \rangle_Y$	msg combined of X and Y
K^{-1}	inverse of K	$\{M_1, \ldots, M_n\}$	msg consists of $M_1, \ldots, M_n$

Table 6.1. Operators and function symbols of the BAN logic. P, Q are principals, K is a key, and X, Y, M_i are formulas.

MM_{sk}	message-meaning	$\dfrac{P \models P \overset{K}{\leftrightarrow} Q \qquad P \triangleleft \{X\}_K}{P \models Q \mid\sim X}$
NV	nonce-verification	$\dfrac{P \models \#X \qquad P \models Q \mid\sim X}{P \models Q \models X}$
JD	jurisdiction	$\dfrac{P \models Q \Rightarrow X \qquad P \models Q \models X}{P \models X}$
SY	symmetry 1,2	$\dfrac{P \models Q \overset{K}{\leftrightarrow} R}{P \models R \overset{K}{\leftrightarrow} Q} \qquad \dfrac{P \models Q \overset{K}{\rightleftharpoons} R}{P \models R \overset{K}{\rightleftharpoons} Q}$

Table 6.2. Some BAN inference rules

The BAN logic as presented in [Burrows *et al.*, 1989] is defined via a list of logical postulates (or inference rules). The most important inference rules are shown in Table 6.2. The *message-meaning* rule (MM_{sk}, Table 6.2) concerns

the interpretation of a message. As soon as a principal P believes that it communicates safely with some other principal Q using encrypted messages (with shared key K), and such a message arrived ($P \triangleleft X$), we may infer that P believes that Q has actually sent the message X[3].

Nonce-verification: Nonces are specific messages generated by a principal to guarantee the freshness of a message. The nonce-verification rule (NV) states that a message is "recent", i.e., that the sender (Q) still believes in it when the corresponding nonce X is fresh ($\#X$). The *jurisdiction* rule states that if P believes that Q has jurisdiction (control) over X, then P trusts Q on the truth of X[4]. Two inference rules (which are not defined in [Burrows *et al.*, 1989], but used there) define the symmetry of communication (SY) via shared keys and shared secrets.

Several inference rules concern the "packing" and "unpacking" of messages which consist of more than one atomic message. Furthermore, rules exist that a principal can decipher messages, for which the appropriate keys are known. For a complete list of these inference rules see [Burrows *et al.*, 1989].

In general, an authentication protocol is specified as a finite sequence of messages which are being transmitted between the principals, usually written as (principal P sends a message to principal Q):

$$P \rightarrow Q : \text{message}$$

For analysis with BAN logic, all messages of the protocol must be *idealized*. This means that non-encrypted parts of a message are not included in the formulas. This means that all messages are of the form $\{X\}_K$ or sequences thereof.

If an idealized message $P \overset{id}{\rightarrow} Q : X$ has been received, the formula $Q \triangleleft X$ (Q **sees** X) holds. Thus, a list of such formulas (one for each message) comprises the input specification of a protocol in BAN. Additionally, assumptions concerning the principals and the protocol must be made.

Example 6.1.3. All messages for the Kerberos protocol and their idealization are shown in Table 6.3 (adapted from [Burrows *et al.*, 1989]). Message 1 does not contain any encrypted data. Therefore, it is skipped in the idealization. Furthermore, the sequence number L is omitted and the session key K_{ab} is idealized to the formula: pA can communicate with pB using K_{ab}.

Example 6.1.4. For the analysis of the Kerberos protocol a number of assumptions had to be made (again from [Burrows *et al.*, 1989]). They are listed in Table 6.4 The assumptions in the first row express that principals pA and pB can communicate with the trusted server pS, using the keys

[3] For public keys or shared secret keys similar inference rules are defined.

[4] [Burrows *et al.*, 1989], p. 3

Msg#	Kerberos protocol	idealized Kerberos protocol
1	$pA \to pS : pA, pB$	—
2	$pS \to pA : \{T_s, L, K_{ab}, pB,$ $\{T_s, L, K_{ab}, pA\}_{K_{bs}}\}_{K_{as}}$	$pS \to pA : \{T_s, pA \overset{K_{ab}}{\leftrightarrow} pB,$ $\{T_s, pA \overset{K_{ab}}{\leftrightarrow} pB\}_{K_{bs}}\}_{K_{as}}$
3	$pA \to pB : \{T_s, L, K_{ab}, pA\}_{K_{bs}},$ $\{pA, T_a\}_{K_{ab}}$	$pA \to pB : \{T_s, pA \overset{K_{ab}}{\leftrightarrow} pB\}_{K_{bs}},$ $\{T_a, pA \overset{K_{ab}}{\leftrightarrow} pB\}_{K_{ab}}$
4	$pB \to pA : \{T_a + 1\}_{K_{ab}}$	$pB \to pA : \{T_a, pA \overset{K_{ab}}{\leftrightarrow} pB\}_{K_{ab}}$

Table 6.3. The Kerberos Protocol: messages in the original and idealized format

$pA \models pA \overset{K_{as}}{\leftrightarrow} pS$	$pA \models \#T_s$	$pA \models (pS \models\!\!\rightarrow pA \overset{K}{\leftrightarrow} pB)$
$pB \models pB \overset{K_{bs}}{\leftrightarrow} pS$	$pB \models \#T_s$	$pB \models (pS \models\!\!\rightarrow pA \overset{K}{\leftrightarrow} pB)$
$pS \models pA \overset{K_{as}}{\leftrightarrow} pS$	$pB \models \#T_a$	
$pS \models pB \overset{K_{bs}}{\leftrightarrow} pS$	$pA \models \#T_a$ (†)	$pS \models pA \overset{K_{ab}}{\leftrightarrow} pB$

Table 6.4. Assumptions for analysis of the Kerberos protocol

K_{as}, K_{bs}. Beliefs about the freshness are given in the second row[5]. The formulas in the last row express that pA and pB believe that pS is actually a key server, and that pS is convinced that it generates "good" keys K_{ab} which are useful for communication between pA and pB.

The Proof Tasks. The theorems we want to prove in general depend on the protocol we are analyzing. However, there are default requirements for the services which an authentication protocol should provide. These are of such a form that the principals (pA and pB) believe that they properly communicate via a shared key K: $pA \models pA \overset{K}{\leftrightarrow} pB$ and $pB \models pA \overset{K}{\leftrightarrow} pB$, and the principals believe the beliefs of their respective partners:[6]

$$pB \models pA \models pA \overset{K}{\leftrightarrow} pB, pA \models pB \models pA \overset{K}{\leftrightarrow} pB$$

A summary of the proof task's characteristics are given in Table 6.5. Of particular importance for this kind of medium-sized and medium-difficult proof tasks is the required translation of the logic and the presentation of human-readable proofs on the source level (BAN logic or AUTLOG logic).

[5] The experiments with SETHEO revealed that one assumption essential for the proofs, $pA \models \#T_a$ (marked by a †) is not described in the literature. This fact has also been detected (independently) in [Craigen and Saaltink, 1996].

[6] The informal meaning of these two properties becomes clear, if you assume that pB is the bank and pA the customer in front of an ATM machine: before handing out the money the bank should be entitled to believe that the customer believes that he/she communicates safely with the bank (and therefore entered correct data).

category	value		
deep/shallow	Shallow	**Medium**	Deep
size & richness	**Small**	**Medium**	Large
answer-time	Short	**Medium**	Long
distance	Short	**Medium**	Long
extensions	finite messages		
validity	80%- 90% theorems		
answer	readable proof in BAN logic		
semantic info	Y	some	N

Table 6.5. Proof tasks of PIL/SETHEO: important characteristics

Example 6.1.5. As an example for a proof in the BAN logic, let us again consider the Kerberos protocol. We want to show that

$$pB \models pA \models pA \overset{K_{ab}}{\leftrightarrow} pB \tag{6.1}$$

holds, after messages 1–3 have arrived. Before message 3 has arrived, we already know from a previous proof task that

$$pB \models pA \overset{K_{ab}}{\leftrightarrow} pB. \tag{6.2}$$

By the inference rule "message-meaning" and with idealized message 3 (second part) of the protocol, we obtain

$$pB \models pA \mid\sim \{T_a, pA \overset{K_{ab}}{\leftrightarrow} pB\}_{K_{ab}}. \tag{6.3}$$

Since, by assumption $pB \mid\equiv \#T_a$ (Table 6.4), we have (if a part of a message is believed to be fresh, then the entire message is)

$$pB \models \#(\{T_a, pA \overset{K_{ab}}{\leftrightarrow} pB\}_{K_{ab}}). \tag{6.4}$$

Finally, by (6.3) and (6.4) and "nonce-verification", we can prove our theorem (6.1). q.e.d.

The AUTLOG Logic. In order to overcome several weaknesses of the BAN logic, the AUTLOG logic has been developed [Kessler and Wedel, 1994; Kindred and Wing, 1996]. In particular, the idealization of protocol messages which is required for an analysis with BAN logic makes a detection of many protocol errors impossible with that logic. AUTLOG introduces a new operator $\rho(X)$ (formula X is *recognized*). With the help of slightly modified and additional inference rule, idealization of protocol messages can be avoided and stronger properties can be shown. [Kessler and Wedel, 1994] propose an implementation of their logic in PROLOG, however, they indicated that in many cases the run-times were considerably too long.

6.1.3 Using the Automated Prover

The Architecture. Because no reasonable application system for the specification and analysis of authentication protocols had been available, the tool PIL/SETHEO [Wagner, 1997; Schumann, 1998; Dahn and Schumann, 1998; Eckert, 1998] was designed and implemented around the automated theorem prover SETHEO. The system architecture as shown in Figure 6.2 reflects main user requirements set up in this book:

- PIL/SETHEO allows to enter the protocol specification, the assumptions and the properties to be proven in a problem-oriented, albeit simple specification language.
- The output of PIL/SETHEO is a LaTeX document (see Example 6.1.10) containing a proof of the theorem in the formalism of the BAN or AUTLOG logic.
- The entire tool is running fully automatically and does not require any user interactions.

Thus, PIL/SETHEO can be used by persons without detailed knowledge about first-order theorem proving or the prover SETHEO. Hiding all evidence of the first-order theorem proving additionally makes the operation of the tool more elegant and user-friendly.

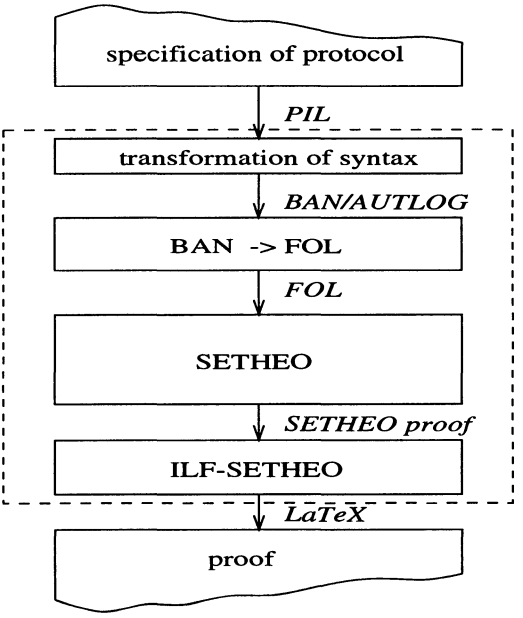

Fig. 6.2. Basic architecture of PIL/SETHEO. All steps in the dashed box are kept transparent to the user.

PIL/SETHEO takes a specification of the problem(s), given in the language PIL. This language is rather primitive, close to BAN logic and has been developed for this tool. PIL/SETHEO first transforms the syntax into an internal representation. Here, infix operators are replaced by corresponding prefix operators and all statements are transformed into formulas of the BAN logic. Since PIL is very close to BAN logic, this transformation is merely a syntactical one. With the directive: `use AUTLOG`, PIL/SETHEO can switch from BAN logic to AUTLOG logic. The input syntax for AUTLOG is a superset of that for the BAN logic, allowing uncrypted parts in messages and the additional operator ρ.

Example 6.1.6. Figure 6.3 contains the input specification for the Kerberos protocol in PIL. First, starting with `Objects`, all principals `A, B, S`, keys `K_a_b, K_a_s, K_b_s`, and the nonces `T_a, T_s` are declared. Please note that the sorts of the BAN logic are so weak that nonces have to be declared as general-purpose BAN formulas of sort `statement`. Variables (in this case, a shared key K) are declared as `variable sharedkey`. Assumptions are given next. The first line corresponds to the two assumptions $pA \models pA \overset{K}{\leftrightarrow}_{as} pS$ and $pS \models pA \overset{K}{\leftrightarrow}_{as} pS$. All other assumptions (as given in Table 6.4) are left out in this figure.

The three idealized messages (as given in Table 6.3) specify the protocol itself. Properties to be shown are introduced by the keyword `Conjectures`. Here 4 properties (which must hold after the first two, three or four messages have been received) must be shown. These give rise to four individual proof tasks.

```
Objects:                    // declaration of all objects
   principal A,B,S;
   sharedkey K_A_b, K_a_s, K_b_s;
   statement T_a, T_s;
   variable sharedkey K;
Assumptions:                // assumptions (abbreviated)
   (A,S) believes sharedkey K_a_s;
   (B,S) believes sharedkey (B,S,K_b_s);
   ...
Idealized Protocol:
   message 2: S -> A  {T_s, sharedkey (A,B,K_a_b),
          {T_s, sharedkey(A,B,K_a_b)}(K_b_s)}(K_a_s);
   message 3: A -> B  {T_s, sharedkey K_a_b}(K_b_s),
          {T_a,sharedkey K_a_b}(K_a_b);
   message 4: A <- B  {T_a, sharedkey K_a_b}(K_a_b);
Conjectures:                // conjectures (4 tasks)
   after message 2: A believes sharedkey K_a_b;
   after message 3: B believes sharedkey K_a_b;
                    B believes A believes sharedkey K_a_b;
   after message 4: A believes B believes sharedkey K_a_b;
```

Fig. 6.3. PIL specification of the Kerberos protocol and security properties

Flow-control for the entire tool (one input specification can result in a number of individual proof tasks) is done using the UNIX tool *make*. This utility ensures that all proof tasks eventually get processed and that the final LaTeX documents are generated.

Translation into First Order Logic. The BAN logic and the AUTLOG logic is defined by an alphabet of atoms (principals and keys), functions and operators, and inference rules as a Hilbert system. One way to transform such a logic into first-order predicate logic (FOL) is "simulation": on a meta-level, all formulas of BAN are represented as first-order *terms*, and the inference rules are specified as first-order formulas[7]. Hence, for the BAN logic we obtain the following transformation. A unary first-order predicate $holds(X)$ is defined with the meaning: $holds(X)$[8] is true, if and only if the BAN formula X is *derivable* from other BAN formulas using BAN inference rules.

Additionally, each BAN inference rule is transformed into a FOL formula. Given an inference rule of the form

$$\frac{\mathcal{F}_1 \ldots \mathcal{F}_n}{\mathcal{G}}$$

we obtain: $holds(\mathcal{F}_1) \wedge \ldots \wedge holds(\mathcal{F}_n) \rightarrow holds(\mathcal{G})$. Variables in the formulas of the inference rule are implicitly all-quantified in the translation.

Example 6.1.7. We translate the nonce-verification rule NV (Nonce Verification)

$$\frac{P \models \#X \qquad P \models Q \mid\sim X}{P \models Q \models X}$$

into

$$\forall P, Q, X : holds(P \models \#X) \wedge holds(P \models Q \mid\sim X) \rightarrow holds(P \models Q \models X).$$

The BAN formula $A \models \#T_s$ (A believes the freshness of nonce T_s) is translated into $holds(A \models \#T_s)$.

Then, in our approach, the entire input formula for SETHEO is of the form

$$\mathcal{A}_1 \wedge \ldots \wedge \mathcal{A}_n \wedge \mathcal{M}_1 \wedge \ldots \wedge \mathcal{M}_m \wedge \mathcal{R}_1 \wedge \ldots \wedge \mathcal{R}_k \rightarrow \mathcal{T}$$

where the $\mathcal{A}_i$ are the transformed assumptions, $\mathcal{R}_i$ are the transformed inference rules, $\mathcal{M}_i$ are the transformed messages of the idealized protocol, and $\mathcal{T}$ is the translated conjecture to be shown. This formula can easily be transformed into clausal normal form obtaining a set of Horn clauses.

[7] Since all BAN formulas are ground, and quantifiers (universal quantifiers if any) only occur on the outermost level, this transformation is safe.

[8] In shorthand, we write $\vdash X$.

Preprocessing and Sorts. Of course, our transformation has to make sure that the sorts of all objects of the BAN logic (principal, message, formula) are handled correctly. Due to the structure of the inference rules, however, this sort information can be neglected: given correctly sorted BAN formulas, all derivable formulas are automatically of the correct sort. Explicit handling of sorts, however, greatly facilitates user input. With its help, many user errors can be detected easily. Therefore, checking of sorts is done in the parsing module of PIL/SETHEO.

Auxiliary predicates. Within the BAN logic, a message can consist of a (finite) sequence of messages enclosed in braces {}. A notation of such sequences of variable length as well as predicates for accessing, breaking-up and constructing sequences has to be defined in first-order logic as *auxiliary predicates*. For this purpose, we have defined an additional function symbol *cons* with arity two, and a symbol *nil*, representing the empty sequence. A message $\{m_1, m_2, \ldots, m_n\}$ is represented as the term

$$cons(m_1, cons(m_2, \ldots cons(m_n, nil)\ldots)$$

Example 6.1.8. The auxiliary predicate for selecting a sub-message S of a message M: $S \sqsubseteq M$ is defined by the following three axioms:

$$nil \sqsubseteq nil \wedge$$
$$\forall F, T, R : T \sqsubseteq R \rightarrow cons(F, T) \sqsubseteq cons(F, R) \wedge$$
$$\forall F, T, R : T \sqsubseteq R \rightarrow T \sqsubseteq cons(F, R)$$

Preselection of Axioms. In PIL/SETHEO, we only add those inference rules which share operators with the conjecture, the messages of the protocol and the assumptions. These straightforward methods considerably reduce run-times of the theorem prover and contribute to the tool's practical usability.

Control. In this application, SETHEO is running with its default parameters. In the auxiliary predicates, we restrict the length of all sub-messages to be less than the maximal length of all messages occurring in the proof task.

Handling of Non-theorems. If a conjecture cannot be proven SETHEO usually will not terminate. In that case, SETHEO's time-out is reached and PIL/SETHEO notifies this situation as "not provable within XX seconds". From the user's point of view, this behavior is not satisfying. The user wants to have feedback on why no proof could be found. Therefore, PIL/SETHEO provides a "belief-generator": given the protocol's messages and assumptions, all possible beliefs can be generated by SETHEO[9]. In order not to be overwhelmed by the number of generated terms, the user can select specific classes of terms. For details of this technique see Section 7.5.3.

[9] Here, techniques of the DELTA iterator are used which will be described in Section 7.6.1.

Example 6.1.9. All terms which denote the belief of pA with respect to communications using the key K_{ab} can be generated by the following command. Patterns (wild-cards) are written as "_" in a PROLOG-style manner.

```
getbeliefs: pA believes sk(_,_,Kab).
```

Postprocessing. One of the major benefits of using a logic like BAN is that an analysis yields small, easy to understand proofs which give feedback to the protocol designer. Therefore, PIL/SETHEO is able to produce human-readable proofs on the level of the BAN (or AUTLOG) logic. We have used the postprocessor ILF-SETHEO (see Section 7.7) to convert the proofs found by SETHEO (i.e., Model Elimination tableaux) into proofs of the corresponding modal logic. The required transformation tables are generated by PIL/SETHEO. Thus, a LaTeX document can be automatically generated for each proof task. For details see Section 7.7.

Example 6.1.10. The fourth proof task of the Kerberos protocol

$$pB \models pA \models pA \overset{K_{ab}}{\leftrightarrow} pB$$

generates a proof as shown in the following text. This text has been generated by ILF-SETHEO, and no changes were applied to it before including the proof into this book[10].

Theorem 6.1.1. *query.*

Proof (by SETHEO). We show directly that

$$query. \tag{6.5}$$

Because of *message_3*

$$pB \triangleleft (\{\{\{T_S, pA \overset{K_{A,B}}{\leftrightarrow} pB\}\}_{K_{B,S}}, \{\{T_A, pA \overset{K_{A,B}}{\leftrightarrow} pB\}\}_{K_{A,B}}\}). \tag{6.6}$$

Because of *query_3*

$$query \leftarrow pB \models pA \models pA \overset{K_{A,B}}{\leftrightarrow} pB. \tag{6.7}$$

Because of *break_up_message*

$$P \models Q \models X \leftarrow X \sqsubseteq Y \wedge P \models Q \models Y. \tag{6.8}$$

Because of *nonce_verification*

$$P \models Q \models X \leftarrow P \models Q \sim X \wedge P \models \#X. \tag{6.9}$$

Because of *freshness* $P \models \#M_i \leftarrow P \models \#\{M1, \ldots, M_n\}$. Because of *assumption_11* $pB \models \#T_A$. Therefore

[10] A formula $\mathcal{A} \leftarrow \mathcal{B}$ is equivalent to $\mathcal{B} \rightarrow \mathcal{A}$ (PROLOG-style implication).

$$pB \models \#(\{T_A, pA \overset{K_{A,B}}{\leftrightarrow} pB\}). \tag{6.10}$$

Because of *message_meaning*

$$P \models Q \mid\sim X \leftarrow P \triangleleft \{X\}_K \wedge P \models Q \overset{K}{\leftrightarrow} P. \tag{6.11}$$

Because of *conjecture_2*

$$pB \models pA \overset{K_{A,B}}{\leftrightarrow} pB. \tag{6.12}$$

Because of *sees_components* $P \triangleleft M_i \leftarrow P \triangleleft \{M_1, \ldots, M_n\}$. Hence by (6.6) $pB \triangleleft \{\{T_A, pA \overset{K_{A,B}}{\leftrightarrow} pB\}\}_{K_{A,B}}$. Hence by (6.11) and by (6.12) $pB \models pA \mid\sim (\{T_A, pA \overset{K_{A,B}}{\leftrightarrow} pB\})$. Hence by (6.9) and by (6.10) $pB \models pA \models (\{T_A, pA \overset{K_{A,B}}{\leftrightarrow} pB\})$. Hence by (6.8) $\neg$ *query*. Hence by (6.7) *query*. Thus we have completed the proof of (6.5).

q.e.d.

6.1.4 Experiments and Results

PIL/SETHEO has been evaluated with a number of well-known protocols (Table 6.6). For each protocol with M messages, N proof tasks had to be processed. T is the run-time for all proof tasks and measured in seconds on a Sun Ultra 2. The generation of the LaTeX document usually takes less than 15 seconds. All specifications of the protocols and the conjectures have been taken from the literature [Burrows *et al.*, 1989; Burrows *et al.*, 1990; Kessler and Wedel, 1994].

BAN logic			
Protocol	M	N	T[seconds]
Kerberos	4	4	0.26
Andrew Secure RPC Handshake	4	6	0.10
Needham-Schroeder (NS)	5	5	0.93
NS-public keys	7	6	0.18
Otway-Rees	4	6	0.92
Wide-mouthed frog	2	2	0.10
Yahalom	4	12	14.67
CCITT-X509	3	2	0.12
AUTLOG-Logic			
Kerberos	4	4	13.80
ISO10181	2	2	0.54
RPC Handshake	4	6	8.69

Table 6.6. Experimental results with PIL/SETHEO. M is the number of messages of the protocol, N the number of proof tasks. T is the run-time to prove all N conjectures.

6.1.5 Assessment and Discussion

As demonstrated with Table 6.6 all proof tasks arising from the verification of well-known authentication protocols (using BAN or AUTLOG) could be shown fully automatically. PIL/SETHEO accepts problem-oriented input and produces human-readable proofs in the source logic. Although these logics can only be used to verify limited aspects of a protocol[11], PIL/SETHEO is a valuable tool especially for early protocol design phases.

The architecture or PIL/SETHEO reflects important requirements set up in the previous chapter. Based on the proof tasks' characteristics (Table 6.5) the translation of the modal source logic into first-order logic, the pre-selection of axioms, control of the prover (short answer times!), and postprocessing of the proofs by ILF-SETHEO has been of great importance. These issues (which will be dealt with in the next chapter) were actually the key-points to bring this application to success.

6.2 Verification of a Communication Protocol

6.2.1 Introduction

This case study describes the application of an automated theorem prover for the refinement of specifications. The prover SETHEO has been applied to proof tasks arising during the formal development of a communication protocol, the *Stenning* protocol. Its formal development and verification using the formal specification language FOCUS has been described in [Dendorfer and Weber, 1992b]. During the refinement of the specification the correctness of each step in terms of liveness and safety properties has to be shown on a formal and detailed level. These proof tasks, which were originally proved manually, have been replicated here by using an automated theorem prover to show the usefulness of automated theorem proving for such a task.

6.2.2 The Application

The Formal Method Focus. The typical development of a protocol specification in FOCUS [Dederichs *et al.*, 1993] starts from an abstract description of the services provided by the upper layer and those provided by the lower layer (here: level of transport medium). FOCUS covers all design steps from a non-constructive global specification down to an executable program. This case study focuses on the first steps, namely the refinement of the *global* requirements specification (a trace specification) into a *modular* requirements specification (also a trace specification). This refinement process is necessary since the global requirements of the upper layer cannot be separated into local requirements of the transmitter, the receiver, and the transport medium.

[11] For example, confidentiality cannot be modeled with the BAN logic.

A specification in *trace logic* states requirements for histories ("traces") of a distributed system as a sequence of *actions*. The description is in a way similar to linear-time temporal logic [Pnueli, 1986]. The protocol is defined by its actions, (e.g., sending or receiving data). The requirements of the protocol are given as a set of *liveness* and *safety* properties using predicate logic formulae, specifying "allowed" traces. The task of a protocol verification then is to show that these properties are not violated during any refinement step.

For the rest of this section, no detailed knowledge about FOCUS is necessary (see [Dendorfer and Weber, 1992a] for details). Below, we only give a list of important operators and their informal meaning. For traces t, u, v, and actions a, b, c

- $u \cdot t$ or $a \cdot t$ denotes concatenation of traces.
- $t \sqsubseteq u$ means "trace t is a prefix of u".
- $\#t$ denotes length of t. If the trace is infinite, $\#t = \infty$.
- $\langle a_0, \ldots, a_n \rangle$ is a trace consisting of actions $a_0, \ldots, a_n$.
- $t[k]$ denotes the k-th element of t (strong definition).
- $a©t$ denotes the filtered trace t that contains only actions a.
- a *in* t holds, if and only if action a occurs in trace t.
- $\langle d, k \rangle$ is a pair, consisting of a piece of data d and an integer k.

The Proof Tasks. The given proof tasks originate from [Dendorfer and Weber, 1992a]. The *Stenning Protocol* [Stenning, 1976] is a simple protocol which ensures reliable communication of upper layer data units (UDUs)[12] using an unreliable transport medium ("lower layer") as depicted in Figure 6.4.

This layer can lose packages ("lower layer data units", LDUs) or permute the sequence of packages. Reliable communication is accomplished by adding a unique sequence number to each UDU and by introducing acknowledge messages: each package is sent repeatedly, until an acknowledge message with the correct sequence number is received by the transmitter[13].

The actions which define the Stenning protocol are $snd(d)$ and $rec(d)$ (send/receive an upper layer data unit, $d \in UDU$), $get_R(x)$ and $put_T(x)$ (receive/send a lower layer data unit, $x \in LDU$), and $get_T(ack)$ and $put_R(ack)$ to receive (and send, respectively) an acknowledge message (over the transport medium). The set of all actions of a given kind is defined as $Snd := \{snd(d) | d \in UDU\}$. Rec, Get_R, Put_T are defined in a similar way. A *clock* action "$\surd$" is defined to be able to express behavior over time.

The FOCUS specification of the Stenning protocol starts with services which are provided by the upper layer and the lower layer. Those provided on the upper layer are specified in the liveness property UL and the safety

[12] These data units are seen as black boxes and are not further specified.

[13] With this technique, the Stenning protocol is a generalization of the well-known alternating-bit protocol [Bartlett *et al.*, 1969]. Furthermore, this protocol is equivalent to a sliding-window Kermit protocol [Huggins, 1995] with window-size one.

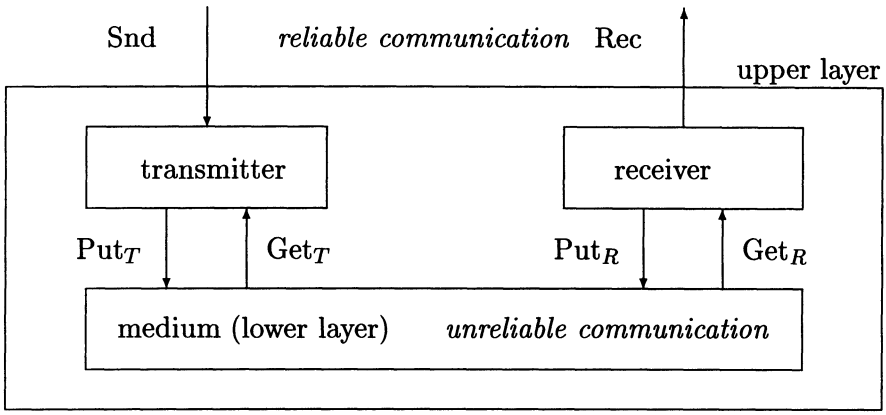

Fig. 6.4. The Stenning protocol (Fig. 1 of [Dendorfer and Weber, 1992b])

property US. All properties have been directly taken from [Dendorfer and Weber, 1992a] and are shown for reference in Table 6.7 and Table 6.8. UL means that the number of packages (UDU's) sent is equal to the number of received packages, i.e., no package has been lost, whereas US expresses that nothing "wrong" has been received. The properties LL^1, LL^2 and LS^1, LS^2 specify the behavior of the transport medium (in both directions): $LS^{1,2}$ states that if a data package (LDU) is received on one side, it must have been put previously onto the medium on the other side. However, it does not say that any sent package must be also received; losses are possible. $LL^{1,2}$ specifies that the medium is not broken forever (if we have sent an infinite number of packages, then an infinite number of packages is received after an infinite number of clock cycles).

The refinement of the specification with the goal of a modular requirements specification is performed in several steps. All requirements which are not local to the receiver or the transmitter, or which are not provided by the transport medium, have to be refined. In [Dendorfer and Weber, 1992a] this is accomplished by introduction of sequence numbers ($Step_1$), introduction of acknowledge messages ($Step_2$), and complete localization ($Step_3$). In the original paper [Dendorfer and Weber, 1992a], these proof obligations gave rise to several propositions and lemmata.

Each refinement step (Table 6.9) is defined as a conjunction of the liveness and safety properties from Table 6.7 and Table 6.8. Thus, a proof task has the general form:

$$Step_i(t) \Rightarrow Step_{i-1}(t)$$

Usually, one proof task is split up into several individual ones, obtaining a total of 10 proof tasks. The exact definition of all proof tasks is given in Table 6.13 below, their characteristics in Table 6.10.

$$
\begin{aligned}
UL(t) &\equiv \#(Rec©t) = \#(Snd©t)\\
LL^1(t) &\equiv (\#(\mathrm{put}_T(e)©t) = \infty \Rightarrow \#(\mathrm{get}_R(e)©t) = \infty) \wedge \#(\surd©t) = \infty\\
LL^2(t) &\equiv (\#(\mathrm{put}_R(e)©t) = \infty \Rightarrow \#(\mathrm{get}_T(e)©t) = \infty) \wedge \#(\surd©t) = \infty\\
L_1(t) &\equiv data(Snd©t) \sqsupseteq \langle d_0, \ldots, d_n\rangle \Rightarrow \forall k \leq n : \mathrm{put}_T(\langle k, d_k\rangle)\ in\ t\\
L_2(t) &\equiv \mathrm{put}_T(\langle k, d\rangle)\ in\ t \Rightarrow \mathrm{get}_R(\langle k, d\rangle)\ in\ t\\
L_3(t) &\equiv (\forall k \leq n : \mathrm{get}_R(\langle k, d_k\rangle)\ in\ t) \Rightarrow data(Rec©t) \sqsupseteq \langle d_0, \ldots, d_n\rangle\\
L_4(t) &\equiv \mathrm{put}_T(\langle k, d\rangle)\ in\ t \Rightarrow \mathrm{get}_T(ack(k))\ in\ t\\
L_5(t) &\equiv \#(Snd©t) > k \wedge \neg A(t,k) \wedge \#(\surd©t) = \infty \Rightarrow\\
&\qquad \exists j, d : \neg A(t,j) \wedge \#(\mathrm{put}_T(\langle j, d\rangle)©t) = \infty\\
&\qquad where A(t,k) \equiv \exists s : s \cdot \mathrm{get}_T(ack(k)) \sqsubseteq t \wedge \#(Snd©s) > k\\
L_6(t) &\equiv \#(\mathrm{get}_R(\langle k, d\rangle)©t) = \infty \Rightarrow \#(\mathrm{put}_R(ack(k))©t) = \infty
\end{aligned}
$$

Table 6.7. Liveness properties

$$
\begin{aligned}
US(t) &\equiv \forall s \sqsubseteq t : data(Rec©s) \sqsubseteq data(Snd©s)\\
LS^1(t) &\equiv \forall s \sqsubseteq t : \mathrm{get}_R(e)\ in\ s \Rightarrow \mathrm{put}_T(e)\ in\ s\\
LS^2(t) &\equiv \forall s \sqsubseteq t : \mathrm{get}_T(e)\ in\ s \Rightarrow \mathrm{put}_R(e)\ in\ s\\
S_1(t) &\equiv \forall s \sqsubseteq t : data(Rec©s) = \langle d_0, \ldots, d_n\rangle \Rightarrow\\
&\qquad \forall k \leq n : \mathrm{get}_R(\langle k, d_k\rangle)\ in\ s\\
S_2(t) &\equiv \forall s \sqsubseteq t : \mathrm{put}_T(\langle k, d\rangle)\ in\ s \Rightarrow (Snd©s)[k] = snd(d)\\
S_3(t) &\equiv \forall s \sqsubseteq t : \mathrm{put}_R(ack(k))\ in\ s \Rightarrow \exists d : \mathrm{get}_R(\langle k, d\rangle)\ in\ s
\end{aligned}
$$

Table 6.8. Safety properties

Example 6.2.1. The first refinement step $Step_1(t) \Rightarrow Step_0(t)$ is split up into the two tasks:

$$Step_0(t) \rightarrow UL(t)$$

and

$$Step_0(t) \rightarrow US(t)$$

Whenever a situation occurs where the theorem is a (variable-disjunct) conjunction of formulas, it is advisable to split up that task into several subtasks. Usually, this reduces the search space considerably. Also splitting "$\Leftrightarrow$" into "$\Leftarrow$" and "$\Rightarrow$" helps to reduce the size of the search space.

Summarizing the characteristics of these proof tasks (Table 6.10), we see that the proof tasks have a medium size and syntactic richness, but exhibit long and complex proofs (mainly because of many equational steps). For this case study, answer times had not been an issue, but they should be limited

$$
\begin{aligned}
Step_0(t) &\equiv LS^{1,2}(t) \wedge LL^{1,2}(t) \wedge UL(t) \wedge US(t) \\
Step_1(t) &\equiv LS^{1,2}(t) \wedge LL^{1,2}(t) \wedge S_1(t) \wedge S_2(t) \wedge L_1(t) \wedge L_2(t) \wedge L_3(t) \\
Step_2(t) &\equiv LS^{1,2}(t) \wedge LL^{1,2}(t) \wedge S_1(t) \wedge S_2(t) \wedge S_3(t) \wedge L_1(t) \wedge \\
&\quad L_3(t) \wedge L_4(t) \\
Step_3(t) &\equiv LS^{1,2}(t) \wedge LL^{1,2}(t) \wedge S_1(t) \wedge S_2(t) \wedge S_3(t) \wedge L_3(t) \wedge L_5(t) \wedge \\
&\quad L_6(t)
\end{aligned}
$$

Table 6.9. Definition of the refinement steps

to around one minute. The proof tasks are already in first-order logic, hence the distance between source logic and the prover's logic (first-order logic with equality) is small. Output of human-readable proofs (see Section 6.2.3 below) has been helpful in this case study. No semantic information about traces was given or used in this case study.

category	value		
deep/shallow	Shallow	**Medium**	**Deep**
size & richness	Small	**Medium**	Large
answer-time	Short	**Medium**	Long
distance	**Short**	Medium	Long
extensions	equations		
validity	100 % theorems		
answer	human-readable proof helpful		
semantic info	Y	some	N

Table 6.10. Important characteristics of the proof tasks for verification of the Stenning protocol

6.2.3 Using the Automated Prover

The proof tasks defined above were the basis for this case study. The aim was to show that SETHEO is able to process these tasks automatically. Since no "application system" existed, the formulas had to be entered by hand. Nevertheless, we assumed that the general input for the application are proof tasks of the form as shown above. Since these proof tasks are already in first-order logic, no translation had to take place[14].

Preprocessing and Transformation of Notation. The goal of this first step is to transform all formulae (in the original notation) into a syntactical form which is readable by SETHEO.

[14] However, a temporal logic describing the traces could have been used instead. For the translation of such a logic into first-order logic see Section 7.2.2.

The major part of this step involves the decision which operator (or function symbol) is represented as a predicate symbol, and which symbols are written as (syntactic) function symbols. Because of better readability, we have selected equality (=), and all relational binary operators (i.e., $\leq, in, >, \sqsubseteq$) to be represented as predicate symbols of arity 2 ("equational representation")[15]. All other symbols occurring in the formulae are transformed into prefix function symbols or syntactic constants (Table 6.11).

predicate symbols		function symbols & constants	
original form	SETHEO input	original form	SETHEO input
$A \leq B$	`less(A,B)`	A @ B	`filt(A,B)`
$A\ in\ B$	`in(A,B)`	$get_R(A), get_T(A)$	`getr(A), gett(A)`
$A > B$	`gt(A,B)`	$put_R(A), put_T(A)$	`putr(A), putt(A)`
$A = B$	`equal(A,B)`	$\langle d_0, \ldots, d_n \rangle$	`data_seq(T,N)`
$A \sqsubseteq B$	`ispre(A,B)`	$A \cdot B$	`cons(A,B)`
		$data(A)$	`data(A)`
		$\langle k, d \rangle$	`pair(K,D)`
		$snd(d)$	`snd_data(D)`
		Snd	`snd`
		Rec	`rec`
		$T[n]$	`nth(T,N)`
		∞	`inf`

Table 6.11. Predicate and function symbols in [Dendorfer and Weber, 1992a] and SETHEO-notation

Example 6.2.2. Property L_2: $\text{put}_T(\langle k, d \rangle)\ in\ t \Rightarrow \text{get}_R(\langle k, d \rangle)\ in\ t$ is written in SETHEO-syntax as follows:

```
forall T forall K forall D
(in(putt(pair(K,D)),T) -> in(getr(pair(K,D)),T))
```

Selection of Axioms. The goal of this study has been to *find* proofs for the given proof tasks. Therefore, we did not define a full set of axioms defining all operations on traces. Rather, we started with a small set of "common" axioms (for equality) and added more axioms and primitive lemmata "by need"; lemmata of which it was thought that they might be helpful for the current proof task. Since these lemmata are quite obvious and directly follow from the definition of the operators, we will call them *high-level axioms.* However, the axioms needed for our proof tasks are often rather weak (e.g. $\forall t, a : \#(a©t) = \infty \Rightarrow a\ in\ t$, i.e., if an action occurs infinitely often in a trace, it occurs at least once in that trace). Those axioms are too weak to be useful in a general context.

[15] An alternative representation would be a relational one. Here, an expression like $a = b + c$ would be written as `plus(a,b,c)`. Then, however, a different handling of equality would be necessary (see also Section 3.3.2).

Example 6.2.3. Table 6.12 shows a number of important axioms used for the proof tasks of this case study. Axioms $Ax_{1.1}, Ax_{1.2}, Ax_{1.3}$ describe "$\sqsubseteq$" as a reflexive, antisymmetric and transitive binary relation. Axiom $Ax_{2.1}$ defines a way of extracting data packages out of a trace x. Axiom $Ax_{3.1}$ concerns the strong definition of actions (here: send actions $snd(d)$): if the k-th element of a sub-trace containing only snd-actions is a $snd(d)$ then the length of that trace is greater than k. Note that indices start with 0. $Ax_{3.2}$ to $Ax_{3.4}$ are different representations of this axiom, concerning the data blocks d_i themselves. $Ax_{4.1}$ define the operations *snd* and *data* as inverse to each other.

$Ax_{5.1}$ and $Ax_{5.2}$ describe properties of actions occurring in a trace. $Ax_{6.1}$ finally states the monotonicity of *snd*. For a complete list of axioms (also containing axioms of equality and several other auxiliary axioms) see [Schumann, 1995].

$Ax_{1.1} \equiv \forall t : t \sqsubseteq t$
$Ax_{1.2} \equiv \forall x, y, z : x \sqsubseteq y \wedge y \sqsubseteq z \Rightarrow x \sqsubseteq z$
$Ax_{1.3} \equiv \forall x, y : x \sqsubseteq y \wedge y \sqsubseteq x \Rightarrow x = y$
$Ax_{2.1} \equiv \forall x, k : data(x[k]) = (data(x))[k]$
$Ax_{3.1} \equiv \forall s, k, d : (Snd@s)[k] = snd(d) \Rightarrow \#(Snd@s) > k$
$Ax_{3.2} \equiv \forall t, n : \langle d_0, \ldots, d_n \rangle \sqsubseteq data(Snd@t) \Rightarrow$ $\#(Snd@t) > n$
$Ax_{3.3} \equiv \forall s, n, d, k : (k \leq n \Rightarrow (Snd@s)[k] = snd(d))$ $\Rightarrow \langle d_0, \ldots, d_n \rangle \sqsubseteq data(Snd@s)$
$Ax_{3.4} \equiv \forall t, n, k, x, s :$ $x \sqsubseteq s \wedge k \leq n \Rightarrow \langle d_0, \ldots, d_n \rangle[k] = s[k]$
$Ax_{4.1} \equiv \forall x, y : x = snd(y) \Rightarrow data(x) = y$
$Ax_{5.1} \equiv \forall s, x, t : s \cdot x \sqsubseteq t \Rightarrow x \; in \; t$
$Ax_{5.2} \equiv \forall t, a : \#(a@t) = \infty \Rightarrow a \; in \; t$
$Ax_{6.1} \equiv \forall x, y : snd(x) = snd(y) \Rightarrow x = y$

Table 6.12. Important problem specific axioms for this case study

Example 6.2.4. Proof task 1 from Table 6.13 ($Step_1(t) \Rightarrow US(t)$) thus consists of the following formula. For the original experiments with SETHEO, the standard axioms for equality have also been added.

$$S_1 \wedge S_2 \wedge LS^1 \wedge LS^2 \wedge L_1 \wedge L_2 \wedge L_3 \wedge Ax_{1.X} \wedge Ax_{3.4} \wedge Ax_{6.1}$$
$$\Rightarrow \forall s, t : s \sqsubseteq t \Rightarrow (data(Rec@s) = \langle d_0, \ldots, d_n \rangle$$
$$\Rightarrow \langle d_0, \ldots, d_n \rangle \sqsubseteq data(Snd@s))$$

Postprocessing Steps. All proofs found by SETHEO have been postprocessed by ILF-SETHEO (Section 7.7) in order to obtain human readable proofs in LaTeX type-setting. Although this tool has not been available during the first experiments in this case study, it would have been extremely helpful for setting up the set of required axioms and to debug both formulas and axioms.

6.2.4 Experiments and Results

Table 6.13 gives an overview of the results of the 10 proof tasks. Proof tasks (6),(7) and (10) are variations of task (5) and (9), respectively. For each proof task, we show its definition, and the number of clauses after transformation into clausal form (*cl*); run times for SETHEO (V3.2) are in seconds and have been obtained on a SUN Sparc10. SETHEO was started with its default parameters, and iterative deepening over the depth of the tableau (A-literal depth). Only in those cases, where SETHEO did not find the proof within a few seconds, two additional techniques (column "*R*") have been used: enabling the additional "fold-up" inference rule (Section 3.3.2) [Letz *et al.*, 1994], and using the preprocessor DELTA (Section 7.6.1).

#	definition	*cl*	T_{run} [s]	*R*	length of proof		
					total	Ext	Red
1	$Step_1(t) \Rightarrow US(t)$	23	0.2	–	10	9	1
2	$Step_1(t) \Rightarrow UL(t)$ (strong)	22	0.2	–	6	5	1
3	$Step_2(t) \Rightarrow Step_1(t)$	24	0.6	*f*	18	18	-
4	(3) without Lemma (8)	23	7.7	Δ	20	20	-
5	$Step_3(t) \Rightarrow Step_2(t)$	25	17.0	Δ	20	20	-
6	$Step_3(t) \Rightarrow Step_2(t)$	40	8.0	–	20	20	-
7	$Step_3(t) \Rightarrow Step_2(t)$	37	0.4	–	6	6	-
8	Lemma for task 3	24	0.2	–	11	11	-
9	Lemma for task 5,6,7	26	0.1	–	3	3	-
10	Lemma for task 5,6,7	24	0.1	–	10	10	-

Table 6.13. Experimental results for all proof tasks

6.2.5 Assessment and Discussion

This case study was intended to study an ATP application from scratch, i.e., without any application system to generate the proof tasks. Hence, the run-times shown in Table 6.13 do not reflect the time needed to carry out the case study. The entire case study was done within a few working days[16]. Major time-consuming tasks were (in decreasing order): setting up an infrastructure (to keep all formulas and axioms in a consistent state), to debug formulas, to find the appropriate axioms, and to check the machine-proofs for correctness.

The goal of this work was to study how the proof tasks must be formulated and which preprocessing steps have to be performed such that they can be solved automatically with the theorem prover SETHEO[17]. In order to be as

[16] This time is only an estimate, because the author did not log the sessions. Furthermore, it does not include the time needed to document the results.

[17] Most proof tasks of this case study have also been successfully tackled with Protein [Baumgartner and Schumann, 1995].

"realistic" as possible we took all proof tasks and their first-order formalization directly from [Dendorfer and Weber, 1992a][18]. One of the most difficult steps in this case study was to find the right set of axioms and lemmata which were needed for the actual proof task. In this case, we selected the way of directly giving the axioms for each proof task rather than developing a complete set of axioms with subsequent preselection. The axioms were also set up in such a way that explicit induction could be avoided[19]. With these simplifications we could study the issues of handling equality (by adding axioms), control of the prover, and the generation of readable proofs by ILF-SETHEO.

During this case study it turned out that two extensions of the theorem prover could have facilitated the work extremely: a sorted input first-order logic can can detect many errors in the input (e.g., typing `file` instead of `filt` for @) early[20]. Any method for detecting non-theorems reduces much of the time needed to debug conjectures which are not formulated correctly or searching for missing axioms. Both items are an important prerequisite for using an automated theorem prover for verification tasks without an elaborate application environment.

6.3 Logic-based Component Retrieval

6.3.1 Introduction

Reuse of approved software components is certainly one of the key factors for successful software engineering. Although the reuse process also covers many non-technical aspects [Zand and Samadzadeh, 1995], retrieving appropriate software components from a reuse library is a central task. This case study concentrates on a deduction-based approach[21]. Here pre- and postconditions are used as the components' indexes and search keys. A component matches a search key if the involved pre- and postconditions satisfy a well-defined logical relation, e.g., if the component has a weaker precondition and a stronger postcondition than the search key. From this matching relation a proof task is constructed and an ATP is used to establish (or disprove) the match. The pre- and postconditions comprise a *contract* for the component. As the contract language, the specification language VDM/SL is used. This approach has several advantages over retrieval methods which take the entire implementation of a component for matching into account. For example, we get smaller and easier proof tasks and there is no need for a full specification of

[18] For an alternative representation of the trace logic see Example 7.2.7 in Section 7.2.2.

[19] In [Dendorfer and Weber, 1992a] all manual proofs were also carried out without induction.

[20] However, in this application, sorts are not necessary for correctness and they are not able to reduce the search space.

[21] For an overview on logic-based component retrieval see [Mili *et al.*, 1998].

the modules. However, in a contract based approach, the correctness of the relationship between contract and implementation must be ensured by the developer or administrator of the reuse library. Our application system is the tool NORA/HAMMR[22] which generates the proof tasks for the automated prover. This system and the ATP application has been tested extensively with a realistic library containing 119 reusable components. This case study has been carried out by the author in tight cooperation with the TU Braunschweig, Germany (B. Fischer, G. Snelting [Schumann and Fischer, 1997; Fischer *et al.*, 1998; Fischer and Schumann, 1997; Fischer, 2001]).

6.3.2 The Application

Design Rationales. The goal of the project NORA/HAMMR [Fischer *et al.*, 1998; Schumann and Fischer, 1997] was to build a practical deduction-based retrieval tool for the working software engineer. From the user's point of view, practicability of reuse is determined by response times, *recall* ("do we get all matching components?") and *precision* ("do we get the right components?"). In logic-based component retrieval based on contracts, the entire software process also plays a major role (cf. [Fischer and Snelting, 1997]). In a formal process (level 2 or 3 according to our classification in Chapter 2) it is most important to find *provably* matching components. In contrast to this, in a non-formal environment, the user might want to get as many *promising* candidates as soon as possible. Here, more relaxed matching rules allow to find more candidates (higher recall), but these must be manually inspected and modified to fit exactly. Hence, the goals of retrieval must be user-definable in an easy way.

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

Number and structure of proof tasks pose severe problems. Since each single query already induces a number of complex proof tasks which is of the order of the size of the component library, construction and processing must be done fully automatically. Additionally, since reasonable libraries contain only few matches for any query, most of the tasks describe non-theorems. These non-theorems must be removed as soon as possible to prevent the ATP from drowning. These requirements must be reflected in system design and architecture, and in the adaptation of the actual theorem prover.

System Architecture. NORA/HAMMR's system architecture (Figure 6.5) reflects these diverse requirements. The system NORA/HAMMR is built

[22] NORA is no real acronym, HAMMR is a highly adaptive multi-method retrieval tool which has been developed at the Technische Universität Braunschweig, Germany.

around a pipeline of independent filters through which the candidates are fed.

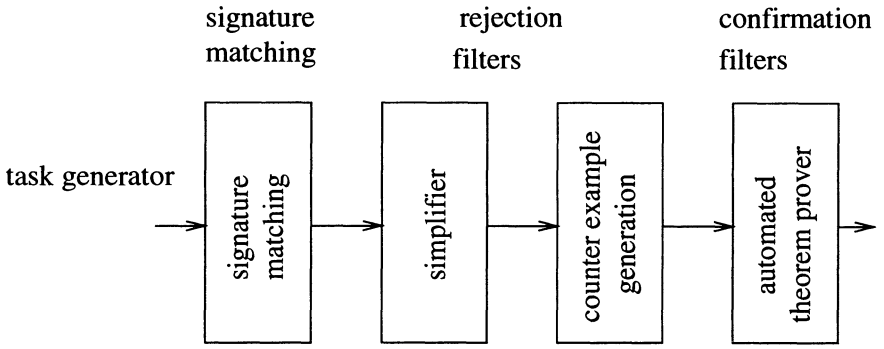

Fig. 6.5. NORA/HAMMR's system architecture

Typically, a filter pipeline starts with a *signature matching* filter which checks for compatible calling conventions. Then, *rejection* filters try to efficiently discard as many non-matches as possible. Finally, *confirmation* filters are invoked to guarantee a high precision of the retrieval results. The pipeline itself can be freely customized although any configuration should maintain a fair balance between initial fast responses and a final high precision. Each component which passes a filter can immediately be inspected by the user. This early feedback is essential for effective control of the retrieval process. NORA/HAMMR allows the integration of arbitrary non-deductive methods, e.g., full text retrieval (see, e.g., [Maarek and Smadja, 1989; Maarek *et al.*, 1991; Frakes and Nejmeh, 1987]). This is especially important in a non-formal process where the established methods should be augmented rather than be replaced in order to gain acceptance.

The prototypical graphical user interface (cf. Figure 6.6) reflects our system architecture. The filter pipeline may be preselected via a pull-down menu and may easily be customized through the central icon pad. Each icon also hides a specialized control window which allows some fine-tuning of the filter. Additional windows display intermediate results and grant easy access to the components. They also allow to save retrieved components into new library files which may be used for subsequent retrieval runs. A simple, VCR-like control panel provides single-button control of the entire system. The objective of the GUI is to hide evidence of automated deduction techniques as far as possible but to grant control over these techniques as far as necessary. Hence, using NORA/HAMMR as a *retrieval tool* requires only knowledge of the contract language (VDM/SL) and the target languages.

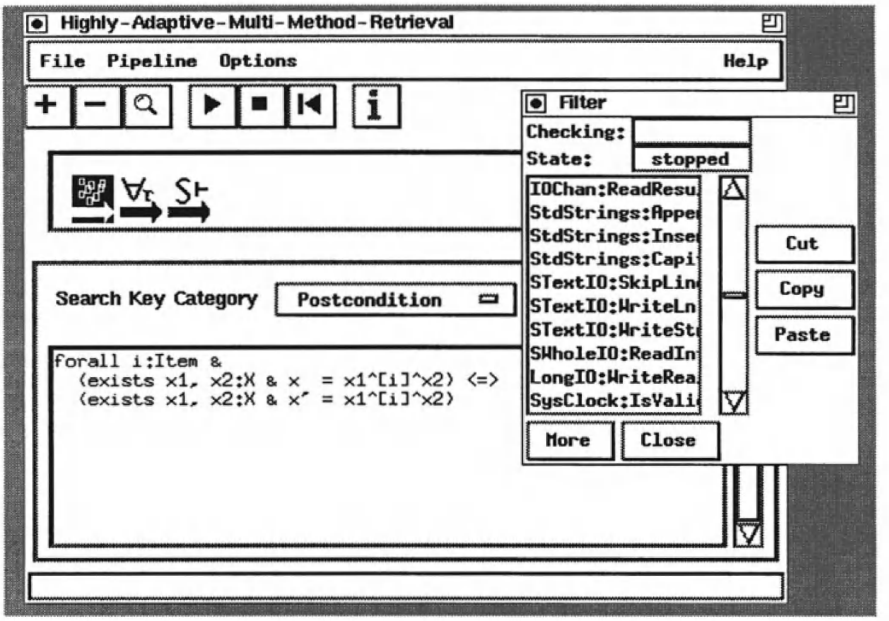

Fig. 6.6. NORA/HAMMR's graphical user interface

However, it should be noted that usually, a reuse-administrator is also required for smooth operation. This person is responsible for maintaining the libraries, and controlling new entries into the libraries, as well as tuning of the system. Therefore, the administrator must have some knowledge about the system architecture and the underlying reasoning techniques.

Filter Pipeline. The filter pipeline of NORA/HAMMR tries to discard as many non-matches (non-theorems) as possible while trying to solve trivial tasks. After passing the filters, only very few proof tasks should remain for processing by the ATP, as shown in Figure 6.7. Otherwise, the automated prover will be drowned by proof tasks, making practical retrieval impossible. Therefore, the pipeline is not only the interface for the ATP but can also be seen as a sequence of additional preprocessing stages for the ATP. Therefore, we will describe the filters in some detail.

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

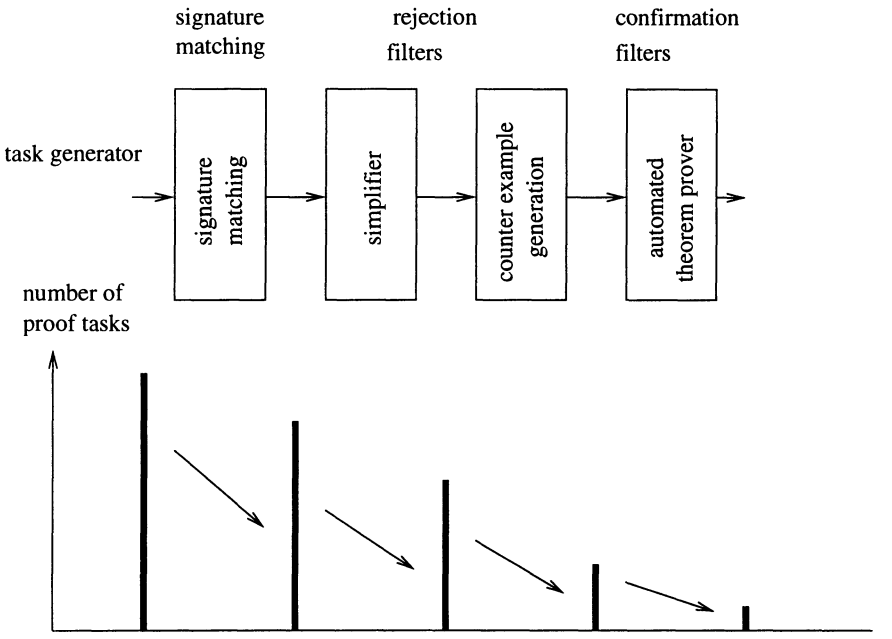

Fig. 6.7. NORA/HAMMR's filter pipeline and the relative number of proof tasks processed by the filters.

Signature matching has two important side effects. It identifies the respective formal parameters of the two components and it maps the respective domains (i.e., types) onto each other. Both effects are required to "join" the different scopes of two contracts together. This makes signature matching also a necessary prerequisite for the subsequent filters. In NORA/HAMMR, signature matching is implemented as a variant of E-unification [Fischer *et al.*, 1998].

Rejection Filters. Detecting and rejecting non-theorems is an extremely important step in deduction-based retrieval because most of the tasks result from non-matching candidates. Unfortunately, most ATPs for full first-order logic are not suited for this. They exhaustively search for a proof (or refutation) of a conjecture (or its negation) but they usually fail to conclude that it is *not* valid (or contradictory). Therefore, rejective filters use efficient decision procedures for logics which are weaker than general first-order logic and translate or abstract the given proof tasks.

Such filters are necessarily incomplete in a sense that non-theorems may still pass through. However, from a practical point of view, this is not too severe as this only reduces the effectiveness of the filters. Unfortunately, some rejection filters may also be unsound and valid tasks may be rejected improperly, e.g., because the applied abstraction is not sufficiently precise. Such

candidates are lost for the retrieval process and reduce the recall of the entire pipeline. Thus, in practice a balance between this loss and the achieved reduction must be found.

In the following, we will focus on logic simplification filters for detecting trivial matches and discarding non-matches (studied by B. Fischer in [Fischer, 2001; Fischer *et al.*, 1998; Fischer and Schumann, 1997]) and applying model generators to find counterexamples.

Logic Simplification. In NORA/HAMMR, extensive experiments with different levels of simplification procedures have been made in [Fischer *et al.*, 1998; Fischer, 2001]. On the lowest level (called FOL-level), propositional constants TRUE and FALSE are eliminated and quantifier scopes are minimized. The next level handles equalities (EQU) with at least one side being a single variable (or constant). Due to the structure of the proof tasks it is necessary to use inequalities for conditional rewriting, too.

However, in order to achieve a significant filtering effect more semantic information must be utilized. Semantic information is extracted automatically from the reuse library and used for rewriting the formula. On a lower level, all appropriate axioms and sort information is used for that purpose. On a second level, a single unfolding step of recursively defined data types (e.g., list) is performed. Then, the resulting formulas for the base case and a simplified step case (similar to "poor man's induction" in Section 7.2.1) are simplified again.

These methods of logic simplification will be described in more detail in Section 7.3. In this context, however, simplification is not used primarily to prune the search space. Rather, the simplifier tries to rewrite formulas ("trivial cases") to TRUE or FALSE, thus eliminating them from the filter pipeline.

Counterexample Generation. Obviously, a component can be rejected if we find a "counterexample" for its associated proof task because it then cannot be valid. In this case study, we used the model generator MACE [McCune, 1994b] to find such counterexamples (which are nothing more than variable instantiations under which the formula evaluates to FALSE). However, such a model generator only terminates if all involved domains are finite (or at least have finite model property). Generally, proof tasks over recursively defined data structures do not have this property. Hence, for our experiments we use domain approximations: from the infinite domain, we select a number of values which seem to be "crucial" for the component's behavior. E.g., for data type list, we pick the empty list [] and some small lists with one or two elements. Then, we search for models or a counterexample. This approach mimics the manual checking for matches: if one looks for matching components, one first makes checks with the empty list and one or two small lists. If this does not succeed, the component cannot be selected. Otherwise, additional checks have to be applied. This approach, however, is neither sound

nor complete. Results on experiments with that filter are described and assessed in Section 6.3.5. Using model generators to detect non-theorems will also be covered in more detail in Section 7.5.

6.3.3 The Proof Tasks

The structure of the proof tasks which are generated by NORA/HAMMR only depends on the contract language (VDM/SL) and the *match relation* which is used[23]. Most often, NORA/HAMMR is configured to ensure *plug-in compatibility* of the retrieved components: component c matches if it has a weaker precondition and a stronger postcondition than the search key q. This is usually formalized as $(pre_q \Rightarrow pre_c) \wedge (post_c \Rightarrow post_q)$. However, a variety of different match relations (e.g., conditional or partial compatibility) can be defined. Whereas some relations (e.g., plug-in compatibility) allow retrieved modules to be used "as is", others require inspection and manual modifications to the module. For practical purposes it can be helpful to switch between different modes. For an extensive survey on match relations see [Fischer, 2001].

Example 6.3.1. Let us consider the following VDM/SL specifications (from [Schumann and Fischer, 1997]): *rotate* places the first element of a non-empty list at its end. *Shuffle* is a function with a parameterized argument type which shuffles all elements of a non-empty list.

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

A proof task (still in VDM/SL's logic) is constructed by taking the pre- and postconditions of the query q (here: *shuffle*) and those of one module of the library, the candidate c (in our case *rotate*) and combining them according to the match relation:

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

Several steps are necessary to construct a sorted FOL formula out of that proof task. First, the formal parameters must be identified, in this case $l = x$ and $l' = x'$.[24] Then, VDM's underlying three-valued logic LPF (logic of

[23] This is an eminent advantage of contract-based approaches over other retrieval approaches where the actual implementation language must also be considered. In NORA/HAMMR only the definition of the formal parameters in the implementation language is required for controlling the signature filter.

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

partial functions, [Barringer *et al.*, 1984]) must be translated into FOL eliminating "undefined" conditions. In NORA/HAMMR the approach of [Jones and Middelburg, 1994] is used. For details see [Fischer *et al.*, 1998].

Example 6.3.2. For our example the translation results in the following sorted first-order formula (which obviously needs simplification):

$$\begin{aligned}
&\forall l, l', x, x' : \mathsf{list} \cdot (l = x \wedge l' = x' \wedge \mathsf{true} \Rightarrow \mathsf{true}) \\
&\wedge\ (l = x \wedge l' = x' \wedge (l = [] \Rightarrow l' = [])) \\
&\wedge\ (l \neq [] \Rightarrow (l \neq [] \Rightarrow l' = (\mathsf{tl}\ l)^\wedge[\mathsf{hd}\ l])) \\
&\quad \Rightarrow (\forall i : \mathsf{item} \cdot (\exists x_1, x_2 : \mathsf{list} \cdot x = x_1{}^\wedge[i]{}^\wedge x_2 \\
&\qquad \Leftrightarrow \exists x_1, x_2 : \mathsf{list} \cdot x' = x_1{}^\wedge[i]{}^\wedge x_2)))
\end{aligned}$$

The main characteristics of the proof tasks in this case study (after translation into first-order logic) are given in Table 6.14. The proof tasks are usually of a medium size and syntactical richness. They have short to medium-size proofs. However, some of them can only be found by induction (see below). Of particular importance for this application is the requirement for short answer times. Proof obligations in VDM/SL are translated by NORA/HAMMR into first-order logic with sorts and equality. However, most of the proof tasks are non-theorems which makes the filter pipeline necessary. No explicit proofs are required from the theorem prover and semantic information (valuable for simplification and preselection of axioms) is provided.

category	value		
deep/shallow	**Shallow**	**Medium**	Deep
size & richness	Small	**Medium**	Large
answer-time	**Short**	Medium	Long
distance	**Short**	Medium	Long
extensions	equality,sorts		
validity	10–15 % theorems		
answer	boolean result		
semantic info	**Y**	some	N

Table 6.14. Proof tasks of NORA/HAMMR: important characteristics

6.3.4 Using the Automated Prover

These proof tasks (already with a first simplification performed) form the input for the automated theorem prover which comprises the final stage of our filter chain. The soundness of the prover ensures that the precision of the retrieval process is always 100%, i.e., no wrong modules are selected[25]. On

[25] Soundness is always guaranteed by the prover, although individual filters can be unsound. Their unsoundness only leads to the situation that more proof tasks must be processed by the ATP.

the other hand, as many proof tasks as possible should be processed within short run-times in order to ensure acceptable recall.

This case study has been developed with the automated theorem prover SETHEO. Extensive experiments with other provers, Protein [Baumgartner and Furbach, 1994b], Gandalf, OTTER, and SPASS [Weidenbach *et al.*, 1996; Ganzinger *et al.*, 1997] have been made. They are described in detail in [Fischer *et al.*, 1998] and [Fischer, 2001]. However, the architectural design of the interface and many preprocessing steps are common for all provers, e.g., handling of inductive problems, selection of axioms, and logic simplifications. For each proof task, a number of preprocessing steps must be performed as shown in Figure 5.2 on page 79:

- Syntactic translation and conversion into clausal normal form. For the transformation into sorted clausal normal form, a standard algorithm (e.g., [Loveland, 1978]) was used. However, it turned out to be very important to perform optimizations (for the basic ones see again [Loveland, 1978]) in order to avoid Skolem functions with unnecessary high arity which tend to increase the search space considerably[26].
  Furthermore, care must be taken that the correct sort of the Skolem constants and functions are retained. Our prototype, however, uses a conversion module which is not capable of handling sorted first-order logic. Nevertheless, the correct handling of sorts during the transformation can be accomplished easily by renaming all sorted variables in such a way that the sort becomes part of the name of the variable (e.g., $X : t$ becomes `X__t`). If during the Skolemization, the Skolem constants keep the entire variable name (i.e., if we get `a_19_X__t` instead of just a_{19}), the required sort information can be extracted easily.
- Processing of inductive problems. Many proof tasks are based on recursive data structures (in our case list), thus giving rise to inductive proof tasks. SETHEO itself cannot handle induction and our severe time-constraints don't allow us to use an inductive theorem prover (such as e.g., Oyster/Clam [Bundy *et al.*, 1990]). With SETHEO, we made experiments with different kinds of approximations: adding appropriate lemmata, case splitting, and poor-man's induction (all methods described in detail in Section 7.2.1).

Example 6.3.3. Let us assume a query and candidate with the signature l : list. With case splitting over lists and the corresponding proof task of the form $\forall l : \mathsf{list} \cdot \mathcal{F}(l)$ we obtain the following cases:[27]

[26] This is also a major reason, why we use an equational representation for our proof tasks. This means that an expression of the form $X = a \hat{} b$ is written as `equal(X,cons(a,b))`. Another approach, a "relational representation" with a 3-ary predicate (`cons(a,b,X)`) avoids direct equality, but introduces many high-arity Skolem functions in formulas with complex terms.

[27] Although it would be sufficient to have cases 1 and 3 only, we also generate case 2, since many specifications are valid for non-empty lists only. For those

1. $l = [] \Rightarrow \mathcal{F}(l)$
2. $\forall i : \mathsf{item} \cdot l = [i] \Rightarrow \mathcal{F}(l)$
3. $\forall i : \mathsf{item}, l_0 : \mathsf{list} \cdot \mathcal{F}(l_0) \wedge l = [i]^\frown l_0 \Rightarrow \mathcal{F}(l)$

After rewriting the formula accordingly, we get three independent first-order proof tasks which then must be processed by SETHEO. Only if all three proof tasks can be proven the entire proof task is considered to be solved.

- Preselection of axioms. Since most automated theorem provers are very sensitive with respect to the number of clauses added to the formula, several methods for preselecting reasonable sets have been evaluated. With a straightforward axiom preselection similar to that in [Reif and Schellhorn, 1998] (Section 5.7) the number of solved proof tasks could be increased considerably (from 55.9% to 69.2% in our experiments).
- Handling sorted logic. All proof tasks are sorted. The sorts are imposed from the VDM/SL specifications of the modules and are structured in a hierarchical way. All sorts are static and there is only limited overloading of function symbols. Therefore, the approach to *compile* the sort information into the terms of the formula can be used (see Section 7.4.3). Then, determining the sort of a term and checking, if the sorts of two terms are compatible, is handled by the usual unification. Thus there is no need to modify SETHEO and the overall loss of efficiency is minimal. For our case study we used the tool ProSpec (developed within Protein [Baumgartner and Furbach, 1994b]).
- Handling equality. All proof tasks heavily rely upon equations. This is due to the VDM/SL specification style and the construction of the proof tasks. While some equations just equate the formal parameters of the query and the library module, others carry information about the modules' behavior. Therefore, efficient means for handling equalities must be provided. For SETHEO, we currently provide two variants: the naïve approach by adding the corresponding axioms of equality (reflexivity, symmetry, transitivity, and substitution axioms), and the compilation approach used within E-SETHEO [Moser *et al.*, 1997]. Here, symmetry, transitivity and substitution rules are compiled into the terms of the formula such that these axioms need not be added. This transformation, an optimized variant of Brand's STE modification [Brand, 1975], usually increases the size of the formula, but in many cases the length of the proof and the size of the search space becomes substantially smaller.
  Our experiments showed that none of the methods is a clear winner. Proof tasks with a rich and complex structure of terms result in large sets of long clauses, if processed by E-SETHEO's preprocessing module. Such huge formulas often just take too long (w.r.t. to our time limits) to be handled

specifications, case 1 would be a trivial proof task which does not contribute to filtering.

by SETHEO. If only few equational transformations are needed during the proof, the naïve approach certainly needs fewer resources. In many cases, however, complex equational transformations are necessary which clearly favors the use of E-SETHEO. Therefore, both variants are usually tried in a parallel competitive way in our prototype.

- Control of the prover. Once started, the theorem prover has only a few seconds of run time to search for a proof. This requires that the parameters which control and influence the search (e.g., iterative deepening, subgoal reordering) are set in an optimal way for the given proof task. However, such a global setting does not exist for our application domain. In order to obtain optimal efficiency combined with short answer times, *parallel competition* over parameters is used (see Section 7.6.2). The basic ideas have been developed for SiCoTHEO [Schumann, 1996a] and could be adapted easily. On all available processors (e.g., a network of workstations), a copy of SETHEO is started to process the entire given proof task. On each processor, a different setting of parameters is used. The process which first finds a proof "wins" and aborts the other processes.

6.3.5 Experiments and Results

The Experimental Data Base. The experiments in this case study have been made with a test component library, containing 119 specifications of functions about lists[28]. Whereas around 2/3 of these specifications describe actual functions (e.g., get first element, delete minimal element), the others represent queries (some of which are also under-specified, e.g., functions which only require that a non-empty list is returned). In order to obtain a realistic number of proof task each specification was cross-matched against the entire library. This resulted in a total of 14161 proof tasks where 13.1% (or 1838) were valid matches.

Filters I: Simplification. Table 6.15 (adapted from [Fischer *et al.*, 1998]) summarizes the results of the simplification filter for each of the simplification levels. All simplification methods are sound, i.e., no valid matches are lost by this filter (recall $r = 100\%$). However, purely syntactic simplification cannot eliminate any non-matches, as indicated by a fallout (i.e., percentage of non-matches which pass the filter) of 100%.

Adding semantic information (sorts and definitions of data types) improves the situation slightly, but only with unfolding are we able to reject more than 40% of the tasks because these are rewritten to FALSE. The set of surviving candidates still contains more than 53% of the original non-matches (fallout) but its precision already increased by a factor of 1.7. As a positive side-effect, syntactic simplification (levels: FOL and EQU) can rewrite a significant number of valid proof tasks (21.0% and 26.7%, respectively) to TRUE.

[28] All specifications describe functions list $\rightarrow$ list such that no signature matching is necessary.

Level	FOL	EQU	Sorts	Unfolding
recall r	100.0%	100.0%	100.0%	100.0%
fallout	100.0%	100.0%	94.4%	53.2%
rewritten to TRUE	21.0%	26.7%	26.7%	26.7%

Table 6.15. Results of simplification

These tasks directly can be added to the set of retrieved components, thus reducing answer times considerably.

Filters II: Generation of Counter Examples. For the rejection filter using model generation (MACE), experiments with three different models with at most three elements for each data type (in our case list, item) have been made. Due to the large number of variables in the proof tasks, only such small models as shown in Table 6.16 can be used within reasonable runtimes. Our experiments with MACE, however, revealed that these restrictions are not too serious.

Model A

List: $[], l \equiv [i]$

Item: i

function	$[]$	l
cons$(i,_)$	l	$l^\dagger$
tail$(_)$	$[]$	$[]$
head$(_)$	i	i
app$([],_)$	$[]$	l
app$(l,_)$	l	$l^\dagger$

Model B

List: $[], l_1 \equiv [i], l_2 \equiv [i]\hat{}l_1$

Item: i

function	$[]$	l_1	l_2
cons$(i,_)$	l_1	l_2	$l_2^\dagger$
tl$(_)$	$[]$	$[]$	l_1
hd$(_)$	i	i	i
app$([],_)$	$[]$	l_1	l_2
app$(l_1,_)$	l_1	l_2	$l_2^\dagger$
app$(l_2,_)$	l_2	$l_2^\dagger$	$l_2^\dagger$

Model C

List: $[], l_1 \equiv [i_1], l_2 \equiv [i_2]$

Item: i_1, i_2

function	$[]$	l_1	l_2
cons$(i_1,_)$	l_1	$l_1^\dagger$	$l_1^\dagger$
cons$(i_2,_)$	l_2	$l_2^\dagger$	$l_2^\dagger$
tl$(_)$	$[]$	$[]$	l_2
hd$(_)$	$i_1^\dagger$	i_1	i_2
app$([],_)$	$[]$	l_1	l_2
app$(l_1,_)$	l_1	$l_1^\dagger$	$l_1^\dagger$
app$(l_2,_)$	l_2	$l_2^\dagger$	$l_2^\dagger$

Table 6.16. Models A,B, and C used for our experiments. Places marked by † represent arbitrary choices for the function definitions. These had to be introduced to ensure the finiteness of the model and since MACE cannot handle partial functions.

As shown in Table 6.17, the model checking filter (with a run time limit of 20 seconds) is able to recover at least 75% of the relevant components, regardless of the particular model. The large standard deviation, however, indicates that the filter's behavior is far from uniform and that it may perform poor for some queries. Unfortunately, the filter is still too coarse. The values for fallout (i.e., the number of non-matching components which can pass the filter) indicate that about 41% – 55% of non-matches are still in the pipeline placing the performance of this filter at the lower end of our expectations.

Model	A	B	C
$\|List\| + \|Item\|$	2+1	3+1	3+2
recall r	74.7%	76.5%	81.3%
σ_r	0.25	0.26	0.25
fallout	42.8%	41.0%	55.5%

Table 6.17. Results of model checking

SETHEO as Confirmation Filter. With SETHEO, several sets of experiments have been carried out. Since SETHEO cannot handle non-theorems, we only took the valid proof tasks from our experimental database. This resulted in a total of 1838 proof tasks. Table 6.18 gives an overview over the results. All experiments have been made on a Sun Ultra II with a run-time limit of 60s. We give the total number of proof tasks, the number of successfully solved tasks, the number of errors, and recall r, i.e., the percentage of solved proof tasks with respect to all valid proof tasks[29]. The first row (Exp 1) gives the results of a very simple experiment, namely the retrieval of identical components. Here, no axioms about the theory of lists are required. Hence, most of the proof tasks could be solved within the given run-time. This figure shows that the proof tasks have been generated correctly and are in the range of complexity which can be handled by current-technology ATPs.

The next three experiments concern the selection of axioms. For each of the experiments, a standard version of SETHEO has been used. Equality is handled by adding the appropriate axioms. First, we start the system without adding any domain-specific axioms (Exp 2). Many of the proof tasks are so simple such they can be solved already at this level[30]. In the next experiment (Exp 3) we have added all axioms, defining the underlying theories, and additional lemmata. The formulas get much larger which caused the number of errors to increase considerably. Although the number of retrieved components (r_{fullax}) could be raised substantially, the large search space spanned by the axioms causes SETHEO to reach a time-out in many cases. However, with a straightforward preselection of axioms (see Section 5.7), many more proof tasks can be solved within the given run-time limit (Exp 4). Experiment (Exp 5) takes the same proof tasks, but uses a version of p-SETHEO (Section 7.6.2). This version has been tuned for solving benchmark problems taken from the TPTP library. The results revealed by p-SETHEO are dissapointing and reveal that this kind of application proof tasks requires quite different behavior of the automated prover.

Two kinds of induction approximation have been used in experiments (Exp 6) and (Exp 7). Although case splitting (Exp 6) cannot solve as many

[29] In this table, the errors count as proof tasks which have not been solved.

[30] The 1838 proof tasks contain 703 tasks which already could be solved by the simplification filter, i.e., they could be rewritten to TRUE. For these, the ATP obviously yields a success.

tasks in total as in (Exp 4) (mainly because more errors occurred), we were able to prove 36 additional tasks which could not be solved otherwise. Hence, induction can be used for parallel competition. When we combine (Exp 2),(Exp 4), (Exp 5), and (Exp 6), we get a recall of 74.5% which is already acceptable for practical retrieval (Exp 8).

Experiment	total	solved	error	recall [%]
Exp 1	119	108	2	$r_{id} = 90.7\%$
Exp 2	1838	851	130	$r_{noax} = 46.3\%$
Exp 3	1838	1027	143	$r_{fullax} = 55.9\%$
Exp 4	1838	1271	69	$r = 69.2\%$
Exp 5	1838	1238	69	$r_{p-setheo} = 67.4\%$
Exp 6	1838	1235	114	$r_{ind} = 67.2\%$
Exp 7	1838	1302	69	$r_{ind-poor} = 70.8\%$
Exp 8	1838	1370	N.A.	$r_{comb} = 74.5\%$

Table 6.18. Experimental results with SETHEO (explained in text)

Results of experiments carried out with other provers (Protein, p-SETHEO with 2 processors and competition on handling equality, and SPASS) are shown in Table 6.19 (from [Fischer *et al.*, 1998])[31]. All provers exhibit a similar behavior. When combining all provers in a competitive way, a recall of up to 80% can be obtained.

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

Table 6.19. Results of Experiments with several different theorem provers (from [Fischer *et al.*, 1998]).

6.3.6 Assessment and Discussion

In contrast to the other case studies in this book, this work directly aims at *industrial* applicability. Therefore, the integration of automated provers into NORA/HAMMR had to obey specific and strong requirements, in particular fully automatic processing (i.e., hiding any evidence of ATPs from the user)

[31] Figures in this table have been obtained with slightly different proof tasks (different kind of preprocessing and simplification).

and short answer times ("results-while-u-wait"). In order to accomplish this, NORA/HAMMR's filter pipeline is an important prerequisite. Nevertheless, the application of an automated prover to the system posed many challenges. Handling of non-theorems, control of the prover, simplification, and handling of inductive problems, sorts and equational problems are central issues. These will be discussed in detail in the next chapter. Due to the good cooperation between the project partners, this application could be evaluated in depth. Much effort had been spent to set up a realistic component library. In contrast to most other case studies where only few examples are available, statistically significant experimental data could be obtained.

The results of experiments with NORA/HAMMR "[...] show that today's ATP technology makes deduction-based software component retrieval an almost realistic option in a formal development process" [Fischer *et al.*, 1998]. Deduction-based software component retrieval is also the key technique which underlies more ambitious logic-based software engineering approaches, e.g., program synthesis [Lowry *et al.*, 1994a] or component adaptation [Penix and Alexander, 1997]. Therefore, results obtained in this case study comprises a foundation for further ATP applications in those areas.

7. Specific Techniques for ATP Applications

When a first-order automated theorem prover is applied in a specific domain a number of pre- and postprocessing steps have to be performed such that the system fulfills the requirements set up in Chapter 5. These processing steps have been researched in depth by the author with the help of various case studies (some of which have been presented in the previous chapter). From this experience and based on numerous discussions with other researchers, a relatively generic scheme and skeleton architecture has been developed. Although not a universal recipe, this scheme can be methodically used as a basis for most applications in the area of software engineering.

We can distinguish four phases of processing for a given proof task (Figure 7.1): *translation* of the proof task with the aim of generating a first-order proof obligation, *preprocessing* of this task (e.g., simplification, handling of sorts, equality), invocation of the *automated prover* itself and, finally, a *postprocessing* phase where results of the prover (e.g., a proof) are converted into a suitable format for the application system.

After an overview, we will focus on specific techniques and methods which have been identified to be of particular importance (cf. also [Schumann, 1998; Kelly, 1997; Kaufmann, 1998]). In this chapter, we will describe in detail techniques for translating non first-order problems (inductive problems and modal logics), simplification, handling of sorts, and postprocessing (generation of answer substitutions and human-readable proofs). In particular, we will focus on the detection of non-theorems and on controlling the automatic prover by exploitation of parallelism and by combination of search paradigms. Other topics, like handling of equality, arithmetic and preselection of axioms — although equally important — are discussed only briefly, because they open up large research areas by themselves.

All processing steps mentioned in Figure 7.1 and ways of solving the given problems are proposed and discussed with the help of the prover SETHEO[1] and examples from our case studies. A section on pragmatic issues summarizes

[1] The author has selected SETHEO for presenting these methods and techniques, because this prover has been used for most of the case studies. Most techniques can directly be transferred to other top-down (goal-oriented) provers (e.g., Protein [Baumgartner and Furbach, 1994b]), and to a large extent also to resolution-type provers.

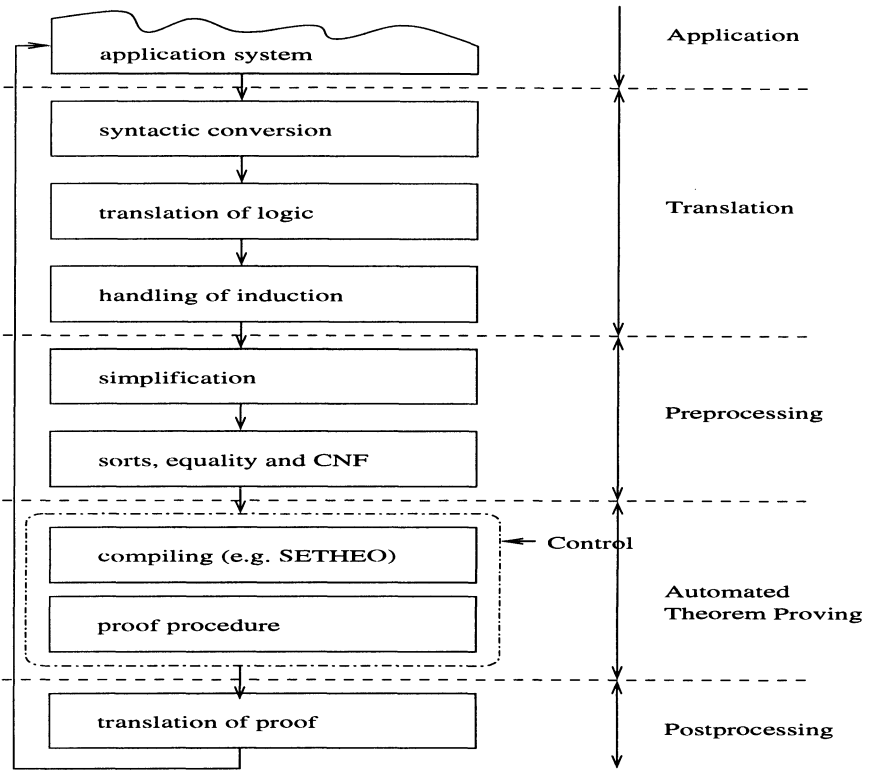

Fig. 7.1. Typical processing steps for an ATP application

the author's experience with pitfalls and practical problems and should be obeyed for the design of automated provers and new applications.

7.1 Overview

As shown in Figure 7.1, a large number of steps has to be performed for the application of an automated prover. Although several of the individual steps can be seemingly trivial syntactic transformations, care must be taken that the entire chain of transformations is robust and implemented correctly. Otherwise, errors which are hard to detect might occur, possibly sacrificing efficiency, soundness, or completeness of the system. In the following, we give an overview of the processing phases and their individual steps. For a more detailed presentation of individual steps, we refer to the corresponding sections in this chapter. Here, we assume that the application system produces one proof task at a time, and expects some answer from the automated theorem prover.

7.1.1 Translation Phase

The aim of this processing phase is to generate one (or more) proof obligations in first-order predicate logic (possibly with sorts, equality, and other theories) out of the given proof task. Typically this phase consists of the following steps:

Transformation of syntax: The syntax of the input formulas usually must be transformed syntactically into a format which is suitable for the subsequent processing modules. This conversion normally has to handle infix operators, specific character sets, and special-purpose symbols[2]. Its output is an ASCII representation of the entire proof task and information on how to regenerate the original representation. This information is typically needed to transform the proof in a postprocessing step.

Translation of source- into target-logic: If the logic of the given proof task is not first-order logic, all formulas in that logic must be translated into first-order logic (possibly with extensions). This translation step will be discussed in detail in Section 7.2.

Handling of induction: Quite often proof obligations — although formalized in first-order logic — cannot be directly solved by a first-order automated theorem prover. This is in particular the case, when recursively defined data types (e.g., lists, natural numbers) are used. The formula must then be proven by induction. In general, finding an inductive proof is extremely hard and thus is restricted to interactive theorem provers and special-purpose inductive provers. However, in practical applications, many proof tasks occur which can be solved by very simple inductive proofs. In Section 7.2.1, we present several techniques to handle simple kinds of induction by approximation.

7.1.2 Preprocessing Phase

With the proof task translated into first-order predicate logic (with sorts, equality and other theories), now, it must be processed in such a way that it can be handled by the automated theorem prover. Its typical input format is pure first-order logic in clausal normal form[3]. Furthermore, all parts of the formula which are redundant or which do not contribute to the proof should be removed. Additional clauses which cannot be used to find a proof usually increase the search space tremendously. Hence, the following steps should be performed during preprocessing:

[2] Although seemingly trivial, the case studies carried out by the author revealed that most errors actually occurred during this phase (e.g., unrecognized operators, long identifier names). For a detailed discussion on these topics see Section 7.8.

[3] Several provers (e.g., OTTER, SPASS, Protein) can handle arbitrary first-order formulas with quantifiers and all connectives ($\vee, \wedge, \rightarrow, \leftrightarrow, \neg$) by first converting the input formula into a set of clauses. Also some provers have built-in techniques for handling sorts (e.g., SPASS, Protein) or equality (almost all bottom-up provers).

Conversion into clausal normal form: Formulas with arbitrary logical operators (e.g., $\wedge, \vee, \leftrightarrow$) and quantified variables ($\forall X$, $\exists Z$) must be converted into a set of clauses. Here, standard techniques (see [Loveland, 1978] and Section 3.2.2) and improvements thereof (e.g., [Nonnengart *et al.*, 1998]) are being used. Modules for transformation are for example: plop [Schumann and Ibens, 1998] for SETHEO, a module built into OTTER [McCune, 1994a], flotter [Weidenbach *et al.*, 1996] for SPASS, or ProSpec for Protein [Baumgartner and Furbach, 1994b].

The topic of generating clauses is also of particular interest if a full first-order proof is to be generated out of the machine proof during postprocessing (Section 7.7). For the generation, information about quantified variables and the structure of the formula is required which otherwise can be easily lost during generation of clausal normal form.

Simplification: This step tries to remove parts of the formula which are redundant and produces a formula which is simpler in structure and complexity without affecting its provability. This step turned out to be extremely important for most applications. It is covered in Section 7.3.

Theory and sort preprocessing: Many proof tasks are based upon some theory (e.g., equality, theory of lists) and are given in a sorted logic. In Section 7.4, we present methods which allow an ATP for pure first-order logic to efficiently handle sorts. For handling of equations within SETHEO, we used the axiomatic representation or E-SETHEO's compilation technique (Section 3.3.2). Since no single method is a clear winner in applications, both methods should be considered.

Integrating efficient means for handling theories in top-down theorem proving is still an evolving research area. Although combination techniques like linear completion [Baumgartner and Schumann, 1995] have a clear potential, they are still not working satisfactory. Resolution-style theorem proving facilitates the combination between deduction and theory reasoning (like it is done in AMPHION, see Section 4.7.1). However, many problems concerning control and efficiency are to be solved.

Preselection of axioms: Usually the underlying theory as well as additional assumptions are defined as a set of axioms. Since additional, but useless axioms increase the search space, care must be taken to only generate and preselect required axioms. This topic is an important issue. Fortunately, the axioms in most applications are well structured according to a hierarchy of theories There, straightforward and powerful methods (like that described in Section 5.7) can easily be implemented. Such a mechanism of preselection should be an integral part of all automated theorem provers.

Simplification again: Virtually all steps shown in Figure 7.1 give rise to further simplification. Therefore, simplification should be performed continuously during all steps.

7.1.3 Execution Phase: Running the ATP

Finally, the proof task is in a format suitable for the automated prover which in turn is started and eventually returns a result. For practical usability, however, the prover must fulfill strong requirements (Section 5.9) with respect to answer times, effectiveness, soundness, and completeness. Therefore, in Section 7.6 we will focus on the following issues:

Finding good parameters: Each modern automated prover has dozens of parameters to set. Which ones are good? (Section 7.6)

Enhancement by combination of search paradigms: In Section 7.6.1 we describe a combination of top-down and bottom-up search for the prover SETHEO, increasing its deductive power.

Enhancement by parallel execution: Specific parallel execution models can be used to enhance efficiency and decrease answer times of automated theorem provers. With the availability of much computing power (e.g., networks of fast workstations) such models have become extremely important for applications (Section 7.6.2).

Handling of non-theorems: Due to the semi-decidability of first-order logic, most theorem provers do not terminate when given a non-theorem. We present a technique using first-order model generation for finding counterexamples and a generative approach for providing feedback in case a proof fails (Section 7.5).

7.1.4 Postprocessing Phase

After a successful run of the automated prover, its result must be returned to the application system. If only a binary value (proof found/no proof found) is required, things are easy. Often, additionally, answer substitutions are required. In Section 7.7.1 we describe how such information can be assembled by SETHEO without much overhead. More difficulties arise when a proof is required. In Section 7.7.4, we present ILF-SETHEO, a system which converts a machine proof (e.g., from OTTER or SETHEO) into a natural-deduction style, human-readable proof. This proof then can be automatically typeset using LaTeX.

7.2 Handling of Non-First-Order Logics

If the source logic of the application is not the same as the target logic (in our case first-order predicate logic), a transformation of the proof task must be carried out. Depending on the source logic and the "distance" between source and target, such a transformation can be very simple and straightforward, or can be extremely complex if not impossible (e.g., for "real" higher-order logic).

In many cases, the transformation is already done in the application system, resulting in proof obligations which are already in first-order logic (e.g., in NORA/HAMMR, Section 6.3). Here, we only will focus on two important topics: handling of inductive (first-order) problems, and processing of modal and temporal logics.

7.2.1 Induction

A well-known proof principle is *induction*. Inductive proof schemata are higher-order, because they talk about formulas. Thus, they cannot be directly processed by a first-order theorem prover. Because most data structures in programming languages are defined recursively (e.g., strings, lists, trees), and all loops (iterations or recursions) lead to recursive formulas. Most proof obligations, arising in the area of software engineering require induction. In order to carry out an inductive proof, the following information must be provided:

1. variable(s) over which induction is to be carried out ("induction over i"),
2. an appropriate induction scheme, e.g., $n-1 \to n$, or $1, 2, \ldots, n \to n+1$, and
3. induction hypotheses (and additional lemmata).

Then, the original proof obligation can be split up into (in general) two cases ("base case" and "step case")[4] and the resulting first-order formulas can be processed separately. In general, carrying out inductive proofs can be extremely complicated. Difficulties which require much thought arise when weak (i.e., generalized) induction hypotheses are needed, or when specific induction schemes (and orderings) are to be "invented". These cases are certainly way beyond automatic processing and must be left to the user.

Many proof obligations which arise in software engineering applications, however, are fairly "standard". This means that points 1,2, and 3 are known and no specific generalized hypotheses or lemmata are required.

Example 7.2.1. Let us consider the following proof obligation (taken from case study in Section 6.3) which tries to match two different specifications of the function *hd*:

$$\forall l : \mathsf{list}\ l = [\,] \vee (m = [hd\ l] \to (\exists i : \mathsf{item}, \exists l_1 : \mathsf{list} \cdot l = [i]\,{}^\frown l_1 \wedge m = [i]))$$

From the definition of the proof task, it is obvious that induction is to be performed over l : list. The induction has to be performed on the recursively defined data type list. This means that we have — if we abbreviate the above formula by $\mathcal{F}(l)$ — the following induction scheme: the base case $\mathcal{F}(l \backslash [\,])$ where l is substituted by the empty list [], and the step case

$$\forall l_0 : \mathsf{list}\ \forall i : \mathsf{item}\ \ \mathcal{F}(l \backslash l_0) \wedge l = [i]\,{}^\frown l_0 \to \mathcal{F}(l).$$

[4] When induction is to performed over n distinct variables, up to 2^n cases have to be explored, unless simultaneous induction can be performed.

Full power for finding inductive proofs can only be obtained by a combination between inductive theorem provers (e.g., Oyster/Clam [Bundy *et al.*, 1990]) and high performance provers for first-order logic. This research topic, however, will not be covered in this work. Rather, we will focus on *approximation* of induction based on the available information (about variables and data types). Given the recursive (higher-order) definition of the data types, it is often possible to automatically generate induction schemata and appropriate lemmata. We will have a look at three methods of approximation, namely trying to solve the problem without induction only with the help of appropriate lemmata, the classical case-splitting approach (without generalization), and a simpler version, "poor man's induction". It is obvious that none of the approximations are complete. Nevertheless they seem to be suited for practical purposes, as we will show with experimental data obtained from our case study in Section 6.3.

Automatic Generation of Induction Schemes. When data types are defined recursively (e.g., lists, streams, natural numbers), they usually give rise to inductive proof tasks. However, it is possible to formally derive induction schemata and lemmata from a given higher-order recursive definition. With the help of an interactive higher-order prover, it is even possible to do this generation (and verification) in a semi-automatic way. A further advantage of this approach is that from these definitions rewrite rules for simplification (see Section 7.3) can be easily generated.

In general, supplying appropriate induction schemata and lemmata is done by the application system. However, it is interesting to investigate the possibilities of having such an induction-scheme generator as a stand-alone preprocessing module for an automated theorem prover. In that case, the proof task would be augmented by the recursive higher-order definitions of the data types used in the (sorted) formula. Then, induction schemata are correctly generated and applied to the proof task, before it is processed by the automated prover. Since this generation involves a lot of overhead (and possibly user interactions), such a generation should be made only once for each data type, and the produced schemata and lemmata should be kept in a data base for fast access by the preprocessing modules and the simplifier.

Example 7.2.2. Let us consider a theory of *lists*. In [Regensburger, 1994], HOLCF is used as a framework to define the theories and to verify higher-order theorems (e.g., induction schemata) and first-order lemmata. This example defines polymorphic lists "α `list`" as initial solutions of the equation α `list` $= \alpha * *(\alpha$ `list`$)u$. More explicitly, lists are defined as an extension to the theory of natural numbers (`Dnat`) in the following way (from [Regensburger, 1994]):

```
List          = Dnat  +

consts
list_rep      :: (α list) → (α **(α list)u)
list_abs      :: (α **(α list)u) → (α list)
cons          :: α → α list → α list
...
lhd           :: α list → α
ltl           :: α list → α list
...
rules

list_abs_iso   list_rep[list_abs[x]] = x
list_rep_iso   list_abs[list_rep[x]] = x
...
cons_def       cons ≡ (Λx l.list_abs[x##up[l]])
lhd_def        lhd ≡ list_when[⊥][Λx l.x]
ltl_def        lhd ≡ list_when[⊥][Λx l.l]
```

First, type declarations are given for all theory constants (representation, abstraction, constructor `cons`, head `lhd` and tail `ltl`). The section `rules` defines axioms, describing important properties of the given constants, here as an example the isomorphism of the combination of `list_rep` and `list_abs`, and the well-definedness of `cons`, head `lhd` and tail `ltl`. For the entire specification and details on syntax and semantic see [Regensburger, 1994], or the Isabelle distribution [URL Isabelle, 2000] which contains all theories.

Based on this definition induction schemata and simplification rules can be generated using Isabelle's HOLCF tactics [Paulson, 1994]. For details on the proofs see again [Regensburger, 1994]. From the variety of different induction schemata (e.g., finite induction, coinduction), here we only present the general-purpose induction scheme: (`adm(P)` means that first-order formula `P` is admissible, [] is the empty list):

```
list_ind [[adm(P);P([]);⋀x s1.[[x¬=[];P(s1)]]
          ⟹ P(cons[x][s1])]] ⟹ P(s)
```

As expected, we first have to show the base case `P([])`, then the induction step `s1` → `cons[x][s1]`.

Induction by Lemmata. This approach tries to avoid a proof by induction by adding appropriate lemmata to the proof task. Of course, this method is incomplete, because for proper handling of recursive definitions, an infinite number of axioms would have been to be added. In many practical applications (e.g., our case studies in Chapter 6), however, adding some obvious lemmas helps to find proofs for simple, inductive proof tasks. An advantage of this approach is that the overhead of induction (i.e., adaptation of induction

scheme, splitting up the proof task) is circumvented. Lemmata which can be used for this approach can even be generated from the underlying recursive definition as described above in Section 7.2.1.

Example 7.2.3. For the data type list, the following axiom helps to prove many proof tasks which otherwise would require induction:

$$\forall l' : \mathsf{list} \cdot \exists l, m, r : \mathsf{list} \cdot l' = l \hat{} m \hat{} r$$

The axiom means that each list consists of a front part l, a middle part m, and a tail part r.

Approximation of Induction by Case Splitting. The standard way of approximating induction is by splitting up the formula into separate proof tasks, the base and step cases. The original conjecture is used as an (ungeneralized) induction hypothesis.

Example 7.2.4. For a proof task of the form $\forall l : \mathsf{list} \cdot \mathcal{F}(l)$ and induction on the list l, we obtain the following cases:

$$\forall l : \mathsf{list} \cdot l = [] \rightarrow \mathcal{F}(l)$$
$$\forall l : \mathsf{list} \cdot \forall i : \mathsf{item}, l_0 : \mathsf{list} \cdot \mathcal{F}(l_0) \wedge l = [i] \hat{} l_0 \rightarrow \mathcal{F}(l)$$

Although it is sufficient to have these two cases, it can be of interest to generate an additional case, dealing with a one-element list:

$$\forall l : \mathsf{list} \cdot \forall i : \mathsf{item} \cdot l = [i] \rightarrow \mathcal{F}(l)$$

The reason for that is that many specifications are valid for non-empty lists only. For those specifications, case 1 above would be a trivial proof task.

After rewriting the formula accordingly, we obtain a number of independent first-order proof tasks. Only if all of them can be proven, the entire proof task is considered to be solved. When the formula is split up, simplification is of major importance (cf. Section 7.3). Often the base cases can be directly simplified to TRUE or FALSE without having to start the prover itself.

Poor Man's Induction. Approximation by "poor man's induction" is a simplification of the of the above approach. Instead of generating a full step case, the induction hypothesis is left out. Hence, we get the following induction scheme (again for lists):

$$\mathcal{F}([]) \wedge \forall l_0 : \mathsf{list} \cdot \forall i : \mathsf{item} \cdot \mathcal{F}([i] \hat{} l_0)$$

The main advantage of this approach is that the (static) size of the formula does not get as large as for the standard case (where essentially the conjecture is duplicated). In practical applications where only the simplest kinds of inductive proofs are required, poor man's induction is a way to reduce answer times. Is should be noted that the base case and step case should be treated as two separate proof tasks by the prover. Otherwise the search space (esp. for top-down provers) can increase tremendously.

Experimental Results. For a comparison of the different approaches to handle proof tasks with inductive problems, we take the set of proof obligations (all valid by construction) from Section 6.3 (case study on logic-based component retrieval). The following Table 7.1 shows the results of various experiments. For each method, we show the total number of proof tasks, the number of tasks, solved by SETHEO within a run-time limit of 60*s* (on a Sun ULTRA II), the number of fails (SETHEO did not discover a proof), and the number of errors (e.g., size of formula too large). The last column gives the recall, i.e., the percentage of solved tasks with respect to their total number. For methods which split up proof tasks, we show the results for the base- and step-case. Obviously, the success rate is higher with the (rather simple) base cases, whereas it drops with the larger and more complex step cases.

Method	number of tasks				recall
	total	solved	failed	error	%-solved
axioms only	1838	1039	730	69	**56.5%**
w/lemmas	1838	1271	498	69	**69.2%**
case-splitting	1838	1235	487	114	**67.2%**
base	1838	1658	111	69	90.2%
step	1838	1235	487	114	67.2%
poor man's	1838	1302	467	69	**70.8%**
base	1838	1658	111	69	90.2%
step	1838	1302	467	69	70.8%

Table 7.1. Experimental results on approximation of induction

None of the approaches is complete, and – although relatively similar in their overall recall – they solve a somewhat different set of examples. For example, compared to non-inductive treatment with lemmata, case splitting cannot solve 72 proof tasks. However, with case splitting, we can solve 36 tasks which cannot be solved otherwise. Therefore, the combination of these methods of handling inductive problems in a parallel competitive way (as discussed in detail in Section 7.6.2) is certainly a good idea. With such a combination of both methods, an overall success rate of 71.1% (1307 of 1838 solved) can be obtained.

7.2.2 Modal and Temporal Logics

Many formal methods are based on temporal or modal logics. These logics have additional quantifiers, e.g., $\circ P$ (nexttime P), $\Diamond P$ (eventually P), or P **until** Q, to express the behavior of the predicates in time or modalities. Therefore, modal logics are convenient to describe finite automata, state-transition systems, traces of actions, etc. In this book, we will not go into details of this topic. We rather refer to e.g. [Kröger, 1987; Gabbay and Guenthner, 1983; Abramsky *et al.*, 1992] for further reference.

The semantics of the temporal (or, in general, modal) quantifiers is usually given as a Hilbert system (i.e., as a language and a number of inference rules), or as a Kripke semantics (possible worlds).

Automatic processing of certain kinds of propositional modal logics can be done very efficiently by model checkers. E.g., SMV [McMillan, 1993] can prove formulas of a temporal logic, called computation-tree logic (CTL). However, there don't exist powerful automated provers for modal predicate logics. Therefore, it is important to *translate* formulas in such a logic into first-order logic (FOL) and process the resulting formulas by a first-order automated prover. In this book, we describe two approaches: a direct encoding of a Hilbert system in first-order logic, and the translation of modal and temporal quantifiers into FOL, using the SCAN algorithm.

Hilbert-style Transformation (T-Encoding). In this approach, proof obligations in a modal or temporal logic are directly translated into first-order logic by using a "meta-approach": for a formula $\mathcal{F}$ in the source logic M, we define:

$$\mathbf{T}(\mathcal{F}') \equiv \text{TRUE} \Leftrightarrow \mathcal{A} \vdash_M \mathcal{F}$$

The first-order predicate $\mathbf{T}$ which takes the translated formula as its argument holds, if and only if $\mathcal{F}$ can be derived from a set of axioms $\mathcal{A}$ in our logic M. The translation of $\mathcal{F}$ just represents the entire formula as a first-order term $\mathcal{F}'$. Additionally, all inference rules of the Hilbert system, defining M must be translated into first-order formulas. For an inference rule of the form

$$\frac{\mathcal{F}_1 \ldots \mathcal{F}_n}{\mathcal{G}}$$

we obtain: $\mathbf{T}(\mathcal{F}'_1) \wedge \ldots \wedge \mathbf{T}(\mathcal{F}'_n) \rightarrow \mathbf{T}(\mathcal{G}')$.

Example 7.2.5. Propositional Hilbert systems with $\rightarrow$ (implies) as its only logical connective and a minimal numbers of axioms have been studied by Łukasiewicz [Łukasiewicz, 1970]. The only inference rule is modus ponens. Then, one has to show that for any valid propositional formula (e.g., $A \rightarrow A$) a derivation using the given axioms and modus ponens can be found. For processing these proof tasks with a first-order prover (e.g., OTTER [McCune, 1993; McCune and Wos, 1991; Pfenning, 1988]) the above approach can be used. The modus ponens inference rule

$$\frac{P \quad P \rightarrow Q}{Q}$$

is translated into:

$$\mathbf{T}(P) \wedge \mathbf{T}(\mathtt{i}(P, Q)) \rightarrow \mathbf{T}(Q)$$

where `i` is the term representation of $\rightarrow$. A typical single axiom, e.g.,

$$(((P \rightarrow Q) \rightarrow (R \rightarrow S)) \rightarrow (T \rightarrow ((S \rightarrow P) \rightarrow (R \rightarrow P))))$$

becomes:

$$\forall P, Q, R, S, T \cdot \mathbf{T}(i(i(i(P,Q), i(R,S)), i(T, i(i(S,P), i(R,P)))))$$

Even simple theorems like $\mathbf{T}(A \rightarrow A)$ comprise hard problems for automated theorem provers [McCune, 1993; Veroff and McCune, 2000], because the proofs usually require many inference steps.

Example 7.2.6. The BAN logic of our case study in Section 6.1.2 is a multi-sorted modal logic, defined by a set of inference rules. This approach has been used to translate formulas of the BAN logic into first-order logic. All inference rules (Table 6.2 on page 104) have been translated into Horn clauses. For example, the so-called nonce-verification rule

$$\frac{P \models \#X \qquad P \models Q \mid\sim X}{P \models Q \models X}$$

is translated into

$$\forall P, Q, X : \mathbf{T}(P \models' \#' X) \wedge \mathbf{T}(P \models' Q \mid\sim' X) \rightarrow \mathbf{T}(P \models' Q \models' X).$$

Since all BAN formulas consist of a conjunction of atomic BAN formulas, the atomic formulas can be translated independently from each other.

Ohlbach's Transformation. This transformation of formulas in a modal logic works by introducing "world-transition relations" and relating it to Kripke-style semantics [Ohlbach, 1988]. This transition between worlds is accomplished by augmenting the formulas with a term $\mathcal{W}$, denoting the current "world-situation". Thus, for an atom $p(t_1, \ldots, p_n)$ in the source logic, we obtain

$$p(t_1, \ldots, p_n, \mathcal{W}).$$

First-order axioms, defining the behavior of the modal quantifiers modify these world variables.

Often, the semantics of the modal quantifiers is given as relations, e.g., $\Diamond\Diamond A = \Diamond A$ or $\Box \circ P = \circ\Box P$. These relations can be translated into a set of first-order formulas using the SCAN algorithm [Ohlbach, 1993; URL SCAN Algorithm, 2000]. As shown in Figure 7.2, these properties are entered into the SCAN algorithm which (in case it terminates) results in a set of axioms which are added to the syntactically translated formula. The resulting first-order formula can be very efficiently processed by an automated prover (cf. [URL MSPASS, 2000]). Obviously, the SCAN algorithm only needs to be invoked once for each modal logic.

Example 7.2.7. Let us consider safety property LS^1 (Table 6.8) of case study 6.2:

$$\forall s, t : \mathsf{trace}\ \forall e : \mathsf{data} \cdot s \sqsubseteq t : \mathrm{get}_R(e)\ in\ s \Rightarrow \mathrm{put}_T(e)\ in\ s \qquad (7.1)$$

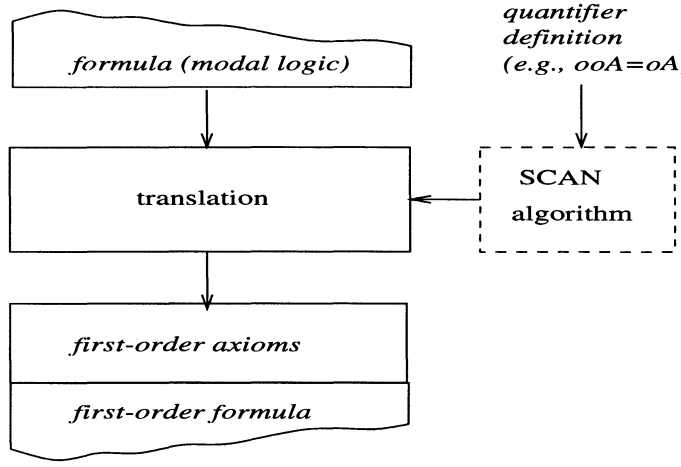

Fig. 7.2. Translation of a modal proof task

Here, get_R and put_T are actions which can be in a trace. If we define a simple temporal logic, the FOCUS operator *a in s* (action *a* occurs in trace *s*) would read: $\Diamond_s a$ (*a* eventually occurs in trace *s*). For the prefix operator $\sqsubseteq$ we would yield the **until**. Then (7.1) would be transformed into:

$$s \textbf{ until } t \wedge \Diamond_s \text{get}_R(e) \rightarrow \Diamond_s \text{put}_T(e)$$

The properties of **until** are: reflexive, anti-symmetric and transitive, one property of the eventually operator would be: $\Diamond_s \Diamond_s A = \Diamond_s A$. When applying the SCAN algorithm, a complete set of first-order axioms can be obtained. They capture essentially the same properties as the axioms set up manually for our case study. Thus, similar run-time results can be expected[5].

Assessment. The approach of translating modal logics using T-encoding is very general. Each Hilbert system can be translated in such a way. However, the resulting formula exhibits a high complexity and usually results in extremely hard proof obligations, especially when the number of inference rules is considerably large. [Ohlbach, 1998] proposes an optimization of this approach. Here, semantic knowledge about logical connectives is used to reduce the search space, otherwise induced by the translation of the inference rules. Typically, this approach can be used when the Hilbert system under consideration contains operators which have the usual first-order semantics (like $\wedge, \vee, \rightarrow$). However, controlling the automated prover is very complicated and not solved yet in a satisfactory way.

[5] Because the representation of FOCUS using a temporal logic involves detailed knowledge about the semantics of FOCUS, this translation has not been made.

On the other hand, using the SCAN algorithm for the translation into first-order logic usually yields an efficient representation. However, the application of that approach is limited to logics which are fully defined and have a clean denotational semantics. For example, the BAN logic (Section 6.1.2 and Example 7.2.6) has no denotational semantics as it is only defined by its alphabet and inference rules. Hence, Ohlbach's transformation cannot be used in that case.

Regardless of the method of transformation used it is advisable to use as much semantic knowledge as possible to obtain formulas which can be handled efficiently with a first-order theorem prover.

7.3 Simplification

As pointed out in Section 4.2.2, one can observe that in most applications, proof tasks contain many *redundant* parts. These and redundancies during the search lead to large search spaces which must be traversed by the automated theorem prover. Then even trivial proof tasks can consume too many resources or cannot be solved within reasonable times at all.

Interactive theorem provers and verifiers (e.g., PVS, Isabelle, KIV, HOL) have built-in simplifiers for processing the formulas at hand. As demonstrated by most examples and applications, simplification is one of the most powerful tactic available. Therefore, considerable effort has been spent on developing such simplifiers (cf., e.g., [Reif *et al.*, 1998]). Also algebraic manipulation systems (e.g., Mathematica [Wolfram, 1991] or Maple [Char *et al.*, 1988]) obtain much of their usability from powerful and sophisticated simplification algorithms. However, many important aspects of simplification have been neglected by designers of automated theorem provers.

Simplification usually is done, using *rewriting* of the formula (or parts thereof). It can be performed as a preprocessing step (static simplification) or during the search for a proof (dynamic simplification). A further way to classify simplification is the kind of information used for simplification. We can distinguish between *logic simplification* which only takes the syntactic structure of the formula into account. In contrast, *semantic simplification* is often more powerful, but it needs to know the meaning of the function and predicate symbols occurring in the formula. Of course, it is always necessary that the logical properties of the formula, at least its provability, is retained. Otherwise, an unsound deduction system would be the consequence.

There are many approaches for performing simplification and many systems based on rewriting. Examples are the Larch prover [Guttag *et al.*, 1993], Maude [Clavel *et al.*, 1996], OBJ3 [Goguen *et al.*, 1993], RRL [Kapur and Zhang, 1989], or ReDuX [Bündgen, 1998]. These topics are also covered in a series of conferences, most prominently the *Conference on Automated Deduction* (CADE) or *Rewriting Techniques and Applications* (RTA). For an

overview, see, e.g., [URL ARS-Repository, 2000]. In the following, we will focus on several methods for simplification which are straightforward and easy to implement, but which are very powerful in combination with automated theorem provers.

7.3.1 Static Simplification

Static simplification is performed as a preprocessing step[6]. However, it is usually advisable to simplify the formula repeatedly after each major preprocessing step. Thus, a formula should first be simplified on the level of the source logic, then after translation into first-order logic and again when the formula has been transformed into clausal normal form, and so on. Each step (see Figure 7.1 on page 138 for a typical example) opens up new possibilities for simplification, because transformation of the formula usually introduces redundancies.

Logic Simplification. Logic simplification only takes the syntactic structure of the formula into account when trying to make the formula smaller and easier to prove. In this area, simplification gets very close to preprocessing as described in, e.g., [Bayerl *et al.*, 1988; Letz *et al.*, 1988; Letz *et al.*, 1992]. Typical steps for simplification include removal of

- TRUE and FALSE atoms from the formula,
- pure parts, i.e., sub-formulas which cannot contribute to a proof, and
- subsumable parts.

Example 7.3.1. Rules for simplification are usually written as inference rules or rewrite rules. They are used as long as they are applicable. For example, some rules for eliminating TRUE are:

$$\frac{\mathcal{A} \wedge \text{TRUE}}{\mathcal{A}} \qquad \frac{\mathcal{A} \vee \text{TRUE}}{\text{TRUE}} \qquad \frac{\mathcal{A} \rightarrow \text{TRUE}}{\neg\mathcal{A}} \qquad \frac{\text{TRUE} \rightarrow \mathcal{A}}{\mathcal{A}}$$

Here, we will use a notation like $\mathcal{A} \wedge \text{TRUE} \Longrightarrow \mathcal{A}$.

Example 7.3.2. A formula usually contains pure parts when a conjecture is to be proven which only requires some of the properties of the formula. A proof task about a non-empty and sorted list l of items might be formalized as $\neg \mathit{empty}(l) \wedge \mathit{issorted}(l)$. When the rest of the formula only concerns the predicate *empty*, all occurrences of *is_sorted* can be removed from the formula. In formula $\mathcal{A} \vee (\mathcal{A} \wedge \mathcal{B})$, sub-formula $\mathcal{A}$ subsumes the right-hand side. Thus, this formula can be simplified to $\mathcal{A}$.

[6] However, techniques of simplification can also be used to simplify proofs found by the automated prover. This approach can be very helpful when proofs are to be presented in a human-readable form (see Section 7.7).

A further and wide field for simplification is the expansion of definitions. Often, new symbols are introduced by a bi-conditional *new* $\leftrightarrow$ *old* to abbreviate longer sub-formulas or express recursion. For example, the inequality predicate *neq* could be defined as $neq(X,Y) \leftrightarrow \neg equal(X,Y)$. In many cases, expanding the definitions with subsequent simplification has considerable effects on the complexity of the formula in particular when also semantic information is present (see below). Since the size of the formula usually increases, expansion of definitions can also have the adverse effect. This rather difficult topic is related to the pre-selection of axioms.

Equational simplification (in particular when combined with semantic simplification) can be arbitrarily hard (e.g., [Bündgen, 1998]). A straightforward simplification which turned out to be extremely helpful in the case studies is shown in the following example.

Example 7.3.3. Often, a conjecture to be proven has the form

$$\forall X \cdot (X = t \rightarrow \mathcal{F})$$

where t is a term t where X does not occur. When we want to refute the negation of this formula, conversion into clausal normal form and Skolemization yields

$$a_X = t \wedge \neg \mathcal{F}[X \backslash a_X]$$

with a Skolem constant a_X. This formula can be simplified to a formula where all occurrences of a_X (and hence those of X) in $\mathcal{F}$ are replaced by the term t and the equation is dropped. As the result, we obtain $\neg \mathcal{F}[X \backslash t]$.

In our case study on component retrieval (Section 6.3) a proof task typically relates the input and output parameters of the query ($I_1^q, \ldots, I_n^q$, and $O_1^q, \ldots, O_m^q$, respectively) and those of the library component (index c):

$$\begin{array}{l} \forall I_1^q, \ldots, I_n^q, O_1^q, \ldots, O_m^q \cdot \forall I_1^c, \ldots, I_n^c, O_1^c, \ldots, O_m^c \cdot \\ \quad (\; I_1^q = I_1^c \wedge \ldots \wedge I_n^q = I_n^c \wedge O_1^q = O_1^c \wedge \ldots \wedge O_m^q = O_m^c \\ \qquad \rightarrow \mathcal{F} \;) \end{array}$$

Even without first transforming the formula into clausal normal form, we can replace all occurrences of I_i^q and O_i^q in $\mathcal{F}$ by I_i^c, and O_i^c, respectively, and getting rid of all equations, relating input and output variables. This simple rewriting step has been implemented for our case study on component retrieval as a simple AWK-script [Aho *et al.*, 1988] within a few minutes. Nevertheless, the run-time for many problems could be reduced at least by an order of magnitude.

Logic simplification is also important when a formula is transformed into clausal normal form. Both, the simple algorithm (cf. [Loveland, 1978]) or algorithms which introduce new predicate symbols to overcome the exponential explosion [Eder, 1984; Boy de la Tour, 1992] can result in non-optimal sets of clauses. Here, simplification during the conversion and additional heuristics (e.g., to generate a new predicate symbol only if the resulting formula gets smaller) as described in [Nonnengart *et al.*, 1998] yield good results.

Dynamic Simplification. Dynamic simplification tries to simplify the formula or the data structures created during the search (sets of resolvents or tableaux). For automated provers working on sets of clauses, there is not much room for boolean simplification. However, the detection of subsumable and tautological parts play an important role. Bottom-up provers (like OTTER, SPASS, Gandalf) very much rely on efficient subsumption tests for their newly generated clauses. In SETHEO, detection of tautological or subsumable parts of the current tableau is detected using constraint techniques as described in Section 3.3.2.

7.3.2 Semantic Simplification

This kind of simplification uses additional information about the *sorts* of the variables and the *definition* of the data structures for simplification. This information is also represented as rewrite rules for the simplifier which can be used during preprocessing or the proof search. From the definition of the data structures, such rewrite rules can be extracted or generated by a higher-order theorem prover.

Example 7.3.4. Let us consider the data type list with its formal definition shown in Example 7.2.2. Based on this definition, simplification rules as shown in Figure 7.3 below can be generated automatically using HOLCF on top of Isabelle [Regensburger, 1994]. Within this framework also the correctness of these rules can be proven formally. These proofs, however, are lengthy and require user interactions. Nevertheless, for many applications of automated provers, such an approach can be of great interest, because generation of such simplification rules must be performed only once for each data type.

The simplification rules shown in Figure 7.3 are those one would expect and require no further explanations. In [Regensburger, 1994], these rules are used to augment Isabelle's built-in simplifier (the tactic `simp_tac`). For our purposes, these rules are used for simplification during preprocessing and proof search.

Static Semantic Simplification. In contrast to rewrite-based theorem proving, only the correctness of the simplification rules are important. The set of rules need not be confluent. Inconfluent rewrite rules only reduce the effect of the simplifier, but do not sacrifice completeness[7]. Therefore, no time-consuming completion algorithms [Knuth and Bendix, 1970] are needed.

[7] Instead of always generating a normal form, we can get stuck at some other place. Since we do not have to reduce the formula to TRUE or FALSE during preprocessing, this case is not harmful.

$$
\begin{array}{l}
\ldots \\
\texttt{x1}\neg = \bot \Longrightarrow \texttt{lhd[lcons[x][x1]]} = \texttt{x} \\
\texttt{x}\neg = \bot \Longrightarrow \texttt{ltl[lcons[x][x1]]} = \texttt{x1} \\
\ldots\texttt{lnil}\neg = \texttt{lcons[x][x1]} \\
\texttt{lhd[lcons[x][xs]]} = \texttt{x} \\
\texttt{lcons[}\bot\texttt{][xs]]} = \bot \\
\texttt{lcons[x][}\bot\texttt{]]} = \bot \\
\ldots
\end{array}
$$

Fig. 7.3. Some simplification rules for lists (from [Regensburger, 1994]). Simplification rules are written $LHS \Longrightarrow RHS$ or as equations. `lhd`, `ltl` are head and tail of a list, `lcons` the constructor, and $\bot$ the undefined list.

Beside the simplification steps described above, additional steps can be performed: application of induction schemes for simplification, considering domain sizes, and unfolding of definitions.

Induction Schemes. For a formula containing variables of a sort representing a recursively defined data type (e.g., list, nat), the appropriate induction scheme (Section 7.2.1) can always be applied. After generating the individual cases, many new possibilities open up for subsequent simplification. Often, a simple one-step unfolding of the quantifiers is sufficient to simplify the entire formula to TRUE or FALSE.

Example 7.3.5. For lists of sort list of item and a formula of the form $\forall X : \mathsf{list} \cdot \mathcal{F}(X)$, we can generate:

$$\mathcal{F}[X\backslash[]] \ \wedge \ \forall H_1 : \mathsf{item}\ \forall T_1 : \mathsf{list} \cdot \mathcal{F}[X\backslash\texttt{lcons}(H_1, T_1)]$$

Each individual part of the formula now can be simplified. In particular simplification rules concerning the empty list are very powerful in both cases, because in the first formula the empty list occurs, whereas in the second X is substituted by a term which is not equal to [].

As shown in our case study 6.3 this kind of simplification turned out to be extremely powerful. With standard rewriting and the technique just described, more than 40% of the non-valid proof tasks already could be rewritten to FALSE (Table 6.15 on page 133 and [Fischer *et al.*, 1998]).

Domain Sizes. For formulas in a sorted logic, the sizes of the domains are usually known. For simplification, one only has to know if domain $\mathcal{D}$ for sort s is empty, only contains one element, or has more than one element, i.e., $|\mathcal{D}_\mathsf{s}| = 0, 1$ or $|\mathcal{D}_\mathsf{s}| > 1$. Then, formulas containing equations about quantified variables can often be simplified as shown in the following example.

Example 7.3.6. Again, consider lists. Obviously, $|\mathcal{D}_\mathsf{list}| > 1$. Hence, a formula $\forall X : \mathsf{list} \cdot X = [] \wedge \mathcal{F}$ can be immediately rewritten to FALSE.

Unfolding of Definitions. When combined with semantic simplification, unfolding of definitions can be much more powerful than its logic variant, described above. Usually, definitions are used to abbreviate longer sub-formulas, or to express recursion. Hence, quite often definitions contain parts for handling specific cases, e.g., to handle the empty list. With a single unfolding step, the body of the definition is inserted into the formula. Then, another simplification step is attempted on the individual cases.

Example 7.3.7. A predicate *myhd* taking a list of natural numbers might be defined as follows (if the input list is empty, return 0, otherwise the head element):

$$\forall H : \mathsf{nat}\ \ \forall L : \mathsf{list} \cdot\ \mathit{myhd}(H, L) \leftrightarrow \\ ((L = [\,] \rightarrow H = 0) \wedge\ (L \neq [\,] \rightarrow H = \mathit{hd}(L))$$

A formula containing $\mathit{myhd}(X, [\,])$ can be simplified by unfolding this definition with subsequent simplification. In our case, we would yield a substitution $X \backslash 0$. The literal $\mathit{myhd}(X, \mathsf{cons}(F, R))$ can immediately be simplified to $X \backslash F$.

Such an approach can have quite dramatic effects as demonstrated in the following example. Because unfolding of definitions usually increases the size of the formula, this step should only be performed when the size or complexity of the formula gets lower after the final simplification step.

Example 7.3.8. This example has been taken from a case study on the verification of refinements in AUTOFOCUS [Schumann and Breitling, 1998]. Here, a closure of the transition relation S with states z and z' and input and output streams (is, os) is defined recursively as

$$\begin{aligned} S[z, is, os, z'] \leftrightarrow \quad & is = \langle\rangle \wedge os = \langle\rangle \wedge z = z' \\ & \wedge\ \exists z'', x, i', y, o'\ \ is = i' \cdot x \wedge os = o' \cdot y \\ & \qquad \wedge S[z, i', o', z''] \wedge D[z'', x, y, z'] \end{aligned}$$

Here, $\langle\rangle$ means the empty trace, $\cdot$ the concatenation of traces, and $D[z, x, y, z']$ the transition relation itself. This definition is split up into two cases, the case where both traces is and os are empty, and the otherwise case. In the latter case, the definition is recursive.

Let us consider a proof task which occurred in one of the proof attempts in this case study. The conjecture was of the form

$$\ldots\ \ is = \langle\rangle \wedge os = o' \cdot y \rightarrow \neg S[z, is, os, z']$$

Without any simplification, SETHEO needed 3.1s to find a proof for that proof task. With elementary equational and logic simplification, the size of the formula could be reduced from 122 to 68 clauses. The run-time dropped to 1.38s. Unfolding the definition of S and subsequent simplification, i.e., throwing away the first case ($is = os = \langle\rangle$) saved another 17 clauses and reduced SETHEO's run-time considerably to 0.01s.

Dynamic Semantic Simplification. Semantic simplification during the search for the proof usually is integrated into bottom-up theorem provers like OTTER. Here, rewrite rules are used as *demodulators* during resolution. For completeness reasons, these demodulators are processed by a Knuth-Bendix completion algorithm. For details see, e.g., [Wos, 1988]. This approach, however, is not feasible for top-down theorem provers like SETHEO, because most subgoals contain variables during the search. Here, simplification usually cannot provide much effect.

However, semantic simplification in conjunction with SETHEO can easily be performed when the proof search is split up into a top-down and bottom-up part, as done in DELTA (Section 7.6.1). Here, all unit-clauses generated in the bottom-up phase can be simplified before they are used as lemmata in the subsequent top-down run of SETHEO.

Another way of using simplification rules which can be represented as inequalities (e.g., $\texttt{cons}(X,Y) \neq []$) in SETHEO is to use SETHEO's constraint mechanism. Syntactic inequality constraints of SETHEO (Section 3.3.2) are usually used to prune the search space by restricting the ways variables can be bound. As described above, some sorts of dynamic logic simplification is realized this way. However, these constraints can also be used for carrying semantic information, as in the following example.

Example 7.3.9. For proof tasks with variables of sort list, all literals of the form $X = Y$ can be augmented by constraints, expressing that [] cannot be equal to any term of the form cons(...):

```
...
...., equal(X,Y),... : [X,Y] =!= [nil,cons(#Z)].
...
```

This technique simulates the dynamic application of the simplification rule `lnil ¬ = lcons[x][x1]` of Figure 7.3. By augmenting a formula taken arbitrarily from case study 6.3, the run-time of SETHEO could be decreased by almost 50%.

7.4 Sorts

For most practical applications of automated theorem provers, sorts are of central importance. Therefore, means for the efficient handling of sorted problems is vital, unless a theorem prover is used which directly handles sorted problems (e.g., SPASS). A general restriction for effectively handling sorted problems by an ATP is that the sort hierarchy forms an upper semi-lattice. Then, the unification of two sorted terms yields a unique most general unifier (cf., e.g. [Bahlke and Snelting, 1986; Walther, 1988]). Otherwise, unification creates new large search spaces for which no appropriate pruning mechanisms are implemented in provers. However, virtually all practical applications can be handled with this restriction. In general, there exist three approaches to

augment an automated theorem prover for first-order logic towards handling sorted logics: encoding of sorts as unary predicates, extending the unification algorithm, and compilation of sort information into the terms.

7.4.1 Sorts as Unary Predicates

Each sorted problem can be easily transformed into a problem for pure first-order language. For each sort s, we introduce a predicate symbol *is_of_sort_s* which checks the well-sortedness of a term t of that sort.

Example 7.4.1. A sorted formula

$$\forall H : \mathsf{nat}\ \forall T : \mathsf{list} \cdot \mathtt{cons}(H, T) \neq [\,]$$

can be translated into a formula of unsorted first-order logic:

$$\forall H\ \forall T\ \cdot \mathit{is\_of\_sort\_}\mathsf{nat}(H) \wedge \mathit{is\_of\_sort\_}\mathsf{list}(T) \rightarrow \mathtt{cons}(H, T) \neq [\,]$$

Furthermore, the definition of the sorts must be expressed as axioms and added to the formula. For example, for the lists, we would have to add:

$$\begin{aligned} &\mathit{is\_of\_sort\_}\mathsf{list}([\,]) \wedge \\ &\forall H\ \forall T\ \cdot \mathit{is\_of\_sort\_}\mathsf{nat}(H) \wedge \mathit{is\_of\_sort\_}\mathsf{list}(T) \rightarrow \\ &\qquad \mathit{is\_of\_sort\_}\mathsf{list}(\mathtt{cons}(H, T)) \end{aligned}$$

Although this transformation is straightforward, it is not advisable to use it in practice, because it induces huge search spaces. Checking the sort for a deeply nested term requires a long chain of inferences. In particular in tableau-based systems, this leads to proofs which are much deeper than those for pure predicate logic. This situation is shown in Figure 7.4. When performing iterative deepening, not only d levels, but $d + ds$ levels have to be searched. Thus proof tasks with many axioms usually exhibit unacceptable run-times.

7.4.2 Sorted Unification

Another approach to integrate handling of sorted logic into an automated theorem prover is to use *sorted* unification instead of ordinary unification during the search for the proof. Unification algorithms for sorted logics which have DAG-shaped sort hierarchies basically show the same complexity and features as ordinary unification algorithms. Usually, the sort hierarchy is transformed during a preprocessing step into an array, such that the determination of the sort for the unified term essentially is a table lookup (cf. e.g., [Bahlke and Snelting, 1986]). Also SETHEO can be extended in that way. [Beierle and Meyer, 1994] describes an extension to the Warren Abstract Machine which directly could be used for SETHEO. However, the implementation effort is considerable, because additionally all data structures for terms must be augmented to carry sort information.

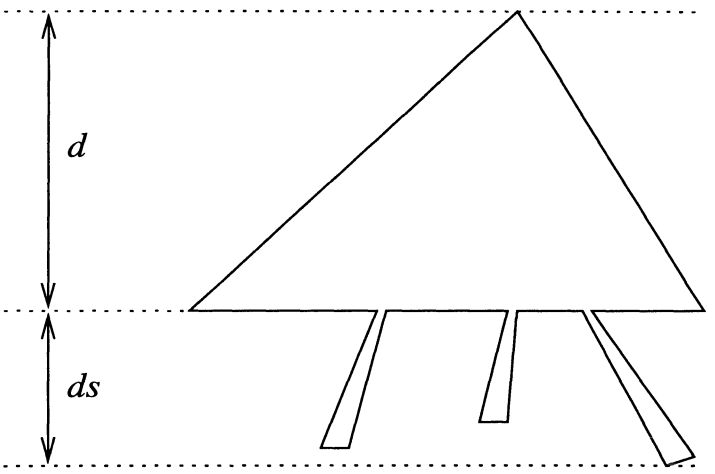

Fig. 7.4. Closed tableau for a formula in sorted logic. Reasoning is performed on the first d levels of the tableau, checking for well-sortedness requires another ds levels.

7.4.3 Compilation into Terms

In this approach, sort information is compiled into the terms of the formula during a preprocessing step. Then, ordinary unification of the theorem prover can check for the correct sorts. The basic idea is that for each sort s a term L_{s} is constructed such that

- $\mathsf{s}_1 \sqsubseteq \mathsf{s}_2$ (s_1 is a supersort of s_2) if and only if L_{s_1} is an instance of L_{s_2}.
- s_1 and s_2 have a common sub-type if and only if L_{s_1} and L_{s_2} are unifiable.

This approach is rather straightforward and has been extended to handle arbitrary DAG-structured sort hierarchies [Mellish, 1988]. ILF and Protein use this approach to encode sort information. Whereas the transformation integrated into ILF allows for DAG-structured sort hierarchies [Dahn, 1996][8], the module in Protein (called ProSpec) is restricted to tree structures. ProSpec has been used for the case studies described in this book.

The Transformation. Given a set of sorts S and a tree-shape sort hierarchy. The root of the tree is labeled $\top$. Then, a constant $a : \mathsf{s}$ with $\mathsf{s} \in S$ is compiled into

$$\mathtt{sort}(a, [\top, [\mathsf{s}_1, [\ldots, [\mathsf{s}_n, \mathsf{s}] \ldots]])$$

where $\mathsf{s}_1, \ldots, \mathsf{s}_n$ are the sorts on the branch in the sort hierarchy from the root node to sort s. `sort` is a new predicate symbol and $[h, t]$ denotes a

[8] The representation of DAG structures can require exponentially long terms L_{s} [Mellish, 1988] and has additional overhead.

PROLOG-style linear list, consisting of head h and tail t. For function symbols and predicate symbols a similar transformation is made. For a variable $X : \mathsf{s}$, a slightly different translation is necessary, because this variable can be instantiated with a term of a sort s' which is a subsort of s. Hence, we get

$$\texttt{sort}(X, [\top, [\mathsf{s}_1, [\ldots, [\mathsf{s}_n, [\mathsf{s}, V] \ldots])$$

where V denotes a new variable.

When we try to unify two terms $t_1 : \mathsf{s}_1$ and $t_2 : \mathsf{s}_2$ in their compiled form, the following conditions must hold:

- The terms t_1 and t_2 must be unifiable
- If one term is a variable, and $\mathsf{s}_1 \sqsubseteq \mathsf{s}_2$, then the list, representing s_2 is a sublist of that representing s_1, i.e., the latter is an instantiation of the first. Unification can be made because we introduced the new variable V at the end of the list.
- If both sorts are not compatible, i.e., $\mathsf{s}_1 \not\sqsubseteq \mathsf{s}_2$ and $\mathsf{s}_2 \not\sqsubseteq \mathsf{s}_2$, then s_1 and s_2 are located on different branches of the hierarchy. Hence, both lists have a segment which is not unifiable. Thus, the entire unification fails.

Example 7.4.2. Let us consider the following sorts $S = \{\mathsf{nat}, \mathsf{int}, \mathsf{float}, \mathsf{color}\}$ and sort hierarchy as shown in Figure 7.5.

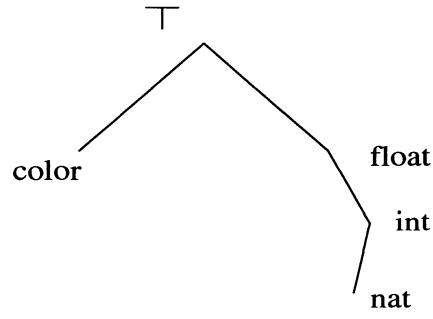

Fig. 7.5. A tree-structured sort hierarchy

Then, a term $-5 : \mathsf{int}$ would be compiled into

```
sort(-5,[top,[float,[int]]])
```

This term is unifiable with $Z : \mathsf{float}$ which is represented as

```
sort(Z,[top,[float,Y]]),
```

but not with $U : \mathsf{nat}$. Its representation is

```
sort(U,[top,[float,[int,[nat,V]]]]).
```

The sort ter of U, `[top,[float,[int,[nat,V]]]]` is not unifiable with `[top,[float,[int]]]`.

Thus, processing of sorted terms can be performed very efficiently, because checking of sorts can be done with a single unification step. However, with deep hierarchies or DAG-shaped sort hierarchies, terms can grow extremely large, causing the time needed for one unification to increase substantially.

Optimizations. When the formula and the sort hierarchy complies to certain restrictions, optimizations can be applied to the transformation. Usually, these restrictions can be detected with little overhead during preprocessing

- A formula is well-sorted , if all variable substitutions are conform with the given sort structure, i.e., for a variable $X : \mathtt{s}$, only terms $t : \mathtt{s}$ can be assigned during unification. Then, all sort information can be neglected without loosing correctness. In that case (which, e.g., we have taken advantage of in the case study in Section 6.2), no transformation is necessary, and thus no overhead occurs.
- If the sort hierarchy is *flat*, all branches of the tree are of length one. Therefore, a direct encoding of $a : \mathtt{s}$ into $\mathtt{sort}(a, \mathtt{s})$ (or even just $(a, \mathtt{s})$) as opposed to $\mathtt{sort}(a, [\mathtt{s}])$ is possible, reducing the size of the formula and increasing efficiency by several percent.
- A tree-structured sort hierarchy can be compiled as shown above. In case, the hierarchy is deep (i.e., resulting in long lists), simplification should be considered. In many applications, large sort hierarchies occur, because they have been assembled from different modules. Therefore, many sorts in a branch have just been introduced for clarity, but they serve no direct purpose. Therefore, such large trees can be broken down quite easily.
- Even in the case, two sorts have a common subsort (i.e., we have DAG-structured sort hierarchy), optimizations can be applied. Such techniques are implemented in ILF and help to reduce the number of different variables necessary to encode the sort information (cf. [Dahn, 1996]).

When integrating such a translation module (e.g., ProSpec) into the prover's preprocessing chain, care must be taken that this transformation does not interfere with other translations. Otherwise, unnecessary large formulas can be the result. For example, if equality-handling by STE modification (as done in E-SETHEO, Section 3.3.2) is performed, it is necessary to bypass the lists as generated by our sort transformation. Otherwise, these terms are subjected to STE-modification, resulting in unnecessary large terms and formulas.

7.5 Handling Non-Theorems

For most applications, the detection of non-theorems, i.e., of formulas which are not valid, and feedback on what went wrong is extremely important. As pointed out in Section 5.8 many proof tasks turn out to be not valid, in particular in early phases of verification, debugging of specifications and implementations, or logic-based component retrieval.

The first idea when trying to handle a non-theorem ($\mathcal{F}$) with an automated theorem prover, is to try out the negation of the original theorem ($\neg\mathcal{F}$). However, in most cases, $\neg\mathcal{F}$ is not valid either. Trying a PROLOG-style "negation as failure" usually does not work as well, because in most cases, the prover does not terminate. Here, resource-bounded reasoning might help: a formula is considered to be invalid if it cannot be shown by the prover within a certain time-limit. Such an approach can be practical in certain applications. However, it is very unreliable, since the notion of validity depends upon speed of machine, choice of the theorem prover and possibly even on the current system load.

In the area of classical automated theorem proving only few approaches to that topic can be found, e.g., model generation, disproving [Protzen, 1992], constraint-based techniques (e.g., [Caferra and Peltier, 1997]), or methods for debugging proofs as described in [Furbach, 1992; Furbach *et al.*, 1995]. Model generation theorem provers (like Satchmo [Schütz and Geisler, 1996], MGTP [Fujita *et al.*, 1992; Hasegawa *et al.*, 1992]) are specifically suited to find counterexamples for conjectures. However, they are restricted to range-restricted problems[9] or finite problems. These provers are complete and sound, however they perform very poorly in non range-restricted cases. Thus efficient implementations like Satchmo or MGTP are mainly used for planning and configuration management. In contrast, model checkers (like Finder [Slaney, 1992] or MACE [McCune, 1994b]) are decision procedures for first-order formulas restricted to finite domains.

Disproving [Protzen, 1992] is a specific technique for the detecting non-theorems. A specific sound and complete calculus allows to find models for universal formulas (i.e., formulas which only contain universal quantifiers). Extension to arbitrary formulas makes this approach incomplete and unsound. This calculus has been integrated into the interactive system INKA for verification of inductive problems [Hutter and Sengler, 1996]. Another approach extends resolution by constraints. By combining constraint solving and resolution, conditions for the existence of a model are assembled. [Caferra and Peltier, 1997] reports applications of this approach to debugging of logic programs. [Furbach *et al.*, 1998] use a tableau-based calculus ("Hyper-tableaux") for the debugging of proofs and the tuning of axiomatizations. In the area of interactive theorem proving and verification systems, the situation is slightly better, because some systems like PVS or KIV provide feedback in situations where a proof attempt got stuck.

In the following, we will present and assess two different techniques for detecting non-theorems and for providing feedback. They have been used successfully in our case studies. However, as we will see, none of the methods developed and found in the literature so far is suited alone to meet practical

[9] A formula is range-restricted, if all its clauses are range-restricted, i.e., all variables which occur in the head of a clause must occur in at least one of its tail literals.

requirements. As described in Section 5.8, an automated prover should be able to detect at least obvious and trivial non-theorems as such (due to FOL's semi-decidability, it never can detect all), and, if required, to provide feedback on the reasons why the formula is invalid.

7.5.1 Detection of Non-Theorems by Simplification

As demonstrated in Section 7.3, logic and semantic simplification is a very powerful method to reduce the complexity of a proof task. In many cases, simplification alone can reduce a formula to TRUE or FALSE. Because simplification is sound (as long as there are no inconsistencies in the simplifier's rewrite rules), it can be used to detect non-theorems: if a formula is reduced to FALSE, it is a non-theorem. If, however, simplification does not yield FALSE, there is no indication about the validity of a formula.

In our case study on logic-based component retrieval (Section 6.3) a simplifier has been used to detect as many non-theorems as possible, thus reducing the workload of subsequent filters and the theorem prover. Its evaluation [Fischer, 2001; Fischer *et al.*, 1998] with the experimental data base (containing more than 14,000 proof tasks, almost 90% of which are not valid) yielded impressive results. With the simplifier, 40.7% of the all proof tasks, i.e., 49.5% of the non-theorems, could be detected within short run-times below 2 seconds (on a Sun ultra-sparc, see Table 6.15). As shown in detail in that table, the effectiveness of detection of non-theoremhood strongly depends on the availability of semantic information. The more information available, the better the method can be. In particular, unfolding of inductive definitions with subsequent simplification (cf. Section 7.3 for details) proved very powerful. For example, for a formula $\forall X : \mathsf{list} \cdot \mathcal{F}$ we yield

$$\mathcal{F}[X\backslash[\,]] \ \wedge \ \forall H : \mathsf{item}\ \forall T : \mathsf{list} \cdot \mathcal{F}[X\backslash\mathsf{cons}(H,T)]$$

Experience showed that many obvious non-theorems can be detected already when considering the case of the empty list.

Example 7.5.1. Let us consider two component specifications with the functionality list $\rightarrow$ list, namely *rotate* (which puts the first element of a non-empty list at its end) and *length_of_list* which returns a singleton list containing a natural number denoting the length of the list. For example, we have $\mathit{length\_of\_list}[a,b,c] = [s(s(s(0)))]$.

$\mathit{rotate}(l : \mathsf{list})\ l' : \mathsf{list}$	$\mathit{length\_of\_list}(x : \mathsf{list})\ x' : \mathsf{list}$
pre true	pre true
post $(l = [] \Rightarrow l' = []) \wedge$	post $(x = [] \Rightarrow x' = [0]) \wedge$
$\quad (l \neq [] \Rightarrow$	$\quad \exists i : \mathsf{item}\ \exists l'' : \mathsf{list}\ \exists n : \mathsf{nat} \cdot x = [i]^\frown j \wedge$
$\quad\quad l' = (\mathsf{tl}\ l)^\frown[\mathsf{hd}\ l])$	$\quad\quad n = \mathit{length\_of\_list}(l'') \Rightarrow x' = [s(\mathsf{hd}(n))]$

Obviously, $\mathit{rotate}([\,]) = [\,]$, whereas $\mathit{length\_of\_list}([\,]) = [0]$. When we perform unfolding of the inductive definition of list and simplification of the first

conjunct, we immediately obtain [] = [0] which yields FALSE. Hence, the entire proof task is not valid.

7.5.2 Generation of Counter-Examples

In general, a proof task is not valid, if we find a *counterexample* for it. This means that we have a substitution σ for all variables in our formula $\mathcal{F}$ such that $\sigma\mathcal{F}$ (the formula with all variables instantiated) evaluates to FALSE. Model generators for FOL like Finder [Slaney, 1994] or MACE [McCune, 1994b] try to find such counterexamples by systematically checking all possible interpretations. This obviously terminates only if all involved domains are finite, as for example in finite group theory or hardware verification problems where such systems are applied with great success (e.g., [Slaney *et al.*, 1995; Fujita *et al.*, 1993]). On the other hand, the highly efficient implementation of most model generators (usually based upon BDD-based Davis-Putnam decision procedures) would make them ideal modules for detecting non-theorems.

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

For experiments in our case studies we used the system MACE [McCune, 1994b], simply because it can handle multi-sorted problems and accepts OTTER input syntax. The proof tasks are thus first converted into sorted clausal normal form. Afterwards OTTER [McCune, 1994a] (in a specific "anldp"-mode) is used to flatten all clauses in such a way that ANLDP[10] can process them. ANLDP itself, given the size and specification of the abstracted or approximated finite domains generates propositional formulas out of these clauses and tests their propositional validity, using a highly efficient implementation of a propositional satisfiability checker (a Davis-Putnam procedure). Finally ANLDP reports, whether a model could be found or not, or if the logical structure of the formula itself is unsatisfiable.

Mapping by Abstraction. One approach to establish this mapping uses techniques from abstract interpretation [Cousot and Cousot, 1977] where the infinite domain is partitioned into a small finite number of sets which are called abstract domains. For each function f an abstraction $\bar{f}$ is constructed such that f and $\bar{f}$ commute with the abstraction function α between original and abstract domains, i.e., $\alpha(f(x)) = \bar{f}(\alpha(x))$. E.g., we may partition the domain of integers into three abstract domains $zero = \{0\}$, $pos = \{x \mid x > 0\}$ and $neg = \{x \mid x < 0\}$. For example, for the multiplication $\times$, we obtain an abstraction, reflecting the well-known "sign rule":

$$neg \bar{\times} pos = pos \bar{\times} neg = neg$$

[10] The combination of OTTER and ANLDP (Argonne National Labs Decision Procedure) comprises the model generator MACE.

Abstract model checking [Jackson, 1994] then represents the abstract domains by single model elements and tries to find an abstract counter-model, using an axiomatization of the abstract functions and predicates with a standard FOL model generator. There is, however, a problem. While abstract interpretation may require extended truth values, standard FOL model generators need the exact concrete domain of TRUE and FALSE. Thus a consistent abstraction may become impossible. E.g., the abstraction of the ordering on the numbers, *less(zero, pos)* is valid but we cannot assign a single truth value to *less(pos, pos)* because two arbitrary positive numbers may be ordered either way. So, while there are some predicates which allow exact abstractions, we have to approximate others. This limits the practical usability of that approach considerably.

Mapping by Approximation. The second approach to map an infinite domain onto a finite one is done by approximation. From the infinite domain, a number of values is selected which seem to be "crucial" for the formula's behavior. E.g., for lists, one usually picks the empty list [] and small lists with one or two elements (e.g., $[a], [a, b]$). Then, we search for a model or counterexample. This approach mimics manual checking for satisfiability: if one has to check whether a formula is valid or not, one first makes checks with the empty list and one or two small lists. If this does not succeed, the formula is certainly not valid. Otherwise, additional checks have to be applied. This approach, however, is neither sound nor complete. There are invalid formulas for which a model can be found in a finitely approximated domain, and vice versa[11].

Example 7.5.2. Obviously, the formula

$$\forall X : \mathsf{list}\ \exists I : \mathsf{item} \cdot X = [i]^\frown X$$

is unsatisfiable, as can be easily seen using a simple argumentation about the lengths of lists: appending a non-empty list to X increases the length of the list; lists of different length are never equal. However, in an approximated finite domain, this formula has a model. As an example, consider the approximated domain of lists $\mathcal{D}'_{\mathsf{list}} = \{[], [a], [a, a]\}$ for some item a and the function-table for the append operator as shown in Table 7.2.

Obviously, $X \backslash [a, a]$ and $I \backslash a$ is an interpretation under $\mathcal{D}'_{\mathsf{list}}$ which yields TRUE. Thus, a model exists. The reason for this behavior can be seen easily when looking at the function table above. The entries marked by † do not reflect the behavior of the original append operator (which, e.g., would require a domain element $[a, a, a]$), but has been *bent* to fit into the approximated finite domain $\mathcal{D}'_{\mathsf{list}}$.

[11] In some cases (e.g., lattice-ordered groups), infinite domains with certain algebraic features can be mapped onto finite domains (e.g., a 3-element lattice) without losing expressiveness (cf. [Dahn *et al.*, 1994]). Such a transformation, however, does not apply for domains like lists.

$\wedge$	$[]$	$[a]$	$[a,a]$
$[]$	$[]$	$[a]$	$[a,a]$
$[a]$	$[a]$	$[a,a]$	$[a,a]^\dagger$
$[a,a]$	$[a,a]$	$[a,a]^\dagger$	$[a,a]^\dagger$

Table 7.2. An approximation of the append function $\wedge$ for the domain of list. The † indicates where "bends" occur.

Example 7.5.3. The formula

$$\exists X, Y, Z : \mathsf{list} \cdot X \neq Y \wedge Y \neq Z \wedge Z \neq X$$

obviously is valid, but only in domains with at least three distinctive elements. E.g., for distinctive elements $l_1, l_2, l_3 : \mathsf{list}$, $X \backslash l_1, Y \backslash l_2, Z \backslash l_3$ is a model.

Evaluation. We have evaluated counterexample generation by approximation for detection of non-theorems with a large database. The data (more than 14,000 proof tasks) are from our case study in Section 6.3. Due to the high number of variables in these proof tasks, only very small domain approximations could be used in practice[12]. They only have up to three elements per sort (list, item).

Example 7.5.4. One of the approximations maps the domain of item onto one element, called a, and list onto a domain with three elements:

$$\mathcal{D}'_{\mathsf{list}} = \{[], [a], [a,a]\}$$

For the function symbols, occurring in the proof tasks, a fixed function table has been defined as shown in Table 7.3.

$\wedge$	$[]$	$[a]$	$[a,a]$
$[]$	$[]$	$[a]$	$[a,a]$
$[a]$	$[a]$	$[a,a]$	$[a,a]^\dagger$
$[a,a]$	$[a,a]$	$[a,a]^\dagger$	$[a,a]^\dagger$

X	tl(X)	hd(X)
$[]$	$[]^\ddagger$	$a^\ddagger$
$[a]$	$[]$	a
$[a,a]$	$[a]$	a

Table 7.3. Approximation of domains for list and item. † indicates where "bends" occur. ‡ marks arbitrary definitions to overcome the partially defined functions head (hd) and tail(tl).

With a run-time limit of 20s (on a Sun ultra-sparc) we have been able to detect around 50% of the non-theorems in the data set. For detailed results on several approximations see Table 6.16 and Table 6.17 (page 134). Although this figure is not too bad, for too many valid formulas, a (fake) counterexample has been found by MACE (around 20%). Therefore, the results of this approach are not very reliable.

[12] Although ANLDP is rather flexible with respect to the number of variables (although recompilation is required), more than about 20–22 variables lead to unmanageable large search spaces.

An Extension. An extension of this simple approximation approach addresses several of the weaknesses mentioned above. Of course, we cannot obtain completeness, but we can make sure that models which are found are really models and that the rate of detected non-theorems can be increased. This extension consists of two separate issues:

- We run the extended model generator (as described below) in parallel with a theorem prover. Both systems are given reasonable run-time limits. After termination (or time-out) we decide upon the truth-value of the formula according to the following scheme: if the ATP terminates (with TRUE or FALSE), take this result. Due to the ATP's completeness and soundness, this result is reliable. Only if the ATP does not terminate, we take the extended model-generator's answer into account. If it finds a counterexample, the formula cannot valid. If the model generator comes up with the result "no model found", we cannot be sure about the formula's truth value. However, in many cases, it is probable that the formula indeed is valid, but the ATP could not find a proof for it. If both systems do not terminate within the given run-time limit, nothing can be said about the result.
- The model-generator, working with domain abstractions is extended with respect to two directions: finding more counterexamples and checking the models in the original domain. The extended model generator works iteratively according to the flow-chart in Figure 7.6. We start with an initial small approximation and search for counterexamples using MACE. If MACE detects that the formula is propositionally unsatisfiable, we can stop with that result. If MACE finds a model M with respect to our approximation, we check, whether M is a real counterexample for the original domain or if it was caused by "bending" the operators. In the latter case, we reject the model and proceed with our search. If MACE reports that no model could be found or all models have been rejected, we increase the size of the approximation and start again. Causes for rejections (i.e., places of the bends) give hints on how to improve the approximation.

Example 7.5.5. Let us again consider the formula from Example 7.5.2 on page 164 and the approximation given there as our initial approximation. MACE returns the model $I\backslash a$ and $X\backslash[a,a]$. When checking this model with respect to the function-table of the append operator, we detect that this model uses a function value which has been set arbitrarily, namely $[a,a] = [a]\hat{}[a,a]$ (a "bent" value, marked by † in the above example). Thus, we reject the model. Since no more models can be found with this abstraction, the approximation is enlarged by increasing the number of distinctive lists. With an additional domain element for lists, namely $[a,a,a]$, this specific bend does not occur. When restarting the search for counterexamples with this approximation, however, we get the same situation — as we expect because

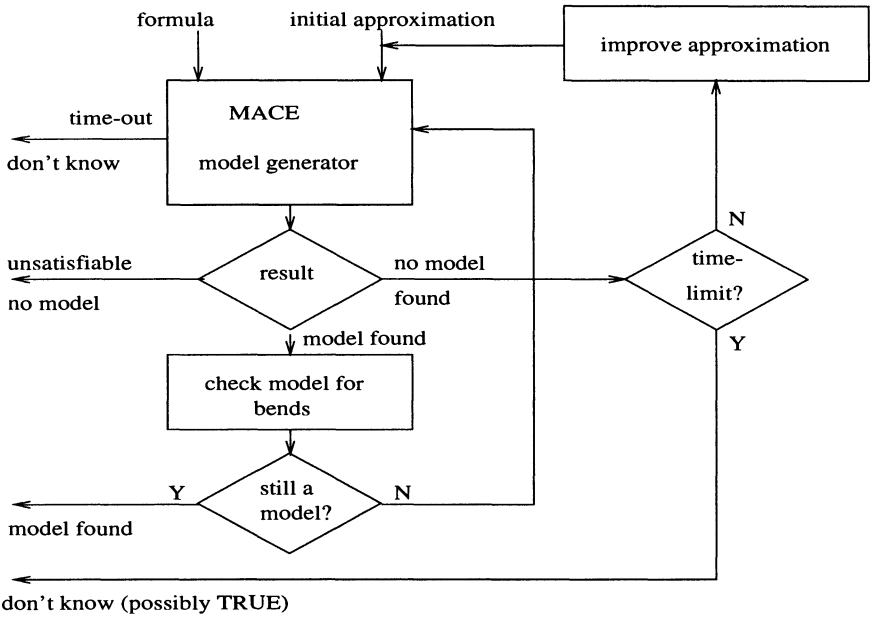

Fig. 7.6. Extended generation and checking of counterexamples.

this formula has no model. Thus, our system will not terminate (it is not complete), but at least it will not return a fake counterexample.

For the formula $\forall X, Y : \mathsf{list} \cdot X\hat{}[\,] = X\hat{}Y$ and the above approximation, MACE finds a model $X \backslash [a]$ and $Y \backslash [a]$. Since all required values for $\hat{}$ do not touch bent values, this model is a real counterexample, as can be seen easily: $[a]\hat{}[\,] = [a] \neq [a, a] = [a]\hat{}[a]$.

This extension guarantees that counterexamples are really counterexamples. Since this extended approach has not yet been fully implemented, we can give no experimental data on how the percentage of detected non-theorems evolves, although the figures in Table 6.16 indicate a rise in detected invalid proof tasks with growing approximations. However, problems of control, handling of large search spaces (induced by a large number of variables in the proof tasks) and heuristics on how to increment the approximation will have to be addressed before this approach can be evaluated and used in practice.

7.5.3 A Generative Approach

In many applications pure detection of non-theoremhood is not sufficient. Feedback is expected rather on what went wrong or which axioms might be missing. In the course of *debugging* a non-valid proof task, one is often interested in "What can be shown with the current axioms, assumptions and

hypotheses?" The structure of the resulting theorems (terms) can give helpful feedback on where the errors might be.

Example 7.5.6. Let us consider one example from our case study on authentication protocols (Section 6.1). The authentication protocol is the Kerberos protocol (for a specification see Example 6.1.3). Let us furthermore assume we had forgotten the assumption $pA \models \#T_a$ (principal A believes the freshness of its own time-stamp T_a)[13]. When we want to prove

$$pA \models pB \models pA \overset{K}{\leftrightarrow} pB$$

PIL/SETHEO, i.e., SETHEO does not terminate, indicating that the proof task might be not valid. But is there an error in the specification or are assumptions missing? Now, we ask PIL/SETHEO which kinds of BAN-formulas pA believes (PIL/SETHEO returns 124 formulas). A second step is to ask what pA believes to be fresh (freshness is an important issue in protocol analysis with BAN logic). PIL/SETHEO returns a list of 33 atoms (which are variants of the 8 formulas shown in Table 7.4). It is now quite obvious that there are no terms which contain any reference to freshness of time-stamp T_a. This is a clear indication that something is wrong with time-stamp T_a. This immediately leads to the missing assumption $pA \models \#T_a$. Adding this assumption to the formula then yields the desired proof.

$pA \models \#T_s$	$pA \models \{pA \overset{K_{ab}}{\leftrightarrow} pS, \#T_s\}$
$pA \models \{pS \models\!\!> pA \overset{K}{\leftrightarrow} pB, \#T_s\}$	$pA \models \{pA \overset{K_{ab}}{\leftrightarrow} pB, \#T_s\}$
$pA \models \{\#T_s, pB \mid\!\sim T_a\}$	$pA \models \{\#T_s, pB \mid\!\sim pA \overset{K_{ab}}{\leftrightarrow} pB\}$
$pA \models \{\#T_s, pB \mid\!\sim \{\}\}$	$pA \models \{\#T_s, pB \mid\!\sim T_a\}$

Table 7.4. Generated BAN-formulas about pA's beliefs concerning freshness

As demonstrated in the above example, such theorem generation can yield valuable feedback. The generation and filtering has been accomplished using a technique based on the DELTA preprocessor. This preprocessor which will be described in more detail in Section 7.6.1 performs bottom-up reasoning and produces unit-clauses in a forward reasoning manner. This means, starting with the axioms, DELTA performs forward chaining and generates unit-clauses iteratively. Its implementation on top of SETHEO allows for simple, yet efficient control of the generation process.

[13] In the original analysis of the protocol [Burrows *et al.*, 1989], this assumption accidently has been skipped. PIL/SETHEO (with the method described here) and [Craigen and Saaltink, 1996] found this omission independently.

Example 7.5.7. The filtering of the interesting beliefs in our example above has been accomplished by using SETHEO's pattern matching facility. Specifically, the following query was generated (here, we use the machine-oriented representation of the BAN-formulas. The predicate `filter(X,Y,Z)` selects all items from list `X` which match `Y` and assembles them into a result-list `Z`) which then can be printed with the predicate `print_result`:

```
?- delta(holds(B)),
     % use the delta iteration to obtain all beliefs
   filter(B,bel(pA,fresh(_)),Z),
     % only formulas about freshness are of interest
   print_result(Z).
     % output the corresponding formulas.
```

A query for PIL/SETHEO to generate all beliefs about T_a:

```
getbeliefs: pA believes any(nonce T_a)
```

is translated into a SETHEO formula, using a PROLOG-style membership predicate to select all generated BAN-formulas which contain the term T_a.

Although this method (which has been integrated into a prototypical version of PIL/SETHEO) proved to be very valuable for developing protocol specifications, it only can have restricted applicability in the area of software engineering, because

- this method can only be used in an interactive environment. It cannot detect any non-theorems by itself. Rather, the DELTA iterator generates theorems, derivable from the axioms. These must be inspected by the user who has to draw own conclusions. Then, the user has to modify the proof task and start the automated prover again.
- the generated unit clauses are represented on the machine level. If preparation of the proof tasks involved translation of logic and representation, this machine level can be of little value for the user. Therefore, this generated information must be postprocessed and converted into a format which is on the problem-oriented level and readable for the human user. This issue will be discussed in Section 7.7.4.
- the number of generated terms can get extremely large, even for small proof tasks. Here, elaborate means for filtering important information and its compact representation is of great importance. Otherwise the user will be drowned by too many data.
- this method is by no means complete. In order to keep the number of generated unit clauses within a reasonable range, usually quite tight bounds (e.g., on the size of terms) have to be applied during iterative generation. Thus, not all clauses are generated which in certain cases can mean a severe loss of information. Nevertheless, the flexibility provided by this system gives a powerful method for interactive "debugging" of formulas which are suspected to be not valid.

7.6 Control

After the proof task has been translated and preprocessed (according to Figure 7.1), it is now time to start the prover itself. In this section, we will focus on how to *control* the prover appropriately, such that it conforms to the requirements set up in Section 5.9, namely

- *practical usability*,
- *smooth behavior* w.r.t. similar proof tasks, and
- *short answer times* (i.e., solving as many proof tasks as early as possible – according to the required application profile).

A current-technology high performance automated theorem prover is nothing more than an extremely efficiently implemented search algorithm with *lots* of knobs to tune the system. Usually a prover has between 20 and 50 options which can be selected and combined. In general, it cannot be known in advance which parameter settings (or even which proof procedure) is good for the given problem. ATP designers and specialists usually claim to have some "feeling" which explains that in many cases, automated provers are particularly powerful when operated by their designers. For practical applications, however, techniques must be applied to control the theorem prover in a robust way. In this book, we will focus on two important topics: *combination of search paradigms* to overcome disadvantages of classes of proof procedures, and *parallel execution* to enhance the performance and to reduce answer times.

7.6.1 Combination of Search Paradigms

Introduction and Motivation. Top-down theorem provers with depth-first search, like SETHEO (Section 3.3), Protein [Baumgartner and Furbach, 1994b], PTTP [Stickel, 1988], or METEOR [Astrachan and Loveland, 1991] have the general disadvantage that during the search the same goals have to be proven over and over again. This causes a large amount of redundancy which cannot be avoided even by using state-of-the-art refinements of the calculus and proof procedure. Resolution-based bottom-up theorem provers (e.g., OTTER [McCune, 1994a; Kalman, 2001]), on the other hand, avoid this problem by performing backward and forward subsumption and by using elaborate storage and indexing techniques [Graf, 1996]. Those provers, however, often lack the goal-orientedness of top down provers[14]. Goalorientedness (i.e., the prover starts with the theorem to be proven) is very helpful in applications, where it is known which parts of the formula are the theorem, and which parts comprise the hypotheses and axioms. Then, during the search, valuable information from the original theorem (e.g., predicate

[14] To some extent the "set of support" strategy in resolution based theorem provers (Section 3.2.3) introduces goal-orientedness.

symbols, constants and function symbols occurring in the theorem) can be used to prune the search space.

Combination of Top-down and Bottom-up Search. In order to combine the advantages of top-down and bottom-up theorem proving, the author has developed the preprocessor DELTA. DELTA processes one part of the search space (the "bottom" part) in a preprocessing phase, using bottom-up techniques. These techniques avoid the repetitive reproving of identical goals, thus taking out much of the redundancy. Then, in the final step, usual top-down search is performed with an augmented formula. This "cutting-up" of the search space and the resulting system architecture is shown in Figure 7.7.

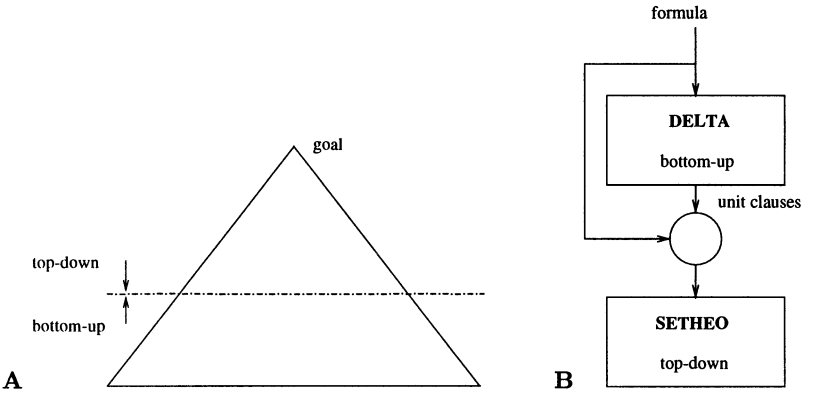

Fig. 7.7. Search space as spanned by a top-down search (**A**). The lower part is preprocessed by DELTA, the upper part by a subsequent top-down search. **B** depicts the resulting system architecture.

In the bottom-up preprocessing phase, we generate *unit-clauses* (clauses consisting of one atom or negated atom) which are added to the original formula. Then, this formula is processed by a top-down theorem prover in the usual way. During this top-down search, the additional unit clauses are used as generalized unit lemmata. Then, in many cases a remarkable gain in efficiency can be obtained[15].

Adding unit clauses to a formula does not affect completeness of the top-down proof procedure. Therefore, many ways of controlling the generation of unit clauses during the preprocessing phase are provided; e.g., the generation

[15] In case of a Horn-formula, a proof can always be found in the final top-down search with a lower bound than needed for a pure top-down proof, assumed that all bottom-up unit clauses have been generated. Since there is always the possibility that a leaf node in the tableau has to be closed by a reduction step almost up to the root, we cannot say as much in the Non-Horn case. Experiments, however, revealed that even in that case DELTA obtains good results (see below).

of new clauses can be restricted to specific predicate symbols, and unit clauses with an excessive size or complexity of terms can be filtered out.

Unit clauses generated by DELTA can even be helpful when used on every level of the top-down search. Restricting their use to the final level of the top-down search (on the "border-line" as suggested in Figure 7.7 A) is even counter-productive[16].

The Delta Iterator. Although this scheme is universal and any bottom-up and top-down prover could be used, the author used SETHEO and its logic programming facilities for the bottom-up and subsequent top-down phase. In order to obtain high efficiency for the bottom-up phase, the tool DELTA generates unit clauses level by level. This technique is similar to delta iteration as it is used in the field of database research, hence the name.

Starting with the original formula, we let SETHEO generate all new unit clauses which can be obtained by one UR-resolution step out of the current formula. This is accomplished by adding to the formula "most general queries" $\neg p(X_1, \ldots, X_n)$ for each predicate symbol p and variables $X_1, \ldots, X_n$. In case of a non-Horn formula, we also add $p(X_1, \ldots, X_n)$. For these queries, SETHEO searches for all solutions within a low bound δ_{bu} (here, usually, a depth-bound (A-literal depth) of 2 is used, but it can be varied freely)[17]. For each obtained substitution σ, we generate a unit clause "$p(\sigma X, \ldots, \sigma X_n)$" (or "$\neg p(\sigma X, \ldots, \sigma X_n)$", respectively). With the help of SETHEO's built-in unit-lemma mechanism (see Section 3.3.2), efficient forward- and backward subsumption is performed, thus reducing the number of generated unit clauses. Additionally, SETHEO's anti-lemma constraints (Section 3.3) substantially reduce the amount of redundant work. Filtering out valuable unit clauses is accomplished by using SETHEO's logic programming features. In the basic version of DELTA which has been implemented as a simple preprocessor using the UNIX tools shell and AWK [Aho *et al.*, 1988], terms can be discarded if their depth or size exceeds a given limit, or if the total number of generated unit clauses is getting too large. More elaborate filters have been developed with AI-SETHEO (see below).

Experimental Results. Table 7.5 shows the results of several experiments with well-known benchmark examples. Examples consisting of Horn clauses only are marked by an "H" in the second column. For this experiment, we have used a simple prototype version of DELTA [Schumann, 1994a]. As a resource limit for the preprocessing and the final top-down search, the depth of the proof tree (A-literal depth) has been used. DELTA has always been started with default parameters, except in cases where the number of newly

[16] The number of inner tableau-nodes is exponentially smaller than that of the leaves, and experiments revealed that in many cases, the additional lemmata yield proofs with less inference steps.

[17] In the case of non-Horn formulas, we also allow Model Elimination reduction steps and factorization steps ("folding-up", Section 3.3) to occur.

generated clauses grew too rapidly. In that case, the maximal term complexity has been restricted to 2 († in the table).

As a comparison, run-times for SETHEO without bottom-up preprocessing are shown (column "SETHEO"). As in the final top-down search, SETHEO has been started with its default parameters (iterative deepening with A-literal depth, and generation and usage of constraints). All run-times shown in this table are in seconds[18].

Example	H ?	iter. level	gener. units	run-time[s] DELTA	top-down	total	SETHEO
wos1		1	36	0.87	3.59	4.46	**1.53**
wos4		1	93	1.79	23.35	25.14	**22.93**
wos15	H	2†	256	15.58	61.82	**77.40**	807.61
wos17	H	2†	328	8.72	9.01	**17.73**	11081.93
wos20		1†	507	3.05	25.79	**28.84**	—
wos21	H	2†	144	3.94	20.03	**34.17**	—
wos31		4†	114	17.2	3.65	**20.85**	—
wos33		4	28	12.97	5.97	**18.94**	—
sam	H	3	66	12.97	5.97	**18.94**	—
LS36	H	1	148	1.35	45.28	**46.63**	87.25
LS37a	H	3†	257	21.67	7.52	**29.19**	—
Bled-1		2	67	1.96	4.40	**6.63**	6.67
Bled-2		2	68	1.90	6.87	**8.77**	114.33

Table 7.5. Experiments with DELTA, compared to pure top-down search

Related Work, Extensions and Discussion. A combination of both approaches would ideally have all the advantages of top-down and bottom-up provers while leaving out their disadvantages. Since such a combined system will largely increase the power of automated theorem proving, several approaches in this direction have been attempted. In [Bibel *et al.*, 1987], an integration of bottom-up features into the Connection Method [Bibel, 1987], a top-down proof-procedure, has been proposed. Techniques, adapted from relational databases are used to obtain and represent all solutions of subgoals simultaneously [Rath, 1992]. An approach to use the advantages of bottom-up provers for performing a top-down search is "Upside-Down Meta-Interpretation" [Stickel, 1991]. The input formula is transformed in such a way that a subsequent processing with a bottom-up prover exactly simulates a top-down derivation. Here, subsumption and indexing capabilities of the bottom-up prover mainly contribute to reduce redundancies, inherent in top-down theorem proving.

Recent extensions to the scheme of DELTA have been proposed by J. Dräger and implemented into AI-SETHEO (See CASC-15 [Suttner and Sutcliffe,

[18] They have been obtained on a Sun sparc II and include full compilation and preprocessing times for SETHEO V2.6. Time-outs have been set to 20,000s.

1999]). Here, the information a unit clause carries is taken into account during selection of valuable unit clauses. A version of DELTA is also part of a parallel cooperative theorem prover CPTHEO [Fuchs and Wolf, 1998].

7.6.2 Parallel Execution

Since the development of high-performance parallel computers, a variety of attempts have been made to increase the power of automated theorem provers by exploitation of parallelism. The approaches are based on many different calculi and proof procedures and explore a large variety of different parallel models. For extensive surveys see [Kurfeß, 1990; Suttner and Schumann, 1993; Schumann *et al.*, 1998]. In this book we investigate models of parallel execution which are suited to fulfill the requirements of practical usability: high performance, short answer times, and stability.

Classification of Models. Parallel execution models for automated theorem proving can be classified according to [Suttner and Schumann, 1993] along two orthogonal issues: the primary issue is about how the search space is explored by the parallel workers relates to each other. *Partitioning* relies on partitioning the exploration of the search space between parallel workers. In contrast, *competition* relies on different approaches to solve the same problem. The second issue is the degree of cooperativeness between the parallel workers. The distinguishing feature here is that information gathered during the processing of a computational task in the parallel system may or may not be shared with other tasks with the intention of saving work.

Table 7.6 shows the resulting classification matrix together with a number of prominent systems (from [Suttner and Schumann, 1993] where these systems are also surveyed).

		uncooperative	cooperative
Partitioning	completeness-based	PARTHEO, OR-SPTHEO Parthenon METEOR	CPTHEO MGTP/N
	correctness-based	AND-SPTHEO MGTP/G	- -
Competition	different calculi	-	HPDS
	one calculus	RCTHEO, SiCoTHEO, p-SETHEO	TWC

Table 7.6. A matrix representation of the classification taxonomy

Partitioning Parallelization. For *completeness-based partitioning* independent parts of the search space are distributed among parallel workers (e.g., classical OR-parallelism). A solution found by an individual worker usually constitutes a solution to the overall problem. Failure to process individual tasks generally causes incompleteness of the proof search. For *correctness-based partitioning*, the tasks (representing parts of the search space) given to workers are interdependent, and an overall solution is built up from the corresponding partial solutions (e.g., AND-parallelism). In general, dropping tasks which should be given to workers destroys the correctness of a deduction.

Both system architectures have been implemented on the basis of SETHEO and assessed extensively. PARTHEO [Schumann, 1991; Schumann and Letz, 1990] is an OR-parallel system (i.e., completeness-based partitioning), OR-SPTHEO [Suttner, 1995] performs static partitioning. However, due to the relatively complex system architecture (no fault-tolerance) and high communication requirements as well as limited scalability (evaluated by tests and simulation-based analysis [Schumann and Jobmann, 1994; Jobmann and Schumann, 1992]) partitioning models are not well suited for practical applications.

Competition Parallelization. Competition parallelization is based on the attempt to solve the same problem using several different approaches. In competition parallelism, the success of a single system is sufficient for a solution and allows the termination of the computation. Obviously, completeness of the overall computation is assured as long as at least one of the competing systems is deduction complete. Correctness is guaranteed if all competitors are correct. Competition can be performed using one calculus and proof procedure or using different theorem provers (heterogeneous competition). As an example for the latter case, ILF [Dahn and Wolf, 1996] uses parallel competition between SETHEO, OTTER [McCune, 1994a; Kalman, 2001], and DISCOUNT [Denzinger and Pitz, 1992]. In this book, we focus on homogeneous competition.

Furthermore, the competitors can additionally *cooperate* (e.g., by exchanging useful intermediate results). However, cooperation (e.g., CPTHEO [Fuchs and Wolf, 1998], DISCOUNT [Denzinger *et al.*, 1997; Denzinger, 1995; Denzinger and Dahn, 1998]) poses many difficult questions regarding parallel system design, e.g., the selection of the individual systems and the amount and type of information exchanged. The model of parallel cooperative competition seems to be suited for handling very hard problems with long run times (several minutes or more). Therefore, its usability for our intended applications is somewhat limited.

Parallel Competitive Systems: Computational Model. Given a sequential theorem proving algorithm[19] $\mathcal{A}(P_1, \ldots, P_n)$ where the P_i are parameters (or strategies) which may influence the behavior of the system and

[19] The definition of a parallel competitive system can easily be generalized to any search algorithm. An algorithm suitable for competition takes a problem as its

its search (for example, completeness bounds or pruning methods). Then, a *homogeneous competitive theorem prover* running on n processors is defined as follows: on each processor p ($1 \leq p \leq n$), a copy of the sequential algorithm $\mathcal{A}(P_1^p, \ldots, P_n^p)$ tries to prove the *entire* given formula. Some (or all) parameters P_i^p are set differently for each processor p. All processors start at the same time. As soon as one processor finds a solution, it is reported to the user and the other processors are stopped ("winner-takes-all strategy").

The efficiency of the resulting competitive system strongly depends on the influence of the parameter settings on the search behavior. The larger the difference, created by the values of P_i^p, the higher the probability that one processor finds a proof very quickly (if one exists, of course). Good scalability and efficiency can be obtained only if there are enough different values for a parameter, and if no good default estimation to set that parameter is known. Only then a large number of processors can be employed reasonably.

SiCoTHEO: Competition over Parameter Ranges. All variants of the parallel prover SiCoTHEO (*Si*mple *Co*mpetitive provers based on SETHEO) [Schumann, 1996c; Schumann *et al.*, 1998; Schumann, 1996a] perform parallel competition over parameter *ranges*. SiCoTHEO-CBC competes via a combination of completeness bounds, SiCoTHEO-DELTA via a combination of top-down and bottom-up processing using the DELTA iterator described in Section 7.6.1 above.

Implementation. SiCoTHEO runs on a network of UNIX workstations. The entire control of the proving processes is accomplished by the tool *pmake* [de Boor, 1989][20], a parallel version of *make*. It exploits parallelism by exporting independent jobs to other processors. *Pmake* stops if all jobs are finished or an error occurs. In our "the winner takes all strategy" the system has to stop as soon as *one* job is finished. Therefore, we adapted SETHEO such that it returns a UNIX "error" as soon as it found a proof. Thus pmake stops as soon as one proof has been found.

SiCoTHEO-CBC. The completeness bound, which is used for iterative deepening, determines the shape of the search space and therefore has a strong influence on the run-time the prover needs to find a proof. There exist many examples for which a proof cannot be found within reasonable time using iterative deepening over the depth of the proof tree (A-literal depth), whereas, iterative deepening over the number of inferences almost immediately reveals a proof, and vice versa[21]. In order to soften both extremes, SiCoTHEO explores a combination of the tableau depth bound d with the

input and tries to solve it. If a solution exists, the algorithm eventually must terminate with a message "solution found".

[20] This initial implementation of SiCoTHEO has been inspired by a prototypical implementation of RCTHEO [Ertel, 1992].

[21] This dramatic effect can be seen clearly in e.g. [Letz *et al.*, 1992], Table 3.

inference bound i_{max}. When iterating over depth d, the inference bound i_{max} is set according to $i_{max} = \alpha d^2 + \beta d$ with[22] $\alpha, \beta \in \mathbf{R}_0^+$.

SiCoTHEO-CBC explores *ranges* of α and β in parallel by assigning different values to each processor. For the experiments we selected $0.1 \leq \alpha \leq 1$ and $0 \leq \beta \leq 1$. Even slight variations in the values of α and β have strong effects on resulting run-time as shown in Figure 5.7 on page 94. The axes in this figure represent α and β, respectively.

SiCoTHEO-DELTA. This competitive system affects the search mode of the prover by combining top-down and bottom-up search as described in Section 7.6.1. The DELTA preprocessor has various parameters to control its operation. Here, we focus on two parameters: the number of iteration levels l, and the maximal allowable term depth t_d. l determines how many iterations the preprocessor executes. In order to avoid an excessive generation of unit clauses, the maximal term depth (t_d) of any term in a generated unit clause can be restricted. For our experiments, we performed competition over l and t_d in the range of $l \in \{1, 2, \ldots, 5\}$ and $t_d \in \{1, 2, \ldots, 5\}$. Furthermore, DELTA is configured in such a way that a maximum number of 100 unit clauses is generated to avoid excessively large formulae[23].

Experimental Results. All experiments with SiCoTHEO have been made with a selection of problems taken from the TPTP [Sutcliffe *et al.*, 1994]. All proof attempts (sequential and parallel) have been aborted after a maximal run-time of $T_{max} = 300s$ (on HP-9000/720 workstations); start-up and stop times can be up to several seconds and have not been considered in these experiments.

Table 7.7 (the first group of rows) shows the mean speed-up values for the experiments with SiCoTHEO-CBC with various numbers of processors (P). The arithmetic mean $\bar{s}_a$, the geometric mean $\bar{s}_g$, and the harmonic mean $\bar{s}_h$ are shown for reference[24]. These figures can be interpreted more accurately when looking at the graphical representation of the ratio between T_{seq} and $T_{||}$, as shown in Figure 7.8. Each dot represents a measurement with one formula. The dotted line corresponds to $s = 1$, the closely dotted line to $s = P$, where P is the number of processors. The area above the dotted line contains examples where the parallel system is *slower* than the sequential prover; dots below the closely dotted line represent experiments which yield a super-linear speed-up.

Figure 7.8 shows that even for few processors a large number of examples with super-linear speed-up exist. This encouraging fact is also reflected in

[22] This polynomial approximates the structure of a tableau described by $i_{max} = d^\eta$ for mean clause length η by a Taylor series with only the linear and quadratic terms.

[23] In order to overcome the negative effects of adding too many lemmas, we added a standard SETHEO to the competitors.

[24] Usually, $\bar{s}_a$ yields values that are too optimistic, while $\bar{s}_h$ is decreased too much by single small values. Therefore, often $\bar{s}_g$ is considered as the best mean value for speed-up measurements.

	mean	$P = 4$	$P = 9$	$P = 25$	$P = 50$
SiCoTHEO-CBC	$\bar{s}_a$	61.21	77.30	98.85	101.99
SiCoTHEO-CBC	$\bar{s}_g$	5.92	12.38	18.18	19.25
SiCoTHEO-CBC	$\bar{s}_h$	2.12	3.37	4.34	4.41

	mean	$P = 4$	$P = 9$	$P = 25$	
SiCoTHEO-DELTA	$\bar{s}_a$	18.39	63.84	76.50	
SiCoTHEO-DELTA	$\bar{s}_g$	5.15	12.07	16.97	
SiCoTHEO-DELTA	$\bar{s}_h$	1.43	2.92	3.71	

Table 7.7. SiCoTHEO: mean speed-up values for different numbers of processors P on 44 TPTP examples with $T_{seq} \geq 1s$

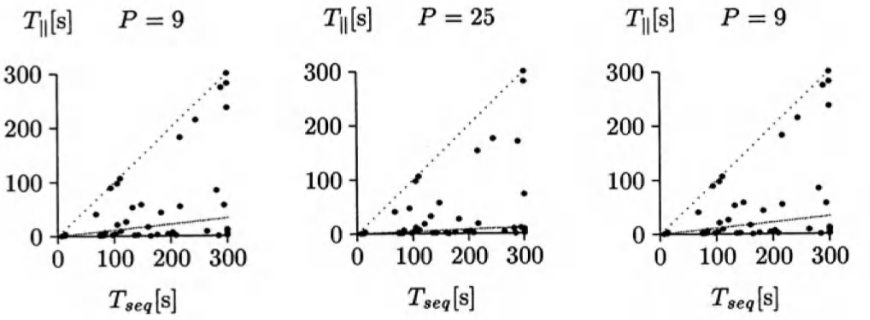

Fig. 7.8. SiCoTHEO-CBC: Comparison of sequential and parallel run-times with different numbers of processors

Table 7.7 which exhibits good average speed-up values for 4 and 9 processors. For our long-running examples and $P = 4$ or $P = 9$, $\bar{s}_g$ is even larger than the number of processors. This means that in most cases, a super-linear speed-up can be accomplished. Table 7.7 furthermore shows that with an increasing number of processors, the speed-up values also increase. However, for larger numbers of processors (25 or 50), the efficiency $\eta = s/P$ decreases. This means that SiCoTHEO-CBC obtains its peak efficiency with about 15 processors and thus is only moderately scalable.

The experiments with SiCoTHEO-DELTA have been carried out with the same set of examples and up to 25 processors. In general, the speed-up figures of SiCoTHEO-DELTA (Table 7.7, lower section) show a similar behavior as those for SiCoTHEO-CBC.

Assessment. Although extremely primitive in its implementation, the SiCoTHEO system shows impressive speed-up results. Looking at examples with a run-time of more than 1 second and a comparatively low number of processors, a super-linear speed-up can be obtained in most cases. However, the scalability of SiCoTHEO is relatively limited (approximately to less than 50 processors). For SiCoTHEO-CBC this is due to the fact that with larger num-

bers of processors the variations of α and β are too small to cause enough changes in run-time behavior. The coarse controlling parameters of DELTA limit the scalability of SiCoTHEO-DELTA. Speed-up and scalability could be increased substantially if one succeeded in producing a greater variety of preprocessed formulas. Furthermore, a combination of SiCoTHEO-CBC with SiCoTHEO-DELTA is of be of great interest.

For practical applications, SiCoTHEO can obtain good results on sets of proof tasks in the same domain. Furthermore, SiCoTHEO can ideally be used for fine-tuning specific parameter settings of the automated theorem prover SETHEO. However, the current implementation of SiCoTHEO using pmake has certain drawbacks in practical applications. For instance, when there is keyboard activity on a workstation, a process started by pmake is aborted automatically. Such a behavior protects the availability of workstations for the user but results in a highly non-deterministic and instable performance of SiCoTHEO.

p-SETHEO: Strategy Competition over Parameter Settings. The selection of more than one search strategy in a parallel competitive model in combination with techniques to partition the available resources (such as time and processors) with respect to the actual task is called *strategy parallelism.* Limitations of resources, such as time or processors, enforce efficient use of these resources by partitioning the resources adequately among the involved strategies. This leads to a highly complex optimization problem (cf. [Wolf and Letz, 1998; Wolf, 1998]).

Implementation. The implementation of p-SETHEO [Wolf and Letz, 1999; Letz *et al.*, 1998] is based on PVM [Geist *et al.*, 1994]. A (hard-coded) configuration contains information about usable hosts, the maximum load allowed per processor, and the strategy allocation for the competing strategies. All theorem proving strategies are variants of SETHEO obtained by modifying the parameter settings[25].

When started, p-SETHEO first select the strategies, computation times, and assigned processors according to the given problem and the available resources. Then, all independent preprocessing and search steps are performed in parallel. Synchronization is ensured by the existence of intermediate files.

Experimental Results. In order to illustrate the performance of p-SETHEO, two sets of experiments are presented (for details see [Schumann *et al.*, 1998; Wolf, 1999]):

- For the first experiment, a set of 420 problems in clausal normal form have been taken from the TPTP library[26]. For a given (non-optimized) set of

[25] p-SETHEO is implemented in in Perl, PVM, C, and shell tools (approx. 1000 lines of code).

[26] This selection has been used during the CASC-14 System Competition [Suttner and Sutcliffe, 1998] and comprises a selection of problems which already have been solved by at least one existing ATP, but not by all provers. Thus the proof task considered to be reasonably "hard".

20 strategies, run-times for each problem and each strategy was computed with a time limit T of 500 seconds per problem. The best single strategy could solve 211 of the problems. With any of the 20 given strategies, 266 of the 420 problems could be solved. After selecting a set of strategies and their corresponding execution times the resulting prover configuration was able to solve 246 problems within the given time limit using time-slicing of the strategies on a single processor.

- A further proof of the performance of the strategy parallel approach is the combination of four simple fixed strategies on four processors: iterative deepening over the depth of the tableau (A-literal depth), with and without the additional folding-up inference rule ($\mathcal{S}_1$, $\mathcal{S}_2$), and iterative deepening over the weighted depth bound, also with and without fold-up[27] ($\mathcal{S}_3$, $\mathcal{S}_4$). The experiments were performed on a test set of 230 tasks[28]. The results are shown in Table 7.8[29].
  This table shows that p-SETHEO on 4 processors could solve between 43% and 89% more problems than the sequential provers in only 22...32% of the time. Even if parallel work is used as a measurement instead of time, p-SETHEO solves significantly more problems with nearly the same costs.

configuration	solutions	% wrt. p-SETHEO	time (s)	% wrt. p-SETHEO
$\mathcal{S}_1$	151	66	106128	377
$\mathcal{S}_2$	159	69	91561	325
$\mathcal{S}_3$	122	53	128587	456
$\mathcal{S}_4$	161	70	87321	310
p-SETHEO	230	100	28168	100

Table 7.8. p-SETHEO: Comparison with single strategies (adapted from [Schumann *et al.*, 1998])

Assessment. p-SETHEO could obtain excellent efficiency as demonstrated on the CASC-15 system competition [Suttner and Sutcliffe, 1999]. Also, p-SETHEO has been used successfully within ILF. Its computational model is ideally suited for applications of automated theorem provers in the area of software engineering, because it can substantially reduce answer times and increase performance of reasoning. However, the current implementation of p-SETHEO still has ample space for improvements. The preselection of appropriate strategies is a hard problem which has been addressed in a feature-based way (cf. [Wolf and Letz, 1998]). However, this often is far from opti-

[27] For details on the parameters see [Letz *et al.*, 1994].

[28] These 230 tasks from TPTP can be solved by at least one of the four configurations within 1000 seconds but are solved by at most two configurations within less than 20 seconds.

[29] The provers tried to solve each problem within at most 1000 seconds. p-SETHEO added the times of the best competitor.

mal and, due to its implementation, this selection cannot be changed easily. Furthermore, the stability of parallel version not satisfactory, mainly due to weaknesses of the underlying PVM system: p-SETHEO exhibits relatively long start-up times, has no flexibility in processor assignment (i.e., the user or administrator has no means to explicitly control the placement of processes onto processors), and often poor synchronization when used with a shared file-system (files are not available remotely although the process generating them has finished already).

SiCoTHEO*: An Extension. Although SiCoTHEO and p-SETHEO are very powerful, there is still a number of important improvements with respect to practical usability. In the following, we propose a system architecture which addresses several weaknesses of SiCoTHEO and p-SETHEO. Both systems usually have severe problems with stability when used on a real-world network of workstations with varying loads. Furthermore, the strategy selection in p-SETHEO and resource allocation is hard-coded into the system. Therefore, its adaptation to new application domains and tuning is rather complicated and error prone. A third weakness which both parallel systems share is their limited possibility to control the system, e.g., to set time limits, to kill the processes, and to obtain intermediate results.

Basic Architecture. Our approach is based on an extended version of a parallel make utility *pmake* [de Boor, 1989]. The control language is extended in several ways to address the issues mentioned above. The basic system architecture of pmake, using a UNIX-daemon for controlling the processes and a shared file system for exchanging data, has been kept.

Setting up competitors in that control language is rather simple. As with the usual make-utility, the DAG of flow control is specified using source and destination labels. In order to achieve a result l_0, prerequisites $l_1, \ldots, l_n$ must be provided and action A (e.g., a preprocessing step) has to be performed. Actions which are independent from each other are executed in parallel. On the last stage of the competition DAG, the leaf nodes, competition between different ATPs or strategies is enforced, killing all processes as soon as one process could solve the problem, i.e., the according prover returns successfully. This is indicated by the keyword `competition`:

$$\begin{array}{ll} l_o & : l_1 \ldots l_n \\ & \quad A \\ \\ \texttt{competition} & : l_1 \ldots l_n \\ & \quad A' \end{array}$$

Distribution of the processes onto processors and simple ways of achieving a good load-balance is kept transparent to the user. Per default, all machines in a network are used on which SiCoTHEO*'s daemon is running and which have a reasonable system load. If more control is desired, processes can di-

rectly be assigned to processors or priorities can be set (see below) which influence the distribution of processes.

Example 7.6.1. Let us consider the following scenario: we want to start SETHEO with two different kinds of equality handling in parallel. Additionally, for each variant, we provide two different preprocessing modules (e.g., different ways of simplification), and the search for the proof should be done with two different set of parameters. Thus, we end up with 8 different processes. Since several preparation steps can be shared, we obtain a control flow as shown in Figure 7.9. First, the syntax is converted and the proof task is translated into clausal normal form. Then, the two modules for equality handling (E1 and E2) are performed in parallel. After the run of E1, an additional module for handling sorts (S1) is required. Finally, the two variants of preprocessing modules (P1, P2) and of the prover (SAM1, SAM2) are started in a competitive mode. We obtain a total of eight competitive processes, numbered (1)–(8) in the figure.

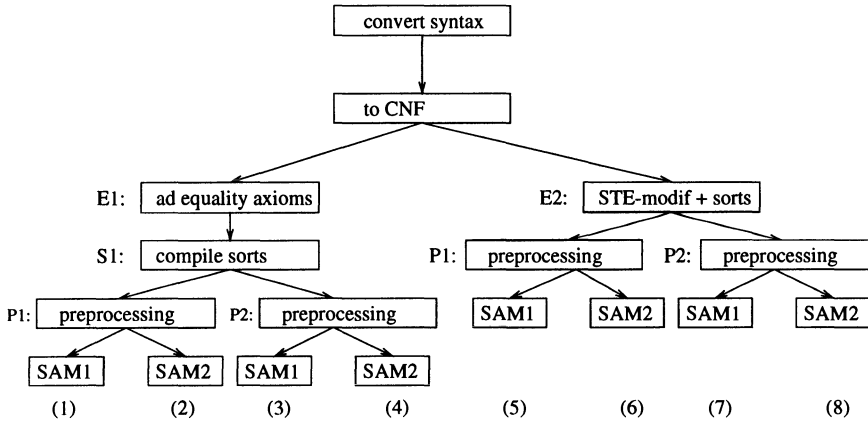

Fig. 7.9. Simple flow for competitive strategies

In SiCoTHEO*'s input language, this problem would be represented as shown in Figure 7.10. For simplified representation, labels of prerequisites can be specified using (UNIX-style) regular expressions (e.g., `P[12]` means `P1` or `P2`). In this example, all details of the actions to be carried out are hidden in variables which are expanded during run-time (e.g., `$(TO_CNF)` would call the transformation module for converting a formula into clausal form).

Selecting Strategies. For practical usability, the user should be relieved of specifying any details of the prover's behavior. Nevertheless, the user expects high performance of SiCoTHEO* for the designated application. Therefore,

```
competition: P[12]
        $(SAM1)

competition: P[12]
        $(SAM2)

P1:          S1 | E2
        $(preprocess1)

P2:          S1 | E2
        $(preprocess2)

S1:          E1
        $(Compile Sorts)

E1:          CNF
        $(equality1)

E2:          CNF
        $(equality2)

CNF:         conv_syntax
        $(TO_CNF)

conv_syntax: $(INFILE)
        $(CONVERT_SYNTAX)
```

Fig. 7.10. SiCoTHEO* control file for process tree shown in Figure 7.9

libraries of pre-defined actions (syntactically similar to the standard rules of the tool *make*) and compact definitions for parameterized strategies and parameter ranges are available.

These libraries also hide standard ways of heuristic selections of appropriate sets of strategies and parameter ranges, based on syntactical criteria (and semantic information) provided by the proof tasks. Thus SiCoTHEO can simulate the strategy selection of p-SETHEO. Our approach also allows to use other techniques of parameter selection, like neural networks [Schumann, 1996a] to be integrated easily.

If an application requires a more low-level control, modifications to the predefined selections can be made manually. Our control language contains modules which automatically analyze the input formula and set appropriate variables, e.g., `NRLIT` for number of literals or `MAXARITY_PRED` for the maximal arity of a predicate. With these variables, calculations can be performed which allow to select or modify the appropriate strategies.

Example 7.6.2. This example shows a simple case of calculating parameter settings from given characteristics of the proof task. The boolean variable `ISHORN` is true if the formula to be processed contains Horn clauses only. According to the formula's syntactic characteristics, the list of command options for SETHEO is adapted, in our case by adding additional command-line parameters.

```
competition:     S1
   if (NRLIT > 100 & ISHORN & MAXARITY_PRED < 3)
       SAMFLAGS=$(SAMFLAGS) " -foldup";
   else
       SAMFLAGS=$(SAMFLAGS) " -dynsgreord 4";
   fi
   $(SETHEO) $(SAMFLAGS) $(FILE)
```

Monitoring Processes. In a competitive model, there always exists the situation that a strategy (or parameter value) obviously does not lead to a positive result. Typical indications for this can be: preprocessing takes much more time than anticipated, or iterative deepening shows an irregular behavior (e.g., levels are processed too fast, or the search seems to be stuck at some level). If such a case can be detected early and the corresponding processes are killed, much resources can be saved and used to explore other strategies. Thus, SiCoTHEO* provides means to *monitor* the individual processes. In contrast to the approach of self-control where each process monitors its own progress and "commits suicide" if there is not enough progress, we adhere to the model of outside control. The processes are monitored constantly by the global control program which also decides if a process is to be killed[30]. Our extended language provides the keywords `monitor` and `kill` for these purposes.

For further control, time limits for individual processes or the overall system can be set and enforced. Priorities for individual processes concern the running priority of the process on a processor (i.e., the UNIX nice-level) and the assignment of processors.

Example 7.6.3. The following two examples show some possibilities for controlling and monitoring individual processes. In the first case, the prover is started with an increased priority. This priority value is only evaluated when the process is to be started and influences the distribution of the processes onto different processors according to their processing power and current load. In the second example, the prover is started in a monitored way. The output of the process is scanned for the string `d = 4` (here indicating that a depth-bound of 4 has been reached during iterative deepening). If a match is found and the consumed CPU-time of that process already exceeds 5 seconds, this process is killed. With such a construct, parameter settings which are likely not to contribute to a solution can be killed early.

```
competition:     S1
        with priority +15 $(SETHEO) $(SAMFLAGS)
competition:     S1
        $(SETHEO) $(SAMFLAGS2)
        monitor (/d = 4/ & $CPU > 5 ) {
               kill
               }
```

[30] In contrast to the cooperative prover DISCOUNT [Denzinger and Pitz, 1992] where specific modules, called referees check the progress made by the members of the team in regular intervals, here checks are made constantly, but on a lower syntactic level.

Assessment. SiCoTHEO* has been designed for high flexibility and stability. It addresses problems which occurred in the systems SiCoTHEO and p-SETHEO, like delays in shared file systems, explicit control of distribution, and control of the individual processes. Its current, still rather prototypical implementation is based upon pmake. With all features integrated which have been described above, SiCoTHEO* is a parallel execution vehicle for automated theorem proving which meets important requirements for practical applicability as set up in Section 5.2 and 5.9.

7.7 Postprocessing

Many applications require more feedback from the automated theorem prover than TRUE/FALSE/ERROR. In this section, we will investigate which kind of information can be returned by the prover and which steps must be performed in order to satisfy the application system's needs. An automated theorem prover usually keeps internal information which allows to reconstruct the proof on machine level, i.e., the sequence of inference steps of the prover's calculus which led to the proof. Depending on the prover's calculus and implementation, this can be a trivial task or can require run-times in the order of the time needed to find a proof (e.g., DISCOUNT [Denzinger and Pitz, 1992]). However, such a proof usually is not a proof in the syntax and logic of the original proof task, because preprocessing involved a number of translation steps (see Figure 7.1 on page 138). Thus we can distinguish 4 levels of representation for the prover's results which will be described below: proof-related information, machine-oriented proof, machine-oriented proof in source logic, and human-readable proof.

7.7.1 Proof-related Information

This kind of information comprises data which have been produced by the automated theorem prover, but which are not a full proof. Besides statistical information and protocols of the run, two issues are of particular importance: *clauses used* and *answer substitution.*

Clauses Used in Proof. Clauses which participate in the proof (i.e., they have been used in inference steps) are often very helpful to detect minimal sets of axioms, properties which are necessary to prove the conjecture, and so on. For most theorem provers, the list of used clauses can easily be obtained from the log-file or the proof the system returns.

Example 7.7.1. For SETHEO the set of clauses used can be extracted by a simple script, using SETHEO's proof output and the input file. Each node in the tableau which has been generated by an extension step (denoted by `ext__`), contains the clause and literal numbers of both complementary literals

as shown in the following example. The formula (from Example 3.2.4 on page 32) consists of two clauses $\neg p(a, Z) \vee \neg p(Z, a)$ and $p(X, Y) \vee p(Y, X)$. The closed tableau which has been generated by SETHEO is shown in Figure 7.11 below. `query__` denotes the label ϵ of the tableau's root node. The tree is encoded as a list of lists (for practical purposes, SETHEO uses PROLOG syntax). The clause numbers can be extracted easily. These numbers are then used to select the appropriate clauses from the input file (e.g., by using a simple AWK script).

```
[ [~query__,[ 0 , ext__(0.1,2.1) ] ,[
    [ query__ ] ,
     [~p(a,a),[ 1 , ext__(2.2,1.1) ] ,[
      [ p(a,a) ] ,
        [p(a,a), [ 2 , red__(1.2,1) ]]
    ] ],
      [~p(a,a),[ 3 , ext__(2.3,1.1) ] ,[
      [ p(a,a) ],
        [p(a,a), [ 4 , red__(1.2,3) ]]
    ] ]
  ] ] ].
```

Fig. 7.11. SETHEO's representation of a closed tableau

Answer Substitution. The term *answer substitution* originates from the area of logic programming and denotes substitutions of variables in the query of a logic program. For example, when the query $\exists X : p(X)$ is given to a logic program $p(a)$, the answer substitution is $X \backslash a$. In logic programs, answer substitutions usually represent calculated results of a logic program, as in the classical PROLOG example [Clocksin and Mellish, 1984]:

```
?-father_of(mary,X).
father_of(mary,john).
```

which results in `X\john`. In automated reasoning answer substitutions are being used in many situations, e.g., when reasoning in knowledge bases, during program synthesis, or for planning. There, in general, an answer substitution consists of the variable substitutions of all existentially quantified variables in the conjecture.

Example 7.7.2. The following example show typical usage of answer substitutions. The AMPHION system (see Section 4.7.1) is used to synthesize FORTRAN programs out of a given problem specification $\mathcal{P}$ and the input-output specification of the FORTRAN subroutines available in the library. Then, a proof task is essentially of the form:

$$\exists F : \mathsf{program} \cdot find(\mathcal{P}, F)$$

For example [Stickel *et al.*, 1994], $\mathcal{P}$ might be equivalent to

```
Tm     = abs(ephemeris-time-to-time(et)) ∧
Pearth = a-findp(earth,Tm) ∧
Pmars  = a-findp(mars,Tm) ∧
D      = two-points-to-distance(Pearth,Pmars)
```

with existentially quantified variables `∃D,Tm,Pearth,Pmars`. This formula describes the specification of the calculation of the distance `D` between earth and mars at a given time `et`. The answer for this rather simple example is the term

```
vdist(findp(earth-naif-id,et),findp(mars-naif-id,et))
```

which then can be translated into a sequence of three FORTRAN subroutine calls: first, calculate the positions of Earth and Mars (using `findp`), then obtain their distance with the subroutine `vdist`.

AMPHION uses SNARK [Stickel *et al.*, 1994], a resolution based theorem prover for the synthesis which has to perform extra computations in order to obtain the answer substitution.

When the formula only consists of Horn clauses, the answer substitution is always a single variable substitution σ. In the Non-Horn case, an answer substitution can be *disjunctive*, i.e., a disjunction of variable substitutions. When we have a formula $p(a) \vee p(b)$ and a conjecture $\exists X \cdot p(X)$ we get the answer substitution

$$X = a \vee X = b.$$

With the theorem prover SETHEO, answer substitutions can be easily generated and displayed in the Horn and Non-Horn case using SETHEO's logic programming features[31]. In the Horn case it is sufficient to display the substitution values for all existentially quantified variables in the conjecture.

Example 7.7.3. A conjecture $p(X,Y)$, written as `<- p(X,Y).` in SETHEO's input syntax, is augmented by procedural built-in predicates[32] to display the required information:

```
?- p(X,Y),$write("X="),$write(X),
          $write(" Y="),$write(Y),
          $write("\n").
```

[31] The prover Protein [Baumgartner and Furbach, 1994a] is also capable of generating disjunctive answer substitutions as described in [Furbach *et al.*, 1995].

[32] The `?-` instead of `<-` disables static reordering of literals in a clause and thus ensures that the values of the variables are printed *after* the proof could be found.

The built-in predicate `$write(t)` prints term `t`. As soon as a proof for $p(X,Y)$ could be found, the substitutions for X and Y are printed. All answer substitutions within given bounds can be printed if a `$fail` is added at the end of the clause. This forces SETHEO to backtrack, after a proof has been found.

In the Non-Horn case, additional information about the variables in the conjecture has to be kept. Whenever the conjecture clause is touched (with the model elimination start step or by an extension into that clause), the substitutions of the variables in that instance form a new disjunct of the answer substitution. This additional book-keeping is accomplished efficiently using SETHEO's backtrackable global variables (Section 3.3.2).

Example 7.7.4. For display of disjunctive answer substitutions, the conjecture `<- p(X)` must be transformed into:

```
?- $ANS := [], query, $write("X= "), $write($ANS),
     $write("\n").  /*(1)*/
query :- p(X), $ANS := [X|$ANS].
~p(X) :- $ANS := [X|$ANS].
```

The global variable `$ANS` which is first set to an empty list `[]` assembles all answers. Whenever the conjecture is used for an extension step, the substitution for variable `X` is pre-pended to the already stored answer substitutions. Only the conjecture has to be instrumented; the rest of the formula remains unchanged. When SETHEO is started and finds a proof, it prints the list of all disjuncts of variable substitution. For example, we might get: `X= [a|[b|[]]]` which corresponds to $X = a \vee X = b$.

If a more convenient output is needed, however, additional clauses must be added in order to convert the list into a correct form. Note, that additional built-ins are needed to bypass the bounds, and to keep the number of inferences in the proof correct (line 16, and lines 3,5,6 respectively in the code below). The following program produces the desired output `X=a ; X=b` (cf. [Schumann and Ibens, 1998]):

```
01 ?- $ANS := [],
02       query,
03       $getinf(I),
04      print_answersubst($ANS),
05      I1 is I -1,
06      $setinf(I1).
07
08 query :- p(X), $ANS := [X|$ANS].
09 ~p(X) :- $ANS := [X|$ANS].
10
11 p(a);p(b)<-.
12
13 print_answersubst([X]) :- $write("X="),$write(X),
14        $write("\n").
15 print_answersubst([X|Y]) :-
16        $setdepth(1),$setinf(1),
17        $write("X="), $write(X), $write(" ; "),
18        print_answersubst(Y).
```

7.7.2 Machine-oriented Proofs

This kind of proof is directly generated by the automated theorem prover. It is a proof in the logic of the ATP and in its calculus. For example, for SETHEO such a proof would be a closed model elimination tableau, for OTTER a sequence of resolution steps. The obvious advantage of such a proof as a result is that it contains exactly and all the information to formally show the truth value of the proof task. Furthermore, such a proof can be checked externally by a proof checking program, enhancing confidence in the theorem provers results.

However, such a proof is usually extremely hard to read even for an expert, because many inference steps and large terms make a clearly arranged representation impossible. Furthermore, translation steps performed on the proof tasks can lead to formula representations which do not have much in common with the original proof task (for example proof tasks of Section 6.1 and Section 6.3).

7.7.3 Machine-oriented Proofs on the Source Level

Translation of machine-oriented proofs on the ATP's level to the source level, i.e., the representation and logic of the original proof task can be extremely difficult. Transformation steps (e.g., translation of logic, construction of clausal normal form) can, albeit preserving the formula's provability, lead to situations where a translation back is impossible without further provisions. Furthermore, main characteristics of the proof can change considerably (e.g., exponential blow-up (or shrinking) of the proof length).

Example 7.7.5. The simple conversion of a first-order formula into clausal normal form (as described in Section 3.2.2) destroys the structure of a formula in such a way that a unique back-translation is not possible. The main reasons are that parts of the formula get distributed into many clauses when logical operators like $\rightarrow$ or $\leftrightarrow$ are eliminated and variables are being Skolemized or renamed.

For example, the formula $\exists X \cdot (p(X) \vee q(X))$ is first converted during an optimization step (minimization of quantifier ranges) into $\exists X_1 \cdot p(X_1) \vee \exists X_2 \cdot q(X_2)$. Skolemization finally produces $p(a_1) \vee q(a_2)$, loosing any connection with the original variable X.

Here, structure preserving transformation algorithms, like definitional normal forms [Eder, 1984] can help. Additionally introduced predicate symbols mark the positions of the sub-formulas in the entire formula. Furthermore, Skolem constants can be named in such a way that the original variable name is retained. For example, $\exists X \cdot p(X)$ can be skolemized into $p(a\_p\_X)$ with Skolem constant a_p_X. These approximations allow to extract the necessary information for back-translation. According to the author's knowledge, no implementation has been attempted yet due to the inherent difficulty of this back-transformation.

In this book, we will focus on the approach of representing machine oriented proofs on a high level as developed within the ILF project [Dahn and Wolf, 1996]. ILF uses a common calculus, the *block calculus* for internal representation of all proofs found by an automated theorem prover or constructed interactively. The block calculus [Dahn and Wolf, 1994] is a sequent-style calculus based on Gentzen's natural deduction. It has been extended to represent structural information, called *blocks*. This calculus which is described in detail in [Dahn and Wolf, 1994; Wolf, 1997] is the basis of the interactive system ILF. This calculus and its tools can be also be used with an automated theorem prover in a stand-alone way (ILF-SETHEO, see below). The block calculus has the following advantages:

- Due to its vicinity to natural deduction, a proof in the block calculus can be easily transformed into a natural language proof (see below). Such proofs are much more readable for humans than proofs in other calculi like resolution or model elimination.
- The block calculus allows to *structure* a proof, such that "uninteresting" parts can be hidden. Similar to a block-structured programming language, the structuring elements largely facilitate a clear representation.
- Proofs in machine-oriented calculi like resolution or model elimination can efficiently be transformed into proofs of the block calculus. For example, ILF provides transformation routines for many automated theorem provers such as Protein, OTTER, or SETHEO.
- Proofs (or parts of proofs) found by different theorem provers or constructed interactively within ILF can be combined into a single proof of the block calculus. This feature is especially important for application systems which combine interactive theorem proving with automated reasoning.

7.7.4 Human-readable Proofs

Many applications in software engineering require the generation of human-readable proofs. The application engineer must be able to check the proof, or at least to understand what is going on in the proof in order to determine the lines of reasoning. This means that the proof must be presented in the source logic, and it needs to be structured in a way to facilitate reading and understanding. Probably the most developed system in this area is ILF.

ILF-SETHEO. ILF-SETHEO [Wolf and Schumann, 1997] (a part of the ILF system) is a stand-alone tool which has been developed using certain functionalities of ILF on top of SETHEO. It is a true postprocessor to SETHEO and has been used for all the case studies in this book. ILF-SETHEO is also a major component of PIL/SETHEO (Section 6.1).

ILF-SETHEO's system architecture, as shown in Figure 7.12, is straightforward. Taking the formula (after preprocessing) and the proof found by SETHEO (a closed tableau), ILF-SETHEO constructs a proof in the block calculus. In a second stage, this proof is converted into a textual format and

type-set using LaTeX commands. Finally, a LaTeX document is generated. A configuration file, containing operator definitions and type-setting instructions allows for a convenient representation of the formulas. This script can use all features of PROLOG[33]. Therefore, elaborate conversions in formula and term-representation and many logic translations can be accomplished easily.

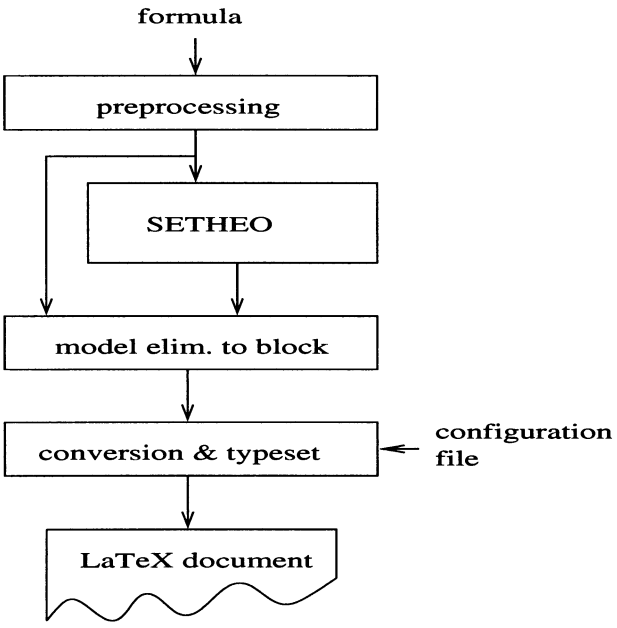

Fig. 7.12. System architecture of ILF-SETHEO

Example 7.7.6. The tool PIL/SETHEO uses ILF-SETHEO to generate readable proofs in the BAN or AUTLOG logic. As described in Section 6.1, the modal BAN (or AUTLOG) logic is translated into first-order logic by introducing a meta-predicate (`holds`) and subsequent representation of all BAN formulas as first-order terms. Thus, the configuration file which is shown in Table 7.9 is rather straightforward. For each symbol on the machine level, its representation, position (infix/prefix), and binding power is given. During type-setting, the symbols are replaced by the corresponding expressions. Here, all occurring symbols are built into LaTeX (e.g., `\leftarrow` produces[34] $\leftarrow$), or are defined by the user (e.g., `\bel` for the belief operator $\models$). Backslash characters have to be preceded by another "\" as an escape character.

[33] An online-manual of ILF can be obtained by sending an email with subject `help` to `ilf-serv@mathematik.hu-berlin.de`.

[34] A formula $\mathcal{A} \leftarrow \mathcal{B}$ is equivalent to $\mathcal{B} \rightarrow \mathcal{A}$ (PROLOG-style implication).

The conversion of the internal representation of a message $\{m_1, \ldots, m_n\}$ is done by a small PROLOG program. The translation of the internal form of a message $\mathtt{and}(m_1, \mathtt{and}(\ldots, \mathtt{and}(m_n, \mathtt{nil})\ldots))$ into the linear form $\{m_1, \ldots, m_n\}$ is accomplished with the PROLOG predicate `p1`. The output of a (simple) example, the first proof task of the Kerberos protocol is shown below (for another example see page 112).

```
TEXOP
  % formatting instructions for constants
struct 500   a__principal    :- "pA"
struct 500   b__principal    :- "pB"
struct 500   k_a_b__key   :- "K_{A,B}"
struct 500   t_a__nonce    :- "T_A"
  % formatting of functions and predicates
xfy   1050    '<-'        :- "\\leftarrow"
fy    800     holds       :- "\\ "
fy    800     fresh       :- "\\#"
xfy   800     bel         :- "\\bel\\ "
xfy   800     said        :- "\\said\\ "
xfy   800     sees        :- "\\sees\\ "
xfy   800     controls    :- "\\controls\\ "
  % formatting of: A,B communicate via Key C
struct 800    sk(A,B,C)  :- z(A),"\\stackrel{",
                z(C),"}\\leftrightarrow ",z(B)
  % typesetting of messages {M1,...,Mn}
struct _      and(U,nil) :- z(U)
struct _      and(X,and(Y,Z)) :- "\\{",
                z(p1(and(X,and(Y,Z))))
struct _      and(U,V)   :- "\\{",z(U),",",
                z(V),"\\}"
struct _      p1(and(X,nil)) :- z(X),"\\}"
struct _      p1(and(X,Y)) :- z(X),",",z(p1(Y))
```

Table 7.9. ILF-SETHEO-configuration file with type-setting instructions

A Proof from Ilf

Axiom 7.7.1 (***Assumption***$_2$) $A\models\ S\mapsto\ A \overset{K}{\leftrightarrow} B$.

Axiom 7.7.2 (***Message***$_2$)
$A \triangleleft \{\{T_S, A \overset{K_{A,B}}{\leftrightarrow} B, \{\{T_S, A \overset{K_{A,B}}{\leftrightarrow} B\}\}_{K_{B,S}}\}\}_{K_{A,S}}$.

Axiom 7.7.3 (***Assumption***$_2$) $A\models\ A \overset{K_{A,S}}{\leftrightarrow} S$.

Axiom 7.7.4 (***Message-Meaning***)
$P\models\ Q\mid\sim\ X_1 \leftarrow P\models\ P \overset{K}{\leftrightarrow} Q \wedge P \triangleleft \{X_1\}_K$.

Axiom 7.7.5 (***Assumption***$_2$) $A\models\ \#T_S$.

Axiom 7.7.6 (*Freshness*) $P\models\ \#X_1 \leftarrow P\models\ \#Y \wedge Y \in \{X_1\}$.

Axiom 7.7.7 (*Nonce-Verification*)
$P\models\ Q\models\ X_1 \leftarrow P\models\ \#X_1 \wedge P\models\ Q\mid\sim\ X_1$.

Axiom 7.7.8 (*Submessage*)
$P\models\ Q\models\ S \leftarrow S \sqsubseteq X_1 \wedge P\models\ Q\models\ X_1$.

Axiom 7.7.9 (*Jurisdiction*)
$P\models\ X_1 \leftarrow P\models\ Q\models\!\Rightarrow\ X_1 \wedge P\models\ Q\models\ X_1$.

Axiom 7.7.10 (*Conjecture*$_1$) $\leftarrow A\models\ A \overset{K_{A,B}}{\leftrightarrow} B$.

Theorem 7.7.1 *query.*

Proof (by SETHEO). We show directly that

query. (7.2)

Because of *Message*$_2$, *Assumption*$_2$, and by *Message-Meaning*

$$A\models\ S\mid\sim\ \{T_S, A \overset{K_{A,B}}{\leftrightarrow} B, \{\{T_S, A \overset{K_{A,B}}{\leftrightarrow} B\}\}_{K_{B,S}}\}.$$

Because of *Assumption*$_2$ and by *Freshness*

$$A\models\ \#\{T_S, A \overset{K_{A,B}}{\leftrightarrow} B, \{\{T_S, A \overset{K_{A,B}}{\leftrightarrow} B\}\}_{K_{B,S}}\}.$$

Therefore by *Nonce-Verification*

$$A\models\ S\models\ \{T_S, A \overset{K_{A,B}}{\leftrightarrow} B, \{\{T_S, A \overset{K_{A,B}}{\leftrightarrow} B\}\}_{K_{B,S}}\}.$$

Hence by *Submessage* $A\models\ S\models\ A \overset{K_{A,B}}{\leftrightarrow} B$. Hence by *Assumption*$_2$ and by *Jurisdiction* $\neg$*query*. Hence by *Conjecture*$_1$ *query*. Thus we have completed the proof of (7.2).

q.e.d.

For a readable presentation it is also important that parts of the proof which are not of interest for the user can be hidden. Also expressions should be represented in a compact manner. Therefore, it is advisable to run a simplifier (as described in Section 7.3) on the resulting proof. Another way of hiding sub-proofs has been built into SETHEO. With a command-line option (`-hide`), all occurrences of literals with a given predicate symbol in the proof are considered to be closed by a single inference step. Thus, subproofs for that subgoal (which can be of considerable length) are condensed into a single inference step.

Example 7.7.7. In PIL/SETHEO, single components of a protocol message and submessages are extracted by auxiliary predicates `oneof` and `subset`. These are defined recursively to proceed through the message and select the appropriate components. For the reader of a proof, however, this recursive calculation of the submessages is not important, the user only wants to know that $m \in \{m_1, \ldots, m_n\}$ or $M \subset \{m_1, \ldots, m_n\}$. Therefore, in PIL/SETHEO, SETHEO is invoked with the option `-hide oneof -hide subset`. In the proof shown in the previous example, this simplification is introduced by a rule called *Submessage*.

7.8 Pragmatic Considerations

Often, the application of an automated theorem prover fails due to pragmatic reasons. As with any combined system, clear interfaces between the individual components are required. Only then, a combination can work successfully and the overhead to develop a running system can be kept reasonably low. In the following, we will discuss issues which are important for the integration of ATPs into an application system. Although most of these topics are obvious for any software development process, they have been badly neglected in ATP design.

7.8.1 Input and Output Language

The input language of an automated theorem prover for first-order predicate logic is obviously first-order predicate logic. However, its syntactical representation differs from prover to prover. Even attempts to standardize the input (like TPTP syntax [Sutcliffe *et al.*, 1994], or the common syntax proposal developed within the German Schwerpunkt Deduktion) did not bring much success. The reason is that such standards are developed on top of the provers, thus only resulting in syntax-converters between the prover's native syntax and the standardized one.

The prover's native syntax often contains extensions to standard logic for handling certain features of the prover itself. Hence, it is really important that the input language is *precisely* defined and documented. Furthermore, the prover's parser must comply to that specification and should perform a rigorous error treatment. Any peculiarities and limitations of the input and output language must be defined and documented, in particular

size of the formula: Many systems have restrictions with respect to the number of clauses, number of literals per clause, the maximal nesting depth of terms, maximal number of variables, maximal arity of predicate and function symbols, or length of identifiers, and others. These restrictions should be documented and handled as severe errors whenever they occur. Otherwise, the prover can become (unexpectedly) incomplete and

unsound. The latter case can occur for example, if clauses containing too many literals are silently truncated.
If implementation techniques make a limit unavoidable, it should be adjustable by additional command-line parameters, or at least by re-compilation of the sources. A good idea is to have a command-line option `-sizes` to print all currently defined limits.

reserved symbols: Often, certain symbols or identifiers are reserved for special purposes (e.g., control of the prover or PROLOG-style built-in predicates). Such symbols have a specific semantics which is usually completely different from that of the corresponding first-order symbol.

Example 7.8.1. In PROLOG (Horn clause logic) the infix-function symbol '+' has a different semantics than in first-order logic with equality. In PROLOG, this addition symbol is used to *evaluate* the expression which requires its arguments to be instantiated. Thus a literal $X = Y + 5$ with an uninstantiated variable Y is not legal in PROLOG (even in a fully declarative logic program), whereas in first-order logic this expression is allowed. Unfortunately, some PROLOG systems and even theorem provers (e.g., a very early version of SETHEO) treat uninstantiated variables as 0, thus being incorrect.

handling of limits: Syntactically extreme cases, like empty formulas, unbound variables, or set of axioms without a theorem must be handled correctly and should not lead to undefined behavior. Since such cases can (but need not) indicate problems and errors in the proof obligation, additional information should be returned by the prover.

7.8.2 Software Engineering Issues

An automated theorem prover is nothing more or less than a complex software system. Hence, its development and maintenance should be performed according to well known techniques of software engineering. According to the author's experience, the most crucial issues include:

- concise *version control* with the appropriate documentation (what were the changes and who applied the changes),
- *portability* to other hardware and software platforms,
- *generation* of the system (e.g., by using make),
- concise and up-to-date *documentation*, and
- automatic *installation* of the system on the target platform and *support*.

In particular, installation of the software can be quite complicated and error-prone, especially if the prover and its supporting modules (pre-, post-processors) are implemented in different languages and require a specific system set-up. Here, tools for packaging and installation of operating system software which is capable of performing pre- and post-installation scripts should be used.

The design of an automated theorem prover must not only focus on the proof procedure itself, but also on supporting modules. Only if the supporting modules (e.g., preprocessor, compiler) are designed and implemented in an efficient and robust way, the overall system can ensure high reliability and good performance. For example, the prover's algorithms must be able to scale up with the size of the formula, so symbol tables with linear search or the generation of large intermediate files is not a good idea.

Example 7.8.2. In many applications, sets of proof tasks have to be processed which are very similar to each other. Given the same set of axioms and hypotheses they only differ in the theorem to be proven. Most theorem provers, however, have to preprocess the entire formula, resulting in much overhead especially when the formulas are large.

Here, standard techniques of compiler design, namely separate preprocessing and compilation of parts of the formula with a subsequent "linking" phase ought to be used in such cases. Then, large but shared parts of the proof tasks (like sets of axioms) are needed to be preprocessed (and compiled) only once.

8. Conclusions

8.1 Summary

Today's automated theorem provers are restricted "more by general usability than by raw deductive power"[1]. In this book we have studied the application of automated theorem provers in the area of software engineering. Whenever formal methods (or formal methods tools) are used in a software development process, proof tasks arise which have to be processed by means of deduction, for example by an automated theorem prover.

Practical application of an automated theorem prover poses quite different requirements on the prover than traditional mathematical problems or benchmarks (e.g., the TPTP library [Sutcliffe *et al.*, 1994]). In particular, a prover should be capable of solving (straightforward) *inductive problems*, handling *sorted logic* and logic with *equality* in an efficient way. An ATP must be able to process large formulas, a task which requires *simplification* and *selection of axioms*. When a proof can be found, the automated system should return a *proof* (possibly human-readable) or *answer substitutions*. Otherwise, at least trivial and obvious *non-theorems* should be recognized as such and additional feedback on what went wrong should be provided upon request. Because applications usually pose severe restrictions upon answer times and computing resources, the prover must be *controlled* in such a way that many problems can be solved within short run-times. Furthermore, prover-specific details of control need to be *hidden* from the user. Finally, a practical usable theorem prover must conform to strict *software engineering criteria* with respect to stability, robustness, clean interfaces, and documentation.

In this book we have investigated these requirements in detail. They are imposed by central *characteristics* of the application's proof tasks to be solved. Typically, we can distinguish between proving in the large (many large proof tasks) or proving in the small, and deep or shallow proving. Deep proofs are long and have a complex structure, whereas shallow proofs only require few inference steps. Further important criteria concern the expected answer time ("results-while-u-wait" or batch operation), the source logic and its relation to first-order logic, and expected results.

[1] M. Kaufmann in his invited talk during CADE 15 [Kaufmann, 1998].

We have demonstrated in this book that current technology automated theorem provers can be applied successfully in key-areas of software engineering. These areas include *verification of protocols*, *analysis and verification of authentication protocols*, and *logic-based software reuse*. Several successful case studies carried out by the author are presented in detail. These areas are top-of-the-line topics with respect to formal methods usage. Current trends in computing lead towards distributed computing and fast, yet reliable software development. The introduction of automated theorem proving in this area is a promising way to achieve security, safety and reliability of large software systems. The application of automated theorem provers will push forward the development of formal methods tools which show a higher degree of automated processing.

Our selection of case studies is aimed towards diversity of addressed topics. Whereas one case study required handling of formulas in a modal logic, others had to deal with inductive problems. The number of processed proof tasks strongly varied between the case studies (from about 15 to more than 14,000). Two of the case studies were carried out with respect to a formal methods tool: for software reuse, the theorem prover was integrated into the existing tool NORA/HAMMR. On the other hand, the tool PIL/SETHEO for protocol analysis was developed by the author within the case study. All case studies had in common that the proof tasks which had to be processed were non-trivial. The global aim was to elucidate requirements and meet these requirements with further development both of the ATP (control, handling of non-theorems, preselection of axioms) and of the ATP-embedded system interface (preprocessing, logical translation, human-readable proofs).

A generic system architecture and implementations of key-techniques leading to success have been described in this book. The main original contributions concern control of the prover for optimal performance by combination of search paradigms and parallel execution (SiCoTHEO*), and detection of non-theorems by generative methods and extended counterexample generation. Other key-techniques include translation of non first-order proof tasks and inductive proof tasks into first-order logic, simplification, handling of sorts, and postprocessing in order to generate answer substitutions and human-readable proofs. Furthermore, a variety of methods and practical solutions were developed to increase general usability of the automated prover within embedded systems.

In this book we have given a (as far as possible hands-on) methodology for approaching new applications of automated provers, opening up many possibilities for useful integration of automated theorem provers in formal methods tools. However, automated theorem provers for first-order logic cannot be used for each and every proof task — their deductive power is still too limited to handle real complex proof obligations (e.g., finding invariants). In order to push forward these limits, we will discuss the following important questions.

8.2 Questions and Answers

- Q: *Are these results "portable" to other automated provers (in particular resolution-based provers) and other application domains?*
  Certainly, most of the techniques presented in this book have been developed or adapted with respect to the prover SETHEO. However, they easily carry over to other top-down theorem provers (e.g., Protein, PTTP, METEOR, ...) and, with the exception of equality handling and DELTA-preprocessing, also to resolution-style provers like OTTER, Gandalf, or SPASS. In fact, proof tasks from several of our case studies (e.g., logic-based component retrieval in Section 6.3 or the verification of the Stenning communication protocol (Section 6.2)) have also been carried out with other high performance provers like OTTER, SPASS, or Protein. Interactive verification systems like ILF of KIV [Reif *et al.*, 1997] even have connected (or are planning to connect) several theorem provers simultaneously.
  Applications of ATPs in other domains like knowledge processing (e.g., the Stanford knowledge-sharing system [Genesereth *et al.*, 1994]) or processing of natural language (e.g., [Scheler and Schumann, 1995]) set up similar requirements as in our case. Here, however, questions of representation in logic, handling of non-monotonicity and reasoning with uncertainties play a much bigger role.
- Q: *Has the methodology and potential for ATP applications been exhaustively studied?*
  No. Although the basic building blocks of the architecture have been presented and described in this books, many methods and techniques still need to be enhanced. There are two topics which have major potential for improvement: handling of theories and detection of non-theorems. Current theorem provers are relatively weak with respect to proof tasks containing additional theories, like equality, arithmetic, recursive data structures, or finite domains. Here, a tight combination between theorem proving and theory reasoning by specific procedures (often decision procedures) will result in considerable improvements. Additionally, simplification with respect to the given theories must be incorporated into the automated prover.
  Although several approaches to detecting non-theorems have been presented in this book, these methods are still too weak to reliably detect most of the non-theorems and the feedback they provide still needs to much user's intuition to be really applicable. Here, much improvements can be made with respect to practical usability.
- Q: *What is actually possible with current ATP technology?*
  Automated theorem provers have gained tremendously in deductive power over the last few years. Our case studies and related work revealed that in certain domains, important applications can be fully automated. Application areas with many "low complexity" proof tasks, or where the complex proof obligations can easily be broken down in simpler ones are particularly

promising for fully automatic processing. The following application areas are the author's favorite candidates:

- *verification* of protocols and protocol stacks, authentication protocols, linear (macro) code, legacy code, or graphical user interfaces. These domains have in common that they do not contain any recursive data-types and control mechanisms which would require highly complicated inductive proof tasks to be solved. Furthermore, abstraction on many details can safely be made (e.g., encryption algorithms in authentication protocols or representation details in graphical user interfaces).
- *synthesis* of programs with little flow control in the areas mentioned above (e.g., AMPHION). Here again, without complicated flow control and ample space for abstraction, proof tasks arise which can be handled automatically.
- *logic-based component retrieval* is an ideal candidate when abstraction (e.g., by using contracts as in NORA/HAMMR) allows to reduce the complexity of proof tasks.
- *generation of test-suites* for the validation of software. A good test-suite should contain test cases for all critical parts of the software. In VLSI-design such test-patterns are automatically generated out of the specification. In the area of software engineering, only few approaches have been made (e.g., [Wang and Goldberg, 1994]) in this direction. Here, focus on simply structured pieces of software like legacy code or graphical user interfaces (e.g., [Brandl, 1995]) can largely facilitate the automatic generation of test cases using deductive methods.
- *optimization* of code or specifications (e.g., protocol stacks for groupware [Kreitz *et al.*, 1998]).
- *configuration management* for software and hardware configuration.

In numerous other domains, current technology theorem provers, when adapted accordingly, can support the systems and relieve the user from much low-level work. Currently, we are at the verge of getting many simple ATP applications which can be used in practice (i.e., not only by the developers of the corresponding automated prover).

- Q: *What are the long-term goals of the application of automated provers in the area of software engineering?*

  Even with much further development, automated provers will probably never be strong enough to process *all* arising proof tasks. The invention of "central proof ideas" will certainly be left to the users. However, automated provers may become similar to what advanced pocket calculators or algebraic systems (like Mathematica [Wolfram, 1991], Maple [Char *et al.*, 1988], or Matlab [Biran and Breiner, 1995]) have become for engineers: setting up (differential) equations is done manually, but doing the calculation and visualization is performed automatically. Here, both the input and output representation has adapted to the engineers' formalisms and needs such that these tools can be used without much effort.

For logic and logic-related tools, things are still quite different: just compare the effort it takes to calculate $\sqrt{2}$ versus solving a simple logical riddle manually and using a tool (calculator and automated theorem prover, respectively).
With any application, it must be checked carefully, if an ATP is the right tool. There always exist domains where, for instance, model checkers or other decision procedures are much more efficient than a first-order prover ever could be. Here, on the long term, reasoning systems, combining the advantages of automated theorem proving and other methods will evolve.

- Q: *Where are the problems and challenges?*
  Further work needs to be done before the entire potential of automated theorem provers in applications can be exploited. Here, three important areas can be identified: (1) handling of non-theorems and feedback in cases of errors, (2) flexibility and level of reasoning, and (3) practical usability within a tool.

  1. A formal methods tool should work somewhat similar to a compiler. If the specification(s) are correct, it should process them silently (and possibly deliver a proof for documentation or reviewing purposes). If, however, something goes wrong (i.e., a conjecture turns out to be not provable or a non-theorem), the user needs feedback on where the error might be. Like with compilers, the feedback must be on the level of the problem. E.g., a compiler error message "reduce conflict on rule 45" (instead of "error: missing endif") is as cryptic and unacceptable as would be the output of an automated prover: "clause 17: must add a p__98(a_37) to prove conjecture".
     Also the messages should be descriptive, not just stating "theorem invalid" (which is as helpful as a "syntax error: aborted"). Here, not only machine-oriented feedback, but also its presentation in a human-readable form is important (for first steps see, e.g., KIV or ILF (Section 7.7.4)). Here, work must be invested, because with first-order theorem proving, things are much more complicated than for compilers because of the undecidability of the logic.
  2. Many applications require a high flexibility with respect to the logics, underlying the proof tasks. Often even single proof obligations contain a variety of different problem classes, like finite-domain problems, inductive problems, etc. Here, any single automated reasoner is not feasible. Rather, a set of (interacting and even cooperating) proof methods is required.
  3. Although this book is intended to facilitate practical usability of automated theorem provers, it only can be a single step into this direction. In the near future, all applications need a tight cooperation between the designers of formal methods tools and automated theorem provers. Only then, tools which are powerful and usable (by software engineers) can be designed and implemented.

- Q: *What can be the impact of successful application of ATPs for the field of software engineering and the design of safe and secure software?*
  An automated theorem prover itself (or a formal methods tool using it) cannot produce safe and secure software. Rather formal methods and their tools must be integrated into the entire software development process and their use by the software engineers be encouraged. These issues are manifold and we will not discuss them here, but cf. [Weber-Wulff, 1993; Storey, 1996; Stevenson, 1990; Kelly, 1997; Rushby, 1993; Craigen, 1999].
  Successful application of automated provers, as demonstrated in this book will lead to formal methods tools which are much more effective and easy to use than nowadays' tools, because their degree of automatic processing will be reasonably high. Such tools, if (and only if) properly designed for practical and industrial use will eventually find their way to widespread application. Although never all of the software written will be developed in a fully formal environment, tools based on automated deduction methods are capable of considerably improving safety and security of software.

References

[Abramsky *et al.*, 1992] S. Abramsky, D. M. Gabbay, and T. S. Maibaum. *Handbook of Logic in Computer Science*, volume 2 (Background: Computational Structures). Clarendon Press, 1992.

[Abrial *et al.*, 1991] J. R. Abrial, M. K. Lee, D. S. Neilson, P. N. Scharbach, and I. H. Sorensen. The B-method (software development). In [Prehn and Toetenel, 1991], pages 398–405.

[Abrial *et al.*, 1996] J.–R. Abrial, E. Börger, and H. Langmaack, eds. *Formal Methods for Industrial Applications: Specifying and Programming the Steam Boiler Control.* Volume 1165 of *Lecture Notes in Computer Science.* Springer-Verlag, 1996.

[Aho *et al.*, 1988] A. Aho, B. Kernighan, and P. Weinberger. *The AWK Programming Language.* Addison-Wesley, 1988.

[Alur and Henzinger, 1996] R. Alur and Th. Henzinger, eds. *CAV '96, Proceedings of the 8th International Conference on Computer-Aided Verification*, volume 1102 of *Lecture Notes in Computer Science.* Springer-Verlag, 1996.

[Ambler *et al.*, 1977] A. L. Ambler, D. I. Good, J. C. Browne, W. F. Burger, R. C. Cohen, C. G. Hoch, and R. E. Wells. GYPSY: A Language for Specification and Implementation of Verifiable Programs. In *Proceedings of the ACM Conference Language Design for Reliable Software. ACM SIGPLAN Notices*, 12(3):1–10, 1977.

[Andrews, 1986] P. Andrews. Theorem Proving via General Matings. *Journal of the ACM*, 28:193–214, 1986.

[Archinoff *et al.*, 1990] G. H. Archinoff, R. J. Hohendorf, A. Wassyng, B. Quigley, and M. R. Borsch. Verification of the Shutdown System Software at the Darlington Nuclear Generating Station. In *International Conference on Control and Instrumentation in Nuclear Installations*, 1990.

[Armando and Jebelean, 1999] A. Armando and T. Jebelean, eds. *Proceedings of Calculemus – Systems for Integrated Computation and Deduction*, volume 23 of *Electronic Notes in Theoretical Computer Science*, Elsevier, 1999.

[Astrachan and Loveland, 1991] O. L. Astrachan and D. W. Loveland. METEORs: High Performance Theorem Provers Using Model Elimination. In R. S. Boyer, ed., *Automated Reasoning: Essays in Honor of Woody Bledsoe.* Kluwer, 1991.

[Brock *et al.*, 1996] B. Brock, M. Kaufmann, and J. S. Moore. ACL2 Theorems about Commercial Microprocessors. In M. Srivas and A. Camilleri, eds., *First international Conference on Formal Methods in Computer-Aided Design*, volume 1166 of *Lecture Notes in Computer Science*, pages 275–293. Springer-Verlag, 1996.

[Bahlke and Snelting, 1985] R. Bahlke and G. Snelting. The PSG Programming System Generator. *ACM SIGPLAN Notices*, 20(7):28–33, 1985.

[Bahlke and Snelting, 1986] R. Bahlke and G. Snelting. The PSG System: From Formal Language Definitions to Interactive Programming Environments. *ACM Transactions on Programming Languages and Systems*, 8(4):547–576, 1986.

[Barrett, 1989] G. Barrett. Formal Methods Applied to a Floating-point number system. *IEEE Trans. Software Eng.*, 15(5):611–621, 1989.

[Barringer *et al.*, 1984] H. Barringer, J. H. Cheng, and C. B. Jones. A logic covering undefinedness in program proofs. *Acta Informatica*, 21(3):251–269, 1984.

[Bartlett *et al.*, 1969] K. A. Bartlett, R. A. Scantlebury, and P. T. Wilkinson. A note on reliable full-duplex transmission over half-duplex links. *Communications of the ACM*, 12:260–261, 1969.

[Baumgartner and Furbach, 1994a] P. Baumgartner and U. Furbach. Model Elimination without Contrapositives. In [Bundy, 1994], pages 769–773.

[Baumgartner and Furbach, 1994b] P. Baumgartner and U. Furbach. Protein: A PROver with a Theory Extension INterface. In [Bundy, 1994], pages 769–773.

[Baumgartner and Schumann, 1995] P. Baumgartner and J. Schumann. Implementing Restart Model Elimination and Theory Model Elimination on Top of SETHEO. Forschungsbericht 5–95, Universität Koblenz-Landau, 1995.

[Bayerl *et al.*, 1988] S. Bayerl, R. Letz, and J. Schumann. SETHEO, A SEquential THEOrem Prover for First Order Logic. ATP-Report, Technische Universität München, 1988.

[Bear, 1991] S. Bear. An Overview of HP-SL. In [Prehn and Toetenel, 1991], pages 571–587.

[Beckert and Hähnle, 1992] B. Beckert and R. Hähnle. An Improved Method for Adding Equality to Free Variable Semantic Tableau. In [Kapur, 1992], pages 507–521.

[Beierle and Meyer, 1994] C. Beierle and G. Meyer. Runtime Type Computations in the Warren Abstract Machine. *Journal of Logic Programming*, 18(2):123–148, 1994.

[Benzmüller *et al.*, 1997] C. Benzmüller, L. Cheikhrouhou, D. Fehrer, et al. ΩMEGA: Towards a mathematical Assistant. In [McCune, 1997], pages 252–255.

[Bibel and Eder, 1993] W. Bibel and E. Eder. *Methods and Calculi for Deduction.* In [Gabbay, 1993], Volume 1: Logical Foundations, pages 68–182, 1993.

[Bibel and Schmitt, 1998] W. Bibel and P. Schmitt, eds. *Automated Deduction: a Basis for Applications*, volume 8–10 of Applied Logic Series. Kluwer, 1998.

[Bibel *et al.*, 1987] W. Bibel, R. Letz, and J. Schumann. Bottom-up Enhancements of Deductive Systems. In I. Plander, ed., *Artificial Intelligence and Information – Control Systems of Robots 87*, pages 1–9. North-Holland, 1987.

[Bibel *et al.*, 1994] W. Bibel, S. Brüning, U. Egly, and T. Rath. Komet. In [Bundy, 1994], pages 783–787.

[Bibel, 1987] W. Bibel. *Automated Theorem Proving.* Verlag Vieweg, second edition, 1987.

[Biran and Breiner, 1995] A. Biran and M. Breiner. *MATLAB for Engineers.* Addison-Wesley, 1995.

[Bjørner *et al.*, 1990] D. Bjørner, C. A. R. Hoare, and H. Langmaack, eds. *VDM '90: VDM and Z — Formal Methods in Software Development*, volume 428 of *Lecture Notes in Computer Science.* Springer-Verlag, 1990.

[Bolignano, 1998] D. Bolignano. Integrating proof-based and model-checking techniques for the formal verification of cryptographic protocols. In [Hu and Vardi, 1998], pages 79–87.

[Börger and Rosenzweig, 1995] E. Börger and D. Rosenzweig. The WAM – Definition and Compiler Correctness. In *Logic Programming: Formal Methods and*

Practical Applications, volume 11 of *Studies in Computer Science and Artificial Intelligence*. North-Holland, 1995.

[Bose *et al.*, 1989] S. Bose, E. M. Clarke, D. E. Long, and S. Michaylov. Parthenon: A Parallel Theorem Prover for Non-Horn Clauses. *In Proceedings Logic in Computer Science (LICS)*, pages 1–10. IEEE Press, 1989.

[Boswell, 1995] A. Boswell. Specification and Validation of a Security Policy Model. *IEEE Transactions on Software Engineering*, 21(2):99–106, 1995.

[Boy de la Tour, 1992] T. Boy de la Tour. An optimality result for clause form translation. *Journal of Symbolic Computation*, 14(4):283–302, 1992.

[Boyer and Moore, 1988] R. S. Boyer and J. S. Moore. *A Computational Logic Handbook*. Academic Press, 1988.

[Boyer and Moore, 1990] R. S. Boyer and J. S. Moore. A theorem prover for a computational logic. In [Stickel, 1990], pages 1–15.

[Boyer and Yu, 1996] R. S. Boyer and Y. Yu. Automated proofs of object code for a widely used microprocessor. *Journal of the ACM*, 43(1):166-192, 1996.

[Brackin, 1996] S. H. Brackin. A HOL Extension of GNY for Automatically Analyzing Cryptographic Protocols. In *Proc. IEEE Computer Security Foundations Workshop IX*, pages 62–76. IEEE Press, 1996.

[Brand, 1975] D. Brand. Proving Theorems with the Modification Method. *SIAM Journal of Computing*, 4(4):412–430, 1975.

[Brandl, 1995] A. Brandl. Verifizierbare und interpretierbare Modelle von graphischen Benutzeroberflächen. Diploma thesis, Technische Universität München, Institut für Informatik, 1992 (in German).

[Bündgen, 1998] R. Bündgen. *Termersetzungssysteme: Theorie, Implementierung, Anwendung*. Verlag Vieweg, 1998 (in German).

[Bundy *et al.*, 1990] A. Bundy, F. van Harmelen, C. Horn, and A. Smaill. The Oyster–Clam System. In [Stickel, 1990], pages 647–648.

[Bundy, 1994] A. Bundy, ed. *Proc. 12th International Conference on Automated Deduction (CADE-12)*, volume 814 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag, 1994.

[Bundy, 1996] A. Bundy. Proof Planning. In B. Drabble, ed., *Proceedings of the 3rd International Conference on AI Planning Systems (AIPS-96)*, pages 261–267. AAAI Press, 1996.

[Burch *et al.*, 1990] J. R. Burch, E. M. Clarke, K. L. McMillan, and D. L. Dill. Sequential Circuit Verification Using Symbolic Model Checking. In *Proceedings of the 27th ACM/IEEE Design Automation Conference*, pages 46–51. IEEE Press, 1990.

[Burch *et al.*, 1992] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic Model Checking: 10^{20} States and Beyond. *Information and Computation*, 98(2):142–170, 1992.

[Bürckert, 1985] H. J. Bürckert. Extending the Warren Abstract Machine to Many-Sorted Prolog. Memo SEKI 85-VII-KL, Universität Kaiserslautern, 1985.

[Burkart, 1997] O. Burkart. *Automatic Verification of Sequential Infinite-State Processes*, volume 1354 of *Lecture Notes in Computer Science*. Springer-Verlag, 1997.

[Burrows *et al.*, 1989] M. Burrows, M. Abadi, and R. Needham. A Logic of Authentication. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, pages 1–13. ACM Press, 1989.

[Burrows *et al.*, 1990] M. Burrows, M. Abadi, and R. Needham. A Logic of Authentication. *ACM Transactions on Computer Systems*, 8(1):18–36, 1990.

[Burstein and Smith, 1996] M. B. Burstein and D. Smith. ITAS: A Portable Interactive Transportation Scheduling Tool Using a Search Engine Generated from

Formal Specifications. In *Proceedings of the 3rd International Conference on AI Planning Systems (AIPS-96)*, pages 35–44. AAAI Press, 1996.

[Caferra and Peltier, 1997] R. Caferra and N. Peltier. A new technique for verifying and correcting logic programs. *Journal of Automated Reasoning (JAR)*, 19(3):277–318, 1997.

[Carnot *et al.*, 1992] M. Carnot, C. DaSilva, B. Dehbonel, and F. Meija. Error-free software development for critical systems using the B-methodology. In *Third International IEEE Symposium on Software Reliability Engineering*, 1992.

[Chang and Lee, 1973] C. L. Chang and R. C. T. Lee. *Symbolic Logic and Mechanical Theorem Proving.* Academic Press, 1973.

[Char *et al.*, 1988] B. W. Char, K. O. Geddes, G. H. Gonnet, B. Leong, M. B. Monagan, and S. M. Watt. *Maple Reference Manual.* WATCOM Publications Limited, fifth edition, 1988.

[Clarke and Wing, 1996] E. M. Clarke and J. Wing. Formal methods: State of the art and future directions. *ACM Comp. Surveys*, 28(4):627–643, 1996.

[Clarke *et al.*, 1993] E. M. Clarke, O. Grumberg, H. Hiraishi, S. Jha, D. E. Long, K. L. McMillan, and L. A. Nessn. Verification of the "Futurebus+" Cache Coherence Protocol. In *Proceedings of the 11th IFIP WG10.2 International Conference on Computer Hardware Description Languages and their Applications (CHDL '93)*, IFIP Transactions A, volume A–32, pages 15–30. IEEE Press, 1993.

[Clarke *et al.*, 2000] E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking.* MIT Press, 2000.

[Clavel *et al.*, 1996] M. Clavel, S. Eker, P. Lincoln, and J. Meseguer. Principles of Maude. In *Proceedings of the 1st International Workshop on Rewriting Logic and its Applications.* Electronic Notes in Theoretical Computer Science, Elsevier, 1996.

[Cleaveland *et al.*, 1993] R. Cleaveland, J. Parrow, and B. Steffen. The Concurrency Workbench: A Semantics-Based Tool for the Verification of Concurrent Systems. *ACM Transactions on Programming Languages and Systems*, 15(1):36–72, 1993.

[Clocksin and Mellish, 1984] W. F. Clocksin and C. S. Mellish. *Programming in Prolog.* Springer-Verlag, 1984.

[Constable *et al.*, 1986] R. L. Constable et al. *Implementing Mathematics with the NUPRL Proof Development System.* Prentice Hall, 1986.

[Corbin and Bidoit, 1983] J. Corbin and M. Bidoit. A Rehabilitation of Robinson's Unification Algorithm. In R. Mason ed., *Information Processing 83. Proceedings of the 9th IFIP World Computer Congress*, pages 909–914. North-Holland, 1983.

[Cousot and Cousot, 1977] 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 ACM Symposium on Principles of Programming Languages*, pages 238–252. ACM Press, 1977.

[Craigen, 1999] D. Craigen. Formal Methods Diffusion: Past Lessons and Future Prospects. Technical Report D/167/6101/1, Adelard, 1999.

[Craigen and Saaltink, 1996] D. Craigen and M. Saaltink. Using EVES to Analyze Authentication Protocols. Technical Report TR-96-5508-05, ORA Canada, 1996.

[Craigen *et al.*, 1993] D. Craigen, S. Gerhart, and T. Ralston. An International Survey of Industrial Applications of Formal Methods. Technical Report NIST GCR 93/626 (Vol. 1–2), National Institute of Standards and Technology, 1993. `ftp://www.nist.gov/itl/lab/nistirs/nistirs.htm`.

[Crow *et al.*, 1995] J. Crow, S. Owre, J. Rushby, N. Shankar, and M. Srivas. A Tutorial Introduction to PVS. In *WIFT '95, Proceedings of the Workshop on*

Industrial Strength Formal Specification Techniques, Computer Science Laboratory, SRI International, 1995.

[Croxford and Sutton, 1996] M. Croxford and J. Sutton. Breaking through the V&V bottleneck. In M. Toussaint, ed., *Proceedings of Ada in Europe. Second International Eurospace – Ada Europe Symposium*, pages 344–354. Springer-Verlag, 1996.

[c't98, 1998] T-online: Hacker knacken Zugangsdaten. *c't Computermagazin*, (7/98):62ff, 1998 (in German).

[Cusumano and Selby, 1997] M. Cusumano and R. Selby. How Microsoft Builds Software. *Communications of the ACM*, 40(6):53–62, 1997.

[Dahn and Schumann, 1998] B. I. Dahn and J. Schumann. *Using Automated Theorem Provers in Verification of Protocols.* In [Bibel and Schmitt, 1998], chapter III.10, pages 195–224.

[Dahn and Wernhard, 1997] B. I. Dahn and C. Wernhard. First Order Proof Problems extracted from an Article in the MIZAR Mathematical Library. In *Proceedings of the 1st International Workshop on First-Order Theorem Proving (FTP)*, pages 58–62, 1997.

[Dahn and Wolf, 1994] B. I. Dahn and A. Wolf. A Calculus Supporting Structured Proofs. *Journal for Information Processing and Cybernetics (EIK)*, 30(5–6):261–276, 1994.

[Dahn and Wolf, 1996] B. I. Dahn and A. Wolf. Natural Language Representation and Combination of Automatically Generated Proofs. In *Proceedings of the First International Workshop on Frontiers of Combining Systems (FroCoS '96)*, volume 3 of Applied Logic Series, pages 175–192. Kluwer, 1996.

[Dahn *et al.*, 1994] B. I. Dahn, J. Gehne, T. Honigmann, L. Walther, and A. Wolf. Integrating Logical Functions with ILF. Technical Report Preprint 94-10, Humboldt Universität Berlin, Department of Mathematics, 1994.

[Dahn, 1996] B. I. Dahn. The ILF Type System part I: Large Monomorphic Types. Technical Report, Humboldt Universität, Berlin, 1996.

[Davis and Putnam, 1960] M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7(3):201–215, 1960.

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

[de Boor, 1989] A. de Boor. *PMake – A Tutorial.* Berkeley Softworks, 1989.

[Dederichs *et al.*, 1993] F. Dederichs, C. Dendorfer, and R. Weber. FOCUS: A Formal Design Method for Distributed Systems. In A. Bode and M. Dal Cin, eds., *Parallel Computer Architectures*, pages 190–202. Springer-Verlag, 1993.

[Dendorfer and Weber, 1992a] C. Dendorfer and R. Weber. Development and Implementation of a Communication Protocol – An Exercise in FOCUS. SFB-Bericht Nr. 342/4/92 A, Institut für Informatik, Technische Universität München, 1992.

[Dendorfer and Weber, 1992b] C. Dendorfer and R. Weber. From Service Specification to Protocol Entity Implementation – An Exercise in Formal Protocol Development. In R.J. Linn and M. Ü. Uyar, eds., *Protocol, Specification, Testing and Verification XII*, volume C-8 of *IFIP Transactions*, pages 163–177, 1992.

[Denzinger and Dahn, 1998] J. Denzinger and B. I. Dahn. *Cooperating Theorem Provers.* In [Bibel and Schmitt, 1998], chapter II.12, pages 383–416.

[Denzinger and Pitz, 1992] J. Denzinger and W. Pitz. The DISCOUNT System. User Manual. SEKI Working Paper SWP-92-16, Universität Kaiserslautern, 1992 (in German).

[Denzinger *et al.*, 1997] J. Denzinger, M. Kronenburg, and S. Schulz. DISCOUNT: A distributed and learning Equational Prover. *Journal of Automated Reasoning (JAR)*, 18:189–198, 1997.

[Denzinger, 1995] J. Denzinger. Knowledge-Based Distributed Search Using Teamwork. In *Proceedings ICMAS-95*, pages 81–88. AAAI-Press, 1995.

[Dill, 1994] D. Dill, ed. *CAV '94, Proceedings of the 6th International Conference on Computer-Aided Verification*, volume 818 of *Lecture Notes in Computer Science*. Springer-Verlag, 1994.

[Dill, 1996] D. L. Dill. The murphi Verification System. In [Alur and Henzinger, 1996], pages 390–393.

[Dowek *et al.*, 1993] G. Dowek, A. Felty, H. Herbelin, G. P. Huet, C. Murthy, C. Parent, C. Paulin-Mohring, and B. Werner. The Coq Proof Assistant User's Guide. Version 5.8. Technical Report, INRIA Rocquencourt, 1993.

[Dräger, 1993] J. Dräger. Anwendung des Theorembeweisers SETHEO auf angeordnete Körper. Technical Report FKI-186-93, Institut für Informatik, Technische Universität München, 1993 (in German).

[Eckert, 1998] C. Eckert. Tool-Supported Verification of Cryptographic Protocols. In *Proceedings of the IFIP TC11 14th International Conference on Information Security*. Chapman & Hall, 1998.

[Eder, 1984] E. Eder. An Implementation of a Theorem Prover Based on the Connection Method. In W. Bibel and B. Petkoff, eds., *Proc. AIMSA'84, Artificial Intelligence - Methodology Systems Application*, pages 121–128. North-Holland, 1984.

[Electronic Frontier Foundation, 1998] Electronic Frontier Foundation. *Cracking DES: Secrets of Encryption Research, Wiretap Politics & Chip Design*. O'Reilly, 1998.

[Ertel, 1992] W. Ertel. OR-Parallel Theorem Proving with Random Competition. In A. Voronkov, ed., *Proceedings of LPAR '92*, volume 624 of *Lecture Notes in Artificial Intelligence*, pages 226–237. Springer-Verlag, 1992.

[Estenfeld, 1986] K. Estenfeld. ECRC Prolog User's Manual Version 1.0. Technical Report TR-LP-08, ECRC, 1986.

[Fischer and Schumann, 1997] B. Fischer and J. Schumann. SETHEO Goes Software Engineering: Application of ATP to Software Reuse. In [McCune, 1997], pages 65–68.

[Fischer and Snelting, 1997] B. Fischer and G. Snelting. Reuse by contract. In *ESEC/FSE'97 Workshop on Foundations of Component-Based Software*, 1997.

[Fischer *et al.*, 1998] B. Fischer, J. Schumann, and G. Snelting. Deduction based component retrieval. In [Bibel and Schmitt, 1998], chapter III.12, pages 265–292.

[Fischer, 2001] B. Fischer. *Deduction Based Software Component Retrieval*. Dissertation, Universität Passau, 2001.

[Fitzgerald *et al.*, 1997] J. Fitzgerald, C. B. Jones, and P. Lucas, eds., *FME '97: Industrial Applications and Strengthened Foundations of Formal Methods, Proceedings of the 4th International Symposium of Formal Methods Europe* , volume 1313 of *Lecture Notes in Computer Science*, Springer-Verlag, 1997.

[Fowler, 1997] M. Fowler. *UML Distilled*. Addison-Wesley, 1997.

[Frakes and Nejmeh, 1987] W-B. Frakes and B. A. Nejmeh. Software reuse through information retrieval. In E. Stohr *et al.*, eds. *Proceedings of the Twentieth Hawaii International Conference on System Sciences*, volume 2, pages 530–535, 1987.

[Fronhöfer, 1996] B. Fronhöfer. Cyclic Rules in Linear Connection Proofs. In G. Görz and S. Hölldobler, eds., *KI-96: Advances in Artificial Intelligence, Lecture Notes in Artificial Intelligence* 1137, pages 67–70, Springer-Verlag, 1996.

[Fuchs and Wolf, 1998] M. Fuchs and A. Wolf. System Description: Cooperation in Model Elimination: CPTHEO. In [Kirchner and Kirchner, 1998], pages 142–146.

[Fujita *et al.*, 1992] M. Fujita, R. Hasegawa, M. Koshimura, and H. Fujita. Model generation theorem provers on a parallel inference machine. In *Proceedings of the International Conference on Fifth Generation Computer Systems.* ICOT, 1992.

[Fujita *et al.*, 1993] M. Fujita, J. Slaney, and F. Bennett. Automatic generation of some results in finite algebra. In R. Bajcsy, ed., *Proceedings International Joint Conference on Artificial Intelligence (IJCAI '93)*, volume 1, pages 52–57. Morgan Kaufmann Publishers, 1993.

[Furbach *et al.*, 1995] U. Furbach, P. Baumgartner and F. Stolzenburg. Model Elimination, Logic Programming and Computing Answers. Fachberichte Informatik 1–95, Universität Koblenz-Landau, 1995.

[Furbach *et al.*, 1998] U. Furbach, M. Kühn, and F. Stolzenburg. Model-Guided Proof Debugging. Fachberichte Informatik 6–98, Universität Koblenz-Landau, 1998.

[Furbach, 1992] U. Furbach. Computing answers for disjunctive logic programs. In D. Pearce and D. Wagner, eds., *Proceedings of the European Workshop JELIA '92 on Logics in AI*, volume 633 of *Lecture Notes in Artificial Intelligence*, pages 357–372. Springer-Verlag, 1992.

[Furbach, 1998] U. Furbach. *Tableaux and Connection Calculi: Introduction*, In [Bibel and Schmitt, 1998], chapter I.0, pages 3–10.

[Gabbay, 1993] D. Gabbay, ed. *Handbook of Logic in AI and Logic.* Oxford, 1993. Volumes I, II.

[Gabbay and Guenthner, 1983] D. Gabbay and F. Guenthner eds. *Handbook of Philosophical Logic.* Kluwer, 1983.

[Ganzinger *et al.*, 1997] H. Ganzinger, C. Meyer, and C. Weidenbach. Soft typing for ordered resolution. In [McCune, 1997], pages 321–335.

[Ganzinger, 1999] H. Ganzinger, ed., *Proc. 16th International Conference on Automated Deduction (CADE-16).* Volume 1632 of *Lecture Notes in Artificial Intelligence*, Springer-Verlag, 1999.

[Gaudel and Woodcock, 1996] M. C. Gaudel and J. Woodcock, eds. *FME '96: Industrial Benefit and Advances in Formal Methods, Proceedings of the Third International Symposium of Formal Methods Europe* , volume 1051 of *Lecture Notes in Computer Science*, Springer-Verlag, 1996.

[Gaul, 1995] T. Gaul. An abstract state machine specification of the DEC-alpha processor family. Verifix Working Paper [Verifix/UKA/4], University of Karlsruhe, 1995.

[Geiger, 1995] J. Geiger. Formale Methoden zur Verifikation kryptographischer Protokolle. Fortgeschrittenenpraktikum, Institut für Informatik, Technische Universität München, 1995 (in German).

[Geist *et al.*, 1994] A. Geist et al. *PVM: Parallel Virtual Machine. A User's Guide and Tutorial for Networked Parallel Computing.* MIT Press, 1994.

[Genesereth *et al.*, 1994] M. Genesereth, N. P. Singh, and M. A. Syed. A Distributed and Anonymous Knowledge Sharing Approach to Software Interoperation. Stanford University, 1994. Available as `http://logic.stanford.edu/sharing/knowledge.html`.

[George *et al.*, 1992] C. George, P. Haff, K. Havelund, A. E. Haxthausen, R. Milne, C. Bendix Nielson, S. Prehn, and K. R. Wagner. *The Raise Specification Language.* Prentice Hall, 1992.

[Gerhart *et al.*, 1994] S. Gerhart, D. Craigen, and T. Ralston. Experience with formal methods in critical systems. *IEEE Software*, 11(1):21–39, 1994.

[Gilmore, 1960] P. C. Gilmore. A Proof Method for Quantification Theory: Its Justification and Realization. *IBM Journal of Research and Development*, 4:28–35, 1960.

[Glass, 1999] R. Glass. The Realities of Software Technology Payoffs. *Communications of the ACM*, 42(2):74–79, 1999.

[Goguen *et al.*, 1993] J. Goguen, T. Winkler, J. Meseguer, K. Futatsugi, and J. P. Jouannaud. Introducing OBJ. In J. Goguen, ed., *Applications of Algebraic Specification using OBJ*. Cambridge, 1993.

[Goller *et al.*, 1994] C. Goller, R. Letz, K. Mayr, and J. Schumann. SETHEO V3.2: Recent Developments (System Abstract). In [Bundy, 1994], pages 778–782.

[Gong *et al.*, 1990] L. Gong, R. Needham, and R. Yahalom. Reasoning about belief in cryptographic protocols. In *Proc. of IEEE Symposium on Security and Privacy, Oakland, Ca., USA*, pages 234–248. IEEE, 1990.

[Good *et al.*, 1988] D. Good, B. DiVito, and M. Smith. Using the Gypsy Methodology. Technical Report, Computational Logic Inc., Austin, TX, 1988.

[Gordon, 1988] M. J. C. Gordon. HOL: A proof generating system for higher-order logic. In G. Birtwistle and P. A. Subrahmanyam, eds., *VLSI Specification, Verification and Synthesis*, pages 73–128. Kluwer, 1988.

[Graf, 1996] P. Graf. *Term Indexing*, volume 1053 of *Lecture Notes in Computer Science*. Springer-Verlag, 1996.

[Guttag *et al.*, 1993] J. V. Guttag, J. J. Horning, S. J. Garland, K. D. Jones, A. Modet, and J. M. Wing. *Larch: Languages and Tools for Formal Specification*. Texts and Monographs in Computer Science. Springer-Verlag, 1993.

[Hähnle *et al.*, 1992] R. Hähnle, B. Beckert, S. Gerberding, and W. Kernig. The Many-Valued Tableau-Based Theorem Prover $_3T^AP$. Technical Report, IBM Germany Scientific Center Institute of Knowledge Based Systems, 1992.

[Hähnle, 1993] R. Hähnle. *Automated Theorem Proving in Multiple-Valued Logics*. Oxford University Press, 1993.

[Hall, 1996] A. Hall. Using formal methods to develop an ATC information system. *IEEE Software*, 12(6):66–76, 1996.

[Harel, 1987] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, 1987.

[Hasegawa *et al.*, 1992] R. Hasegawa, M. Koshimura, and H. Fujita. Lazy Model Generation for Improving the Efficiency of Forward Reasoning Theorem Provers. In *Proceedings of the International Workshop on Automated Reasoning (IWAR'92)*, pages 191–202, 1992.

[Havelund and Shankar, 1996] K. Havelund and N. Shankar. Experiments in Theorem Proving and Model Checking for Protocol Verification. In [Gaudel and Woodcock, 1996], pages 662–681.

[Heimdahl and Leveson, 1996] M. Heimdahl and N. Leveson. Completeness and consistency in hierarchical statebased requirements. *IEEE Transactions on Software Engineering*, 22(6):363–377, 1996.

[Heninger, 1980] K. Heninger. Specifying software requirements for complex systems. *IEEE Transactions on Software Engineering*, 6(1):2–13, 1980.

[Hillenbrand *et al.*, 1999] T. Hillenbrand, A. Jaeger, and B. Löchner System Description: Waldmeister: Improvements in Performance and Ease of Use. In [Ganzinger, 1999], pages 232–236.

[Hoare, 1983] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 26(1):100–106, 1983.

[Holzmann, 1991] G. J. Holzmann. *Design and Validation of Computer Protocols*. Prentice Hall, 1991.

[Houstan and King, 1991] I. Houstan and S. King. CICS Project Report: experiences and results from the use of Z in IBM. In [Prehn and Toetenel, 1991], pages 588–596.

[Hu and Vardi, 1998] A. J. Hu and M. Y. Vardi, eds. *CAV '98, Proceedings of the 10th International Conference on Computer-Aided Verification*, volume 1427 of *Lecture Notes in Computer Science*. Springer-Verlag, 1998.

[Huggins, 1995] J. K. Huggins. Kermit protocol – formal specification and verification. In E. Börger, eds., *Specification and Validation Methods*, pages 247–293. Oxford University Press, 1995.

[Hunt, 1986] W. Hunt. FM8502: A Verified Microprocessor. Technical Report 47, Institute of Computing Science, University of Texas, Austin, 1986.

[Hutter and Sengler, 1996] D. Hutter and C. Sengler. INKA: the next generation. In [McRobbie and Slaney, 1996], pages 288–292.

[Ibens and Letz, 1996] O. Ibens and R. Letz. Subgoal Alternation in Model Elimination. Technical Report, Institut für Informatik, Technische Universität München, 1996.

[Ibens, 1999] O. Ibens. Connection Tableaux Calculi with Disjunctive Constraints. Dissertation, Institut für Informatik, Technische Universität München. Infix Verlag, 1999.

[Jackson, 1994] D. Jackson. Abstract Model Checking of Infinite Specifications. In [Naftalin *et al.*, 1994], pages 519–531.

[Jacky, 1995] J. Jacky. Specifying a safety-critical control system in Z. *IEEE Transactions on Software Engineering*, 21(2):99–106, 1995.

[Jobmann and Schumann, 1992] M. R. Jobmann and J. Schumann. Modelling and Performance Analysis of a Parallel Theorem Prover. In *Proc. 1992 ACM Sigmetrics and Performance '92*, volume 20(1) of *Performance Evaluation Review*, pages 259–260. ACM Press, 1992.

[Johnson, 1992] W. L. Johnson, ed. *Proc. 7th Knowledge-Based Software Engineering Conference*. IEEE Computer Society Press, 1992.

[Jones and Middelburg, 1994] C. B. Jones and K. Middelburg. A typed logic of partial functions reconstructed classically. *Acta Informatica*, 31(5):399–430, 1994.

[Jones, 1990] C. B. Jones. *Systematic Software Development Using VDM*. Englewood Cliffs, second edition, 1990.

[Jonsson *et al.*, 1991] B. Jonsson, J. Parrow, and B. Pehrson, eds. *Protocol Specification, Testing and Verification*. IFIP, North-Holland, 1991.

[Kappelman *et al.*, 1998] L. Kappelman, D. Fent, K. Keeling, and V. Prybutok. Calculating the cost of year-2000 compliance. *Communications of the ACM*, 41(2):30–39, 1998.

[Kapur, 1992] D. Kapur, ed. *Proc. 11th International Conference Automated Deduction (CADE-11)*, volume 607 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag, 1992.

[Kapur and Zhang, 1989] D. Kapur and H. Zhang. RRL: Rewrite Rule Laboratory User's Manual. Technical Report 89-03, Department of Computer Science, The University of Iowa, 1989.

[Kalman, 2001] J. A. Kalman. Automated Reasoning with OTTER. Rinton Press, 2001.

[Kaufmann, 1998] M. Kaufmann. ACL 2 support for verification projects. Invited talk. In [Kirchner and Kirchner, 1998], pages 220–238.

[KBSE-8, 1993] *Proceedings of the 8th Knowledge-Based Software Engineering Conference*. IEEE Computer Society Press, 1993.

[KBSE-9, 1994] *Proceedings of the 9th Knowledge-Based Software Engineering Conference*. IEEE Computer Society Press, 1994.

[KBSE-10, 1995] *Proceedings of the 10th Knowledge-Based Software Engineering Conference*. IEEE Computer Society Press, 1995.

[KBSE-11, 1996] *Proceedings of the 11th Knowledge-Based Software Engineering Conference.* IEEE Computer Society Press, 1996.

[Kelly, 1997] J. Kelly. *Formal Methods Specification and Analysis Guidebook for the Verification of Software and Computer Systems. Volume II: A Practitioner's Guide.* NASA, 1997. `http://techreports.jpl.nasa.gov/1997/97-0902.pdf`.

[Kessler and Wedel, 1994] V. Kessler and G. Wedel. AUTLOG – An Advanced Logic of Authentication. In *Proc. IEEE Computer Security Foundations Workshop IV*, pages 90–99. IEEE, 1994.

[Kindred and Wing, 1996] D. Kindred and J. Wing. Fast, automatic checking of security protocols. In *2nd USENIX Workshop on Electronic Commerce*, pages 41–52, 1996.

[Kirchner and Kirchner, 1998] C. Kirchner and H. Kirchner, eds., *Proc. 15th International Conference on Automated Deduction (CADE-15).* Volume 1421 of *Lecture Notes in Artificial Intelligence*, Springer-Verlag, 1998.

[Klarlund and Møller, 1998] N. Klarlund and A. Møller. Mona Version 1.2. User Manual. Brics Technical Report, BRICS, University of Aarhus, Denmark, 1998.

[Kneale and Kneale, 1984] W. Kneale and M. Kneale. *The Development of Logic.* Clarendon Press, Oxford, 1984.

[Knight *et al.*, 1997] J. Knight, C. DeJong, M. Gibble, and L. Nakano. Why are formal methods not used more widely? In *Proceedings LFM '97.* NASA, 1997. `http://techreports.larc.nasa.gov/ltrs/refer/papers/NASA-97-cp3356`.

[Knuth and Bendix, 1970] D. E. Knuth and P. E. Bendix. Simple word problems in universal algebra. In J. Leech, ed., *Computational Problems in Abstract Algebra*, pages 263–297. Pergamon Press, 1970.

[Korf, 1985] R. E. Korf. Depth–first Iterative Deepening: an Optimal Admissible Tree Search. *Artificial Intelligence*, 27:97–109, 1985.

[Kreitz *et al.*, 1998] C. Kreitz, M. Hayden, and J. Hickey. A proof environment for the development of group communication systems. In [Kirchner and Kirchner, 1998], pages 316–332.

[Kröger, 1987] F. Kröger. *Temporal Logic of Programs*, volume 8 of *EATCS Monographs on Theoretical Computer Science.* Springer-Verlag, 1987.

[Kurfeß, 1990] F. Kurfeß. *Parallelism in Logic – Its Potential for Performance and Program Development.* Dissertation, Institut für Informatik, Technische Universität München, 1990.

[Letz and Schumann, 1988] R. Letz and J. Schumann. Global Variables in Logic Programming. Technical Report FKI-96-b-88, Technische Universität München, 1988.

[Letz *et al.*, 1988] R. Letz, S. Bayerl, and J. Schumann. SETHEO, A SEquential THEOrem prover for first-order logic. In *ESPRIT Technical Week, Brussels*, 1988.

[Letz *et al.*, 1992] R. Letz, J. Schumann, S. Bayerl, and W. Bibel. SETHEO: A High-Performance Theorem Prover. *Journal of Automated Reasoning (JAR)*, 8(2):183–212, 1992.

[Letz *et al.*, 1994] R. Letz, K. Mayr, and C. Goller. Controlled Integration of the Cut Rule into Connection Tableau Calculi. *Journal of Automated Reasoning (JAR)*, 13(3):297–337, 1994.

[Letz *et al.*, 1998] R. Letz, G. Stenz, and A. Wolf. p-SETHEO: Strategy Parallel Automated Theorem Proving. In Marian Bubak, ed., *Proceedings of the Conference on High Performance Computing on Hewlett-Packard Systems (HiPer-98).* Eidgenössische Technische Hochschule Zürich, 1998.

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

[Lloyd, 1987] J. W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, second edition (first corr. reprint 1993), 1987.

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

[Lowry *et al.*, 1994a] M. Lowry, A. Philpot, T. Pressburger, and I. Underwood. AMPHION: Automatic Programming for Scientific Subroutine Libraries. In Z. W. Raś, M. Zemankova, eds., *Methodologies for Intelligent Systems, Proceedings of the 8th International Symposium on Methodology for Intelligent Systems, ISMIS '94*, volume 869 of *Lecture Notes in Artificial Intelligence*, pages 326–335. Springer-Verlag, 1994.

[Lowry *et al.*, 1994b] M. Lowry, A. Philpot, T. Pressburger, and I. Underwood. A Formal Approach to Domain-Oriented Software Design Environments. In [KBSE-9, 1994], pages 48–57.

[Lowry *et al.*, 1997] M. Lowry, K. Havelund, and J. Penix. Verification and validation of AI systems that control deep-space spacecraft. In Z. W. Raś and A. Skowron, eds., *Proceedings of the 10th International Symposium on Foundations of Intelligent Systems (ISMIS '97)*, volume 1325 of *Lecture Notes in Computer Science*, pages 35–47. Springer-Verlag, 1987.

[Lowry *et al.*, 1998] M. Lowry, S. Zornetzer, A. Gross, and P. Kutler. Advanced software engineering at NASA Ames Research Center. In *Proceedings of the 49th International Aeronautical Conference.* TR number IAF-98-U.3.01, 1998.

[Łukasiewicz, 1970] J. Łukasiewicz. *Selected Works.* North-Holland, 1970.

[Lusk and Overbeek, 1988] E. Lusk and R. Overbeek, ed. *Proc. 9th International Conference Automated Deduction (CADE-9)*, volume 310 of *Lecture Notes in Computer Science.* Springer-Verlag, 1988.

[Maarek and Smadja, 1989] Y. S. Maarek and F. A. Smadja. Full text indexing based on lexical relations an application: Software libraries. In *Proc. 12th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 198–206, 1989.

[Maarek *et al.*, 1991] Y. S. Maarek, D. M. Berry, and G. E. Kaiser. An information retrieval approach for automatically constructing software libraries. *IEEE Transactions on Software Engineering*, SE-17(8):800–813, 1991.

[MacKenzie, 1995] D. MacKenzie. The automation of proof: A historical and sociological exploration. *IEEE Annals of the History of Computing*, 17(3):7–29, 1995.

[Manthey and Bry, 1988] R. Manthey and F. Bry. SATCHMO: a Theorem Prover Implemented in Prolog. In [Lusk and Overbeek, 1988], pages 415–434.

[Martelli and Montanari, 1982] A. Martelli and U. Montanari. An efficient unification algorithm. *ACM Transactions on Programming Languages and Systems*, 4(2):258–282, 1982.

[McCune and Wos, 1991] W. W. McCune and L. Wos. Experiments in Automated Deduction with Condensed Detachment. Technical Report, Argonne National Laboratory, 1991.

[McCune, 1993] W. W. McCune. Single axioms for groups and abelian groups with various operations. *Journal of Automated Reasoning (JAR)*, 10(1):1–13, 1993.

[McCune, 1994a] W. W. McCune. OTTER 3.0 Reference Manual and Guide. Technical Report ANL-94/6, Argonne National Laboratory, 1994.

[McCune, 1994b] W. W. McCune. A Davis-Putnam Program and Its Application to Finite First-Order Model Search: Quasigroup Existence Problems. Technical Report, Argonne National Laboratory, 1994.

[McCune, 1997] W. W. McCune, ed. *Proc. 14th International Conference Automated Deduction (CADE-14)*, volume 1249 of *Lecture Notes in Artificial Intelligence.* Springer-Verlag, 1997.

[McMillan, 1993] K. McMillan. *Symbolic Model Checking: An Approach to the State Explosion Problem.* Kluwer, 1993.

[McRobbie and Slaney, 1996] M. A. McRobbie and J. K. Slaney, eds. *Proc. 13th International Conference on Automated Deduction (CADE 13)*, volume 1104 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag, 1996.

[Meadows, 1995] C. A. Meadows. Formal Verification of Cryptographic Protocols: A Survey. In J. Pieprzyk and R. Safavi-Naini, eds., *Proceedings of the 4th International Conference on the Theory and Applications of Cryptology (ASIACRYPT '94)*, volume 917 of *Lecture Notes in Computer Science*, pages 135–150. Springer-Verlag, 1995.

[Mellish, 1988] C. S. Mellish. Implementing systemic classification by unification. *Journal of Computational Linguistics*, 14(1):40–51, 1988.

[Mili *et al.*, 1998] A. Mili, R. Mili, and R. Mittermeir. A survey of software reuse libraries. *Annals of Software Engineering*, 5:349–414, 1998.

[Millen *et al.*, 1987] J. Millen, S. Clark, and S. Freedman. The interrogator protocol security analysis. *IEEE Software Engineering*, SE-13(2), 1987.

[Miller and Srivas, 1995] S. Miller and M. Srivas. Formal verification of the AAM-P5 microprocessor: a case study in the industrial use of formal methods. In *WIFT'95: Workshop on Industrial-Strength Formal Specification Techniques*. IEEE Press, 1995.

[Moller, 1992] F. Moller. *The Edinburgh Concurrency Workbench (Version 6.1)*. Department of Computer Science, University of Edinburgh, 1992.

[Moormans and Belville, 1990] M. W. Moormans and S. Belville. Getting started on Athena. Technical Report Revision D, MIT, Cambridge, Massachusetts, 1990.

[Moser *et al.*, 1997] 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 (JAR)*, 18:237–246, 1997.

[Moser, 1996] M. Moser. *Goal-Directed Reasoning in Clausal Logic with Equality*. Dissertation, Institut für Informatik, Technische Universität München, 1996.

[Naftalin *et al.*, 1994] M. Naftalin, T. Denvir, and M. Bertran, eds., *FME '94: Industrial Benefit of Formal Methods, Proceedings of the 2nd International Symposium Formal Methods Europe*, volume 873 of *Lecture Notes in Computer Science*. Springer-Verlag, 1994.

[Necula and Lee, 1998] G. C. Necula and P. Lee. Efficient Representation and Validation of Proofs. In *Proceedings of the 13th Annual Symposium on Logic in Computer Science (LICS '98)*, pages 93–104. IEEE Press, 1998.

[Neumann, 1995] P. G. Neumann. *Computer Related Risks*. ACM Press, 1995.

[Newborn, 2001] M. Newborn. *Automated Theorem Proving: Theory and Practice*. Springer Verlag, 2001.

[Nonnengart *et al.*, 1998] A. Nonnengart, G. Rock, and C. Weidenbach. On Generating Small Clause Normal Forms. In [Kirchner and Kirchner, 1998], pages 397–411.

[Ohlbach, 1988] H. J. Ohlbach. A resolution calculus for modal logics. In [Lusk and Overbeek, 1988], pages 500–516.

[Ohlbach, 1993] H. J. Ohlbach. Optimized translation for multi-modal logic into predicate logic. In A. Voronkov, ed., *Proceedings of the 4th International Conference on Logic Programming and Automated Reasoning (LPAR '93)*, volume 698 of *Lecture Notes in Artificial Intelligence*, pages 253–264. Springer-Verlag, 1993.

[Ohlbach, 1998] H. J. Ohlbach. Combining Hilbert style and semantic reasoning in a resolution framework. In [Kirchner and Kirchner, 1998], pages 205–219.

[Parnas, 1994] D. Parnas. Inspection of safety-critical software using program-function tables. In *Proceedings of the 13th IFIP World Computer Congress*, volume A–13, pages 270–277. IFIP, 1994.

[Paulson, 1994] L. C. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *Lecture Notes in Computer Science*. Springer-Verlag, 1994.

[Paulson, 1997a] L. C. Paulson. Proving properties of security protocols by induction. In *PCSFW: Proceedings of the 10th Computer Security Foundations Workshop*. IEEE Press, 1997.

[Paulson, 1997b] L. C. Paulson. Mechanized Proofs of Security Protocols: Needham-Schroeder with Public Keys. Technical Report 413, University of Cambridge, Computer Laboratory, 1997.

[Pelletier and Rudnicki, 1986] F. J. Pelletier and P. Rudnicki. Non-Obviousness. *Association for Automated Reasoning Newsletter*, (6):4–5, 1986.

[Penix and Alexander, 1997] J. Penix and P. Alexander. Toward automated component adaptation. In Natalia Juristo, ed., *Proc. 9th International Conference on Software Engineering and Knowledge Engineering*, pages 535–542. Knowledge Systems Institute, 1997.

[Pfenning, 1988] F. Pfenning. Single Axioms in the Implicational Propositional Calculus. In [Lusk and Overbeek, 1988], pages 710–713.

[Pnueli, 1986] A. Pnueli. Applications of temporal logic to the specification and verification of reactive systems. In J. W. de Bakker, ed., *Current Trends in Concurrency*, volume 224 of *Lecture Notes in Computer Science*, pages 510–584. Springer-Verlag, 1986.

[Prawitz *et al.*, 1960] D. Prawitz, H. Prawitz, and N. Voghera. A mechanical proof procedure and its realization in an electronic computer. *Journal of the ACM*, 7:102–128, 1960.

[Prehn and Toetenel, 1991] S. Prehn and W. J. Toetenel, eds., *VDM 91: Formal Software Development Methods, Proceedings of the 4th International Symposium of VDM Europe*, volume 552 of *Lecture Notes in Computer Science*, Springer-Verlag, 1991.

[Pressman, 1997] R. Pressman. *Software Engineering - a Practitioner's Approach*. McGraw-Hill, 1997.

[Pressburger and Lowry, 1995] T. Pressburger and M. Lowry. Automatic Domain-oriented Software Design Using Formal Methods. In *Proceedings of Software Systems in Engineering Energy Sources*, pages 33–42, 1995.

[Pressburger, 1929] M. Pressburger. Über die Vollständigkeit eines gewissen Systems der Arithmetik ganzer Zahlen, in welchem die Addition als einzige Operation hervortritt. *Sprawozdanie z I Kongresu Matematikow Krajow Slowcanskich Warszawa*, pages 92–101, 1929.

[Protzen, 1992] M. Protzen. Disproving conjectures. In [Kapur, 1992], pages 340–354.

[Rajan *et al.*, 1995] S. Rajan, N. Shankar, and M. K. Srivas. An Integration of Model-Checking with Automated Proof Checking. In P. Wolper, *Proceedings of the 7th International Conference on Computer-Aided Verification (CAV '95)*, volume 939 of *Lecture Notes in Computer Science*, pages 84–97. Springer-Verlag, 1995.

[Rath, 1992] T. Rath. Datenbankunifikation. Report Aida-92-09, Technische Hochschule Darmstadt, 1992 (in German).

[Regensburger, 1994] F. Regensburger. *HOLCF: Eine konservative Erweiterung von HOL um LCF*. Dissertation, Technische Universität München, 1994 (in German).

[Reif and Schellhorn, 1998] W. Reif and G. Schellhorn. *Theorem Proving in Large Theories*. In [Bibel and Schmitt, 1998], chapter III.9, pages 225–242.

[Reif *et al.*, 1997] W. Reif, G. Schellhorn, and K. Stenzel. Proving system correctness with KIV 3.0. In [McCune, 1997], pages 69–72.

[Reif *et al.*, 1998] W. Reif, G. Schellhorn, K. Stenzel, and M. Balser. *Structured Specifications and Interactive Proofs with KIV.* In [Bibel and Schmitt, 1998], chapter II.1, pages 13–40.

[Reif, 1998] W. Reif. Correct Software for Safety-Critical Systems. Invited talk, SPPD meeting during CADE-15, 1998.

[Robinson, 1965] J. A. Robinson. A Machine-oriented Logic Based on the Resolution Principle. *Journal of the ACM*, 12:23–41, 1965.

[Robinson, 1979] J. A. Robinson. *Logic: Form and Function.* North-Holland, 1979.

[Rushby, 1993] J. Rushby. Formal Methods and Digital Systems Validation for Airborne Systems. Technical Report CSL-93-7, SRI International, Menlo Park, CA, 1993.

[Scheler and Schumann, 1995] G. Scheler and J. Schumann. A hybrid model of semantic inference. In A. Monaghan, ed., *Proceedings of the 4th International Conference on Cognitive Science in Natural Language Processing (CSNLP 95)*, 1995.

[Schellhorn and Arendt, 1998] G. Schellhorn and W. Arendt. *The WAM Case Study: Verifying Compiler Correctness for PROLOG with KIV*, In [Bibel and Schmitt, 1998], chapter III.7, pages 165–195.

[Schulz, 1999] S. Schulz. System Abstract: E 0.3. In [Ganzinger, 1999], pages 297–301.

[Schumann and Breitling, 1998] J. Schumann and M. Breitling. Formalisierung und Beweis einer Verfeinerung aus AutoFocus mit automatischen Theorembeweisern, Fallstudie. SFB Bericht SFB342/7/98A, Technische Universität München, 1998 (in German).

[Schumann and Fischer, 1997] J. Schumann and B. Fischer. NORA/HAMMR: Making Deduction Based Component Retrieval Practical. In *Proceedings 12th Conference on Automated Software Engineering (ASE)*, pages 246–254. IEEE Press, 1997.

[Schumann and Ibens, 1998] J. Schumann and O. Ibens. SETHEO V3.5: Users Manual. AR-Report, Technische Universität München, 1998. Available as: `http://ase.arc.nasa.gov/schumann/SETHEO/setheo-man.html`.

[Schumann and Jobmann, 1994] J. Schumann and M. Jobmann. Analysing the Load Balancing Scheme of a Parallel System on Multiprocessors. In *Proceedings PARLE '94*, number 817 in *Lecture Notes in Computer Science*, pages 819–822. Springer-Verlag, 1994.

[Schumann and Letz, 1990] J. Schumann and R. Letz. PARTHEO: A High-Performance Parallel Theorem Prover. In [Stickel, 1990], pages 40–56.

[Schumann *et al.*, 1998] J. Schumann, C. Suttner, and A. Wolf. *Parallel Theorem Provers Based on* SETHEO. In [Bibel and Schmitt, 1998], chapter II.7, pages 261–290.

[Schumann, 1991] J. Schumann. *Efficient Theorem Provers Based on an Abstract Machine.* Dissertation, Technische Universität München, 1991.

[Schumann, 1994a] J. Schumann. DELTA – A Bottom-up Preprocessor for Top-Down Theorem Provers — System Abstract. In [Bundy, 1994], pages 774–777.

[Schumann, 1994b] J. Schumann. Tableau-based Theorem Provers: Systems and Implementations. *Journal of Automated Reasoning (JAR)*, 13(3):409–421, 1994.

[Schumann, 1995] J. Schumann. Using the Theorem Prover SETHEO for Verifying the Development of a Communication Protocol in Focus — A Case Study. In P. Baumgartner, R. Hähnle, and J. Posegga, eds., *Theorem Proving with Analytic Tableaux and Related Methods, Proceedings of the 4th International Workshop, TABLEAUX '95*, volume 918 of *Lecture Notes in Artificial Intelligence*, pages 338–352. Springer-Verlag, 1995.

[Schumann, 1996a] J. Schumann. SiCoTHEO – Simple Competitive Parallel Theorem Provers Based on SETHEO. In H. Kitano and C. Suttner, eds., *Parallel Processing in AI.* Elsevier, pages 9-44, 1996.

[Schumann, 1996c] J. Schumann. SiCoTHEO: Simple Competitive Parallel Theorem Provers. In [McRobbie and Slaney, 1996], pages 240–244.

[Schumann, 1998] J. Schumann. *Automated Deduction in Software Engineering and Hardware Design.* In [Bibel and Schmitt, 1998], volume III.2, pages 99–103.

[Schumann, 1998] J. Schumann. Automatische Verifikation von Authentifikationsprotokollen. *KI – Künstliche Intelligenz*, 4:48–53, 1998 (in German).

[Schütz and Geisler, 1996] H. Schütz and T. Geisler. Efficient Model Generation Through Compilation. In [McRobbie and Slaney, 1996], pages 433-447. Also as Research Report PMS-FB-1996-2, University of Munich (LMU), 1996.

[Selfridge *et al.*, 1991] P. G. Selfridge, D. White, and L. Hoebel, eds. *Proceedings of the 6th Knowledge-Based Software Engineering Conference.* IEEE Computer Society Press, 1991.

[Semle, 1989] H. Semle. Erweiterung einer abstrakten Maschine für ordnungssortiertes PROLOG um die Behandlung polymorpher Sorten. Technical Report, IBM, Stuttgart, 1989 (in German).

[Shoenfield, 1967] J. Shoenfield. *Mathematical Logic.* Addison-Wesley, 1967.

[Shostak, 1976] R. Shostak. Refutation Graphs. *Artificial Intelligence*, 7:51–64, 1976.

[Slaney *et al.*, 1995] J. Slaney, M. Fujita, and M. Stickel. Automated Reasoning and Exhaustive Search: Quasigroup Existence Problems. *Computers and Mathematics with Applications*, 29:115–132, 1994.

[Slaney, 1992] J. Slaney. FINDER, Finite Domain Enumerator: Version 2.0 Notes and Guide. Technical Report TR-ARP-1/92, Australian National University, 1992.

[Slaney, 1994] J. Slaney. FINDER: Finite domain enumerator. In [Bundy, 1994], pages 798–801.

[Slind *et al.*, 1998] K. Slind, M. Gordon, B. Boulton, and A. Bundy. System description: An interface between Clam and HOL. In [Kirchner and Kirchner, 1998], pages 134–138.

[Smith, 1990] D. R. Smith. KIDS: A Semiautomatic Program Development System. *IEEE Transactions on Software Engineering*, 16(9):1024–1043, 1990.

[Smith, 1998] D. Smith. Deductive Support for Software Development. Invited talk, SPPD meeting during CADE-15, 1998.

[Der Spiegel, 1998] Aussichten eines Klons. *Der Spiegel*, 18, 1998 (in German).

[Spivey, 1988] J. M. Spivey. *Understanding Z.* Cambridge Tracts in Theoretical Computer Science 3. Cambridge University Press, 1988.

[Stenning, 1976] V. Stenning. A Data Transfer Protocol. *Computer Networks*, 1:98–110, 1976.

[Stevenson, 1990] D. E. Stevenson. 1001 Reasons For Not Proving Programs Correct: A Survey. *Philosophy and Computers*, 1990.

[Stickel *et al.*, 1994] M. Stickel, R. Waldinger, M. Lowry, T. Pressburger, and I. Underwood. Deductive Composition of Astronomical Software from Subroutine Libraries. In [Bundy, 1994], pages 341–355.

[Stickel, 1988] M. E. Stickel. A Prolog Technology Theorem Prover: Implementation by an Extended Prolog Compiler. *Journal of Automated Reasoning (JAR)*, 4:353–380, 1988.

[Stickel, 1989] M. E. Stickel. A PROLOG Technology Theorem Prover: A New Exposition and Implementation in PROLOG. Stanford, 1989.

[Stickel, 1990] M. E. Stickel, ed. *Proc. 10th International Conference on Automated Deduction (CADE-10)*, volume 449 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag, 1990.

[Stickel, 1991] M. E. Stickel. Upside-Down Meta-Interpretation of the Model Elimination Theorem Proving Procedure for Deduction and Abduction. ICOT Technical Report, 1991.

[Stolzenburg and Thomas, 1998] F. Stolzenburg and B. Thomas. *Analyzing Rule Sets for the Calculation of Banking Fees by a Theorem Prover with Constraints*. In [Bibel and Schmitt, 1998], chapter III.10, pages 243–264.

[Storey, 1996] N. Storey. *Safety-critical Computer Systems*. Addison-Wesley, 1996.

[Sutcliffe and Suttner, 1997] G. Sutcliffe and C. Suttner, eds. The CADE-13 Automated Theorem Proving System Competition. *Journal of Automated Reasoning (JAR)*, special issue, 18(2), 1997.

[Sutcliffe *et al.*, 1994] G. Sutcliffe, C. B. Suttner, and T. Yemenis. The TPTP Problem Library. In [Bundy, 1994], pages 252–266.

[Suttner and Schumann, 1993] C. Suttner and J. Schumann. Parallel Automated Theorem Proving. In L. Kanal, V. Kumar, H. Kitano, and C. Suttner, eds., *Parallel Processing for Artificial Intelligence I*, pages 209–257. Elsevier, 1993.

[Suttner and Sutcliffe, 1998] C. Suttner and G. Sutcliffe. The CADE-14 Automated Theorem Proving System Competition. *Journal of Automated Reasoning* (JAR), 21(1):99–134, 1998.

[Suttner and Sutcliffe, 1999] G. Sutcliffe and C. Suttner. The CADE-15 ATP System Competition. *Journal of Automated Reasoning* (JAR), 23(1):1–23, 1999.

[Suttner, 1995] C. Suttner. *Parallelization of Search-based Systems by Static Partitioning with Slackness*. Dissertation, Technische Universität München. Volume 101 of *DISKI*. Infix-Verlag, 1995.

[Syverson and van Oorschot, 1994] P. F. Syverson and P. van Oorschot. On Unifying Some Cryptographic Protocol Logics. In *Proceedings of the IEEE Symposium on Research in Security and Privacy*, pages 14–28. IEEE, 1994.

[Tammet, 1997] T. Tammet. Gandalf. *Journal of Automated Reasoning (JAR)*, 18(2):199–204, 1997.

[Taylor, 1990] D. S. Taylor. A Beginner's Strategy Guide to the Larch Prover. S. B. Thesis, Department of Electrical Engineering and Computer Science, MIT, 1990.

[URL Amphion, 2000] `http://ase.arc.nasa.gov/`.

[URL ARS-Repository, 2000] `http://www-formal.stanford.edu/clt/ARS/systems.html`.

[URL Autofocus, 2000] `http://autofocus.in.tum.de`.

[URL Formal Methods Archive, 2000] URL: `http://archive.comlab.ox.ac.uk/formal-methods.html`.

[URL Isabelle, 2000] URL: `http://www.in.tum.de/~isabelle`.

[URL MSPASS, 2000] `http://www.cs.man.ac.uk/ schmidt/mspass/`.

[URL SCAN Algorithm, 2000] `http://www.mpi-sb.mpg.de/~ohlbach/scan/index.html`.

[URL SETHEO, 2000] `http://wwwjessen.in.tum.de/~setheo/`.

[URL Secure Computing, 2000] `http://www.securecomputing.com` go to *Security Libary*.

[URL SNARK, 2000] `http://www.ai.sri.com/~stickel/snark.html`.

[van Hentenryck, 1989] P. van Hentenryck. *Constraint Satisfaction in Logic Programming*. MIT Press, 1989.

[Veroff and McCune, 2000] R. Veroff and W. W. McCune. A Short Sheffer Axiom for Boolean Algebra. Technical Report, Argonne National Laboratory, 1994. `http://www-unix.mcs.anl.gov/~mccune/ba-sheffer-15/`.

[Visser *et al.*, 2000] W. Visser, K. Havelund, G. Brat, and S. Park. Model Checking Programs. In *Proceedings 15th Conference on Automated Software Engineering (ASE)*, pages 3–12. IEEE Press, 2000.

[Wagner, 1997] K. Wagner. PIL: Ein SETHEO-basiertes Werkzeug zur Analyse kryptographischer Protokolle. Fortgeschrittenenpraktikum, Technische Universität München, 1997 (in German).

[Walther, 1988] Ch. Walther. Many-Sorted Unification. *Journal of the ACM*, 35(1):1–17, 1988.

[Wang, 1960] H. Wang. Proving theorems by pattern recognition. *Communications of the ACM*, 4(3):229–243, 1960.

[Wang and Goldberg, 1994] T.C. Wang and A. Goldberg. KITP-93: An Automated Inference System for Program Analysis. In [Bundy, 1994], pages 831–835.

[Warren, 1983] D. H. D. Warren. An Abstract PROLOG Instruction Set. Technical Report, SRI, Menlo Park, Ca, USA, 1983.

[Weber-Wulff, 1993] D. Weber-Wulff. Selling formal methods to industry. In [Woodcock and Larsen, 1993], pages 671–678.

[Weidenbach *et al.*, 1996] C. Weidenbach, B. Gaede, and G. Rock. Spass and Flotter Version 0.42. In [McRobbie and Slaney, 1996], pages 141–145.

[Welty *et al.*, 1997] C. Welty, M. Lowry, and Y. Ledru, eds., *Proc. 12th International Conference Automated Software Engineering (ASE 1997)*. IEEE Press, 1997.

[Williams, 1994] R. N. Williams. Data Integrity With Veracity. Technical Report, 1994. Available as `ftp://ftp.rocksoft.com//papers/vercty10.ps`.

[Wolf and Kmoch, 1997] A. Wolf and A. Kmoch. Einsatz eines automatischen Theorembeweisers in einer taktikgesteuerten Beweisumgebung an einem Beispiel aus der Hardware-Verifikation. SFB Bericht SFB342/20/97A, Technische Universität München, 1997 (in German).

[Wolf and Letz, 1998] A. Wolf and R. Letz. Strategy Parallelism in Automated Theorem Proving. In Diane J. Cook, ed., *Proceedings of the 11h International Florida AI Research Society (FLAIRS-11) Conference*, pages 142–146. AAAI Press, 1998.

[Wolf and Letz, 1999] A. Wolf and R. Letz. Strategy Parallelism in Automated Theorem Proving. *International Journal for Pattern Recognition and Artificial Intelligence (IJPRAI)*, 13(2):219–45, 1999.

[Wolf and Schumann, 1997] A. Wolf and J. Schumann. ILF-SETHEO: Processing Model Elimination Proofs for Natural Language Output. In [McCune, 1997], pages 61–65.

[Wolf, 1997] A. Wolf. A Translation of Model Elimination Proofs into a Structured Natural Deduction. In D. Dankel II, ed., *Proceedings of the 10h International Florida AI Research Society Conference (FLAIRS-10)*, pages 11–15. AAAI Press, 1997.

[Wolf, 1998] A. Wolf. p-SETHEO: Strategy Parallelism in Automated Theorem Proving. In H. de Swart, ed., *Automated Reasoning with Analytic Tableaux and Related Methods, Proceedings of the International Conference, TABLEAUX '98*, volume 1397 in *Lecture Notes in Artificial Intelligence*, pages 320–324. Springer-Verlag, 1998.

[Wolf, 1999] A. Wolf. *Paralleles Theorembeweisen: Leistungssteigerung, Kooperation und Beweistransformation.* Dissertation, Institut für Informatik, Technische Universität München, 1999 (in German).

[Wolfram, 1991] S. Wolfram. *Mathematica – A System for Doing Mathematics by Computer.* Addison–Wesley, 2nd edition, 1991.

[Woodcock and Larsen, 1993] J. C. P. Woodcock and P. G. Larsen, eds. *FME '93: Industrial-Strength Formal Methods*, volume 670 of *Lecture Notes in Artificial Intelligence.* Springer-Verlag, 1993.

[Wordsworth, 1992] J. B. Wordsworth. *Software Development with Z.* Addison-Wesley, 1992.

[Wos and McCune, 1988] L. Wos and W. W. McCune. Challenge Problems Focusing on Equality and Combinatory Logic: Evaluating Automated Theorem-Proving Programs. In [Lusk and Overbeek, 1988], pages 714–729.

[Wos, 1988] L. Wos. *Automated Reasoning: 33 Basic Research Problems.* Prentice Hall, 1988.

[Wos, 1992] L. Wos. *Automated Reasoning: Introduction and Applications.* McGraw Hill, 2nd edition, 1992.

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

Index