Fachbereich Informatik
Universität Kaiserslautern
Postfach 3049
D-6750 Kaiserslautern 1, W Germany

SEKI-PROJEKT

SEKI MEMO

THE MARKGRAF KARL REFUTATION PROCEDURE

KARL MARK G RAPH

Memo-SEKI-MK-84-01

THE MARKGRAF KARL REFUTATION PROCEDURE

KARL MARK G RAPH

Universität Kaiserslautern
Fachbereich Informatik
Postfach 3049
D-6750 Kaiserslautern
WEST GERMANY

Universität Karlsruhe
Institut für Informatik I
Postfach 6380
D-7500 Karlsruhe 1
WEST GERMANY

Druck: Abteilung Foto-Repro-Druck der Universität Kaiserslautern

Karl Mark G Raph is the pseudo name of the authors of this jointly written report. They are the memberbs of the MKRP-Project, which currently (January 1984) consists of the following people:

Susanne Biundo, Dipl. Inf.,
(Induction Prover; Karlsruhe)

Karl-Hans Bläsius, Dipl. Math.,
(Equality Reasoning; Kaiserslautern)

Hans-Jürgen Bürckert, Dipl. Math.,
(Unification Theory; Kaiserslautern)

Norbert Eisinger, Dipl. Inf.,
(Theoretical Problems of the Clause Graph Calculus; Kaiserslautern)

Alexander Herold, Dipl. Math.,
(Unification Theory; Kaiserslautern)

Thomas Käufl, Dipl. Inf.,
(Program Verification; Karlsruhe)

Christoph Lingenfelder, Dipl. Phys.,
(Proof transformation; Kaiserslautern)

Hans-Jürgen Ohlbach, Dipl. Phys.,
(Supervisor; Kaiserslautern)

Manfred Schmidt-Schauß, Dipl. Math.,
(Supervisor; Kaiserslautern)

Jörg H. Siekmann, Prof. Dr. (Ph. D.), grad. Ing.,
(Project leader; Kaiserslautern)

Christoph Walther, Dipl. Inf.,
(Induction Prover, Sorted Logic; Karlsruhe)

University of Kaiserslautern University of Karlsruhe

January 1984

6. THE MARKGRAF KARL REFUTATION PROCEDURE

6. THE MARKGRAF KARL REFUTATION PROCUDURE

> Überhaupt hat der Fortschritt das an sich, daß er viel größer ausschaut, als er wirkliche ist
>
> J. N. Nestroy, 1859

6.1. INTRODUCTION

The current state of development of the Markgraf Karl Refutation Procedure (MKRP), a theorem proving system under development since 1977 at the University of Karlsruhe, West Germany, is presented and evaluated in the sequel. The goal of this project can be summarized by the following three claims: it is possible to build a theorem prover (TP) and augment it by appropriate heuristics and domain-specific knowledge such that

(i) it will display an active and directed behaviour in its striving for a proof, rather than the passive combinatorial search through very large search spaces, which was the characteristic behaviour of the TPs of the past. Consequently

(ii) it will not generate a search space of many thousands of irrelevant clauses, but will find a proof with comparatively few redundant derivation steps.

(iii) Such a TP will establish an unprecedented leap in performance over previous TPs expressed in terms of the difficulty of the theorems it can prove.

With about 25 man years invested up to now and a source code of almost 2000 K (bytes of Lispcode), the system represents the largest single software development undertaken in the history of the field and the results obtained thus far corroborate the first two claims.

The final (albeit essential) claim has not been achieved yet: although at present it performs substantially better than most other automatic theorem proving systems, on certain classes of examples (induction, equality) the comparison is unfavourable for

the MRKP-system. But there is little doubt that these shortcomings reflect the present state of development; once the other modules (equality reasoning, a more refined monitoring and induction) are operational, traditional theorem provers will probably no longer be competitive.

This statement is less comforting than it appears: the comparison is based on measures of the search space and it totally neglects the (enormous) resources needed in order to achieve the behaviour described. Within this frame of reference it would be possible to design the "perfect" proof procedure: the supervisor and the look-ahead heuristics would find the proof and then guide the system without any unnecessary steps through the search space.

In summary, although there are good fundamental arguments supporting the hypothesis that the future of TP research is with the finely knowledge engineered systems as proposed here, there is at present no evidence that a traditional TP with its capacity to quickly generate many ten thousands of clauses is not just as capable. The situation is still (at the time of writing) reminiscent of todays chess playing programs, where the programs based on intellectually more interesting principles are outperformed by the brute force systems relying on advances in hardware technology.

The following paragraph summarizes the basic notions and techniques of theorem proving as far as they are relevant here (and may be skipped by a reader already familiar with the field).

Section 6.2 provides an overview of the whole system, including those parts whose development is not completed yet. This section should give a feeling for the general ideas and principles that guided the design of the overall structure of the MKRP system.

Section 6.3 is written from the methodoloigcal point of view and describes the major novel techniques and methods that are used

within the MKR-Producer.
In contrast section 6.4. uses the flow of control as a guiding line to present the actual working of the Logic Machine, the innermost part of the MKRP-system.

Section 6.5 quickly reviews some of the software tools that were developed for the implementation of the system and section 6.6 gives an overview of how to use the system.

Basic Techniques and Terminology

The language used in this report is that of first-order predicate logic with which we assume the reader to be familiar. From the primitive symbols of this logic we use: u,x,y,z as individual variables; a,b,c,d as individual constants; P,Q,R as predicate constants; f,g,h as function letters. The equality predicate will be denoted by E and mostly written in infix notation as = to improve readability. Individual constants and variables are terms as well as n-placed functions applied to n terms. As metasymbols for terms we use r, s and t. The arity of functions and predicates will be clear from the context. An n-place predicate letter applied to n terms is an atom. A literal is an atom or the negation thereof. For literals we use L,K. The absolute value |L| of a literal L is the atom K such that either L is K or L is ~ K.

A clause is a finite set of literals for which the metasymbols C,D are used. A clause is interpreted as the disjunction of its literals, universally quantified (over the entire disjunction) on its individual variables. The empty clause is denoted as □. A ground clause, ground literal or ground term is one that has no variables occurring in it. A substitution δ is a mapping from variables to terms almost identical everywhere. Substitutions are extended to mappings from terms to terms by the usual morphism. Substitutions are also used to map literals (clauses) to literals (clauses) in the obvious way. A substitution is denoted as a set

of pairs $\delta=\{(v_1 \leftarrow t_1) \ldots (v_n \leftarrow t_n)\}$ where the v_i are variables and the t_i are terms. The term $\delta(t)$ (the literal $\delta(L)$, the clause $\delta(C)$) is called an instance of t (an instance of L, an instance of C). We use δ,σ, for substitutions. A substitution σ is called a unifier for two atoms L and K iff $\sigma(L)=\sigma(K)$; σ iscalled a most general unifier (mgu) of L and K, if for any other unifying substitution δ there exists a substitution λ such that $\delta=\lambda\circ\sigma$, where $\circ$ denotes the functional composition of substitutions. A matcher (or one-way unifier) for two literals L and K relative to L is a substitution σ such that $\sigma L = K$.

The Herbrand Universe H(S) of a set S of clauses is the set of all ground terms that can be constructed from the symbols occurring in S (if no individual constant occurs in S we add the single constant symbol c). A Herbrand instance H(t) of a term t is an instance $\delta(t)$, such that all the terms in δ are from H(S); similarly we define a Herbrand instance of an atom, a literal, a clause. An interpretation T of S is a set of ground literals, whose absolute values are all the Herbrand instances of atoms of S such that for each Herbrand instance L of an atom exactly L or ~L is in T. An interpretation T satisfies a ground clause C iff $C \cap T \neq \emptyset$. T satisfies a clause C if it satisfies every ground instance of C in H(C); T satisfies a set of clauses S if it satisfies every clause in S. A model M of a set of clauses S is an interpretation that satisfies S. If S has no model it is unsatisfiable. For the equality predicate $\equiv$ and a set of clauses S, a model M of S is an E-model if

(i) $t \equiv t \in M$ for all terms t

(ii) if the literals $L \in M$ and $s \equiv t \in M$ and if L' is obtained from L by replacing an occurence of s in L by t then $L' \in M$.

If S has no E-models then it is E-unsatisfiable.

Two literals are complementary if they are of opposite sign and have the same predicate letter.

If C and D are clauses with no variables in common and L and K

are complementary literals in C and D respectively, and if |L| and |K| are unifiable with most general unifier σ, then R=σ(C-{L})∪σ(D-{K}) is a resolvent of C and of R and each literal L in R descends from a literal L´ in C or D.

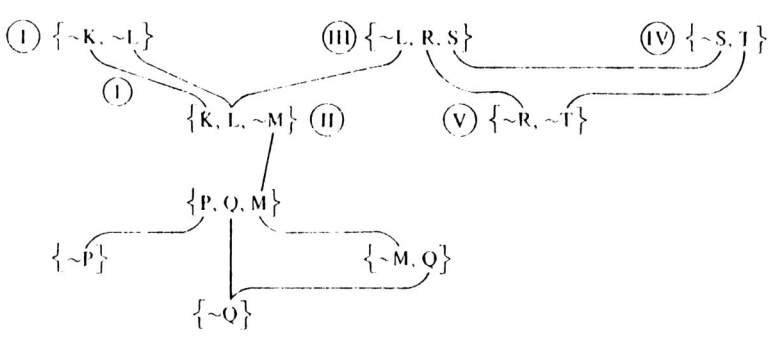
If C is a clause with two literals L and K and if a most general unifier σ exists such that σ(L)=σ(K) then F=σ(C-K) is called a factor of C. If C and D are clauses with no variables in common, and s≡t is a literal in C, and r is a term occurring in D such that there exists σ with σ(s)=σ(r), and D´is obtained from D by replacing r in D by t then P=σ(D´)∪σ(C-{s≡t}) is a paramodulant of C and D. This inference rule is called paramodulation.

A connection graph CG is

(i) a set of clauses S

(ii) a binary relation **R** over literals in S, such that (L,K)ε**R** if |L| and |K| are unifiable and L and K are of opposite sign. Sometimes we write ⟨S⟩ for the connection graph obtained from S.

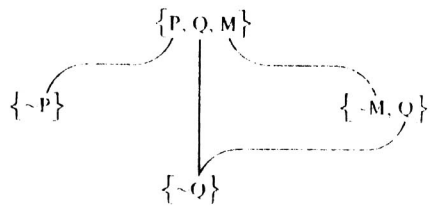
A literal L in S is pure if it does not occur in any of the pairs of **R** i.e. it is not connected and the clause containing L may then be deleted in CG. A connection graph is graphically represented by drawing a link between L and K for every (L,K)ε**R**. L and K are said to be connected. Instead of repeating the definition of the connection graph proof procedure [KO75] we give an example for one derivation step. Consider the following connection graph:



Suppose we want to obtain the resolvent of clause (I) and clause (II), i.e. we want to resolve upon link (1). This is done by adding the resolvent to the graph and by connecting the resolvent in the following way: if a literal L in the resolvent descends from a literal L´ in one of the parent clauses and if L´ was connected to some literal K and if K and L are unifiable, then L and K are connected by a link. Finally the link resolved upon is deleted and all tautologies and all clauses containing pure literals are deleted.

For the connection graph above, resolving upon link (1) leads to a tautology, which is deleted, hence (I) and (II) are deleted since K, ~ K is now pure.

Similarly clauses (III), (IV) and (V) are deleted; i.e. after one step the whole connection graph shrinks to:



This potentially rapid reduction of the original graph causes the practical attraction as well as the theoretical problems of this proof procedure. A more formal presentation of the procedure is contained in Paragraph 6.3.10.

Apart from the deletion of clauses containing pure literals there are additional deletion rules, which become particularily significant in the context of connection graphs: every deletion of a clause may cause further deletions of clauses (and links)

and the resulting complex interplay is still not very well understood theoretically (see e.g. [BI81] [EI81] [SM82]).

A clause C is a tautology if it contains two complementary literals L and K such that $|L|=|K|$ or a literal of the form $t \equiv t$.

A clause C subsumes a clause D if $|C| \leq |D|$ and there exists a substitution δ such that $\delta C \subset D$. (This is the definition of δ-subsumption in [LOV78]).

Subsumed clauses and tautologeous clauses may be deleted from the graph, as discussed in Sec. 6.3.3 and 6.3.4 respectively. The unrestricted use of these delition rules is known to make the respective proof procedure incomplete and even inconsistent.

A traditional refinement restricts the search space by blocking certain possible resolution steps. For example a UNIT refutation, in which at least one of the parent clauses of a resolvent must be a unit clause, is such a refinement.
SET-OF-SUPPORT is also a refinement: the set of clauses is partitioned into two subsets (usually the set of the axiom clauses S and the set of the theorem clauses T) and resolution is only permitted if at least one parent clause is in T. The resolvents are put into T, i.e. the effect of set-of-support is most profitable at the beginning of the search, but it fades the more the deduction proceeds.

A LINEAR refinement selects a top clause from the set of theorem clauses and uses this clause as one of the parents for a resolution step. Then the resolvent becomes the top clause and so on either until the empty clause has been derived or backtracking is necessary.
The development of complete refinements was the main focus of research in theorem proving in the past and there may be close to a hundred now (see e.g. [LOV78] [CHL73]), some of those are used to advantage in the MKR-Procedure as well.

In contrast to a refinement, which only restricts the number of possible steps (and often "cuts off garbage and gold alike"), a strategy gives active advice as to what to do next. The development and integration of such strategies into one system was the main research problem of the MKRP project and the techniques developed so far are presented in the following sections. Strategic information overrides any other information: even if a particular refinement was chosen, the resulting deduction may be very different. Only if nothing better is known, does the system behave like a traditional theorem prover.

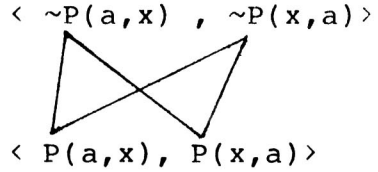
Completeness

The MKR-Procedure is incomplete, yet even worse it is inconsistent as it stands. This is partly so, because the implementation is not completed and partly because there are open theoretical problems in the connection graph procedure itself,see e.g. [BI81] and [SM82]. Most of the cases causing incompleteness (except paramodulation) however are irrelevant for practical examples; quite on the contrary, for some of them it is a hard job to find an example where it is in fact relevant.

In particular there are the following cases:

- As all reductions are performed before factorization, the graph may collapse although the clause set is unsatisfiable (i.e. the system is inconsistent):

  Example:

  All four R=links are tautology links and will be deleted causing purity deletion of both clauses, although the factors <P(a a)> and <NOT P(a a))> would allow for a refutation.

- Tautologies are deleted without any restriction, although this is known to be inconsistent, see [SM82].

- Subsumed clauses and links are deleted without any restriction, which can also cause inconsistency, see [SM82] [EI81].

- Paramodulation and equality reasoning is not fully implemented. Especially the mechanism to generate only those P-links into variables which are necessary for completeness is not yet completed. Unrestricted generation of P-links from each side of an equation into each variable would blow up the graph without significantly increasing the total amount of information. Hence, P-links into variables are not generated so far.

These deficiencies may be the reason that a proof exists, but cannot be found by the system. Even worse, the graph may collapse (usually indicating satisfiability), although the initial clause set is unsatisfiable.

As more theoretical results about clause graph procedures become known [EI83], we hope to eliminate at least the cases causing inconsistency, whereas completeness results, although interesting as they may be from a theoretical point of view, are of course less important for practical purposes.

6.2. OVERVIEW OF THE SYSTEM

The working hypothesis of the MKRP project first formulated in an early proposal in 1975, reflects the then dominating themes of artificial intelligence research, namely that TPs have attained a certain level of performance, which will not be significantly improved by:

(i) developing more and more intricate refinements (like unit preference, linear resolution, TOSS, MTOSS, ...), whose sole purpose is to reduce the search space, nor by

(ii) using different logics (like natural deduction logics, sequence logics, matrix reduction methods etc.)

although this was the main focus of theorem proving research in the past and of course it is not entirely without its merits even today.

The relative weakness of current TP-systems as compared to human performance is due to a large extent to their lack of the rich mathematical and extramathematical knowledge that human methematicians have: in particular, knowledge about the subject and knowledge of how to find proofs in that subject.

To a lesser, but still important extent the relative weakness of current TP-systems can be attributed to the insufficient emphasis which in the past has been laid onto the software engineering problems and - sometimes even minor - design issues that in their combination account more for the strength of a system than any single refinement or "logical improvement".

Hence the object of the MKRP-project is firstly to carefully design and develop a TP system comparative in strength to traditional systems and secondly to augment this system with the appropriate knowledge sources and heuristic methods. As a test case and for the final evaluation of the project's success or

failure, the knowledge of an algebraic treatment of automata theory shall be made explicit and incorporated such that the theorems of a standard textbook [DE71] can be proved mechanically. These proofs are to be transformed into ordinary natural language mathematical proofs, thus generating the first standard textbook entirely written by a machine.

The MKRP system is also heavily used as the deduction component of the program verification project, as described elsewhere in this report.

A Bird's-eye View

Proving a theorem has two distinct aspects: the creative aspect of how to find a proof, usually regarded as a problem of psychology, and secondly the logical aspect as to what constitutes a proof and how to write it down on a sheet of paper, usually referred to as proof theory.

These two aspects are in practice not as totally separated as this statement suggests (see e.g. [SZ69]), however we found it sufficiently important to let it dominate the overall design of the system:

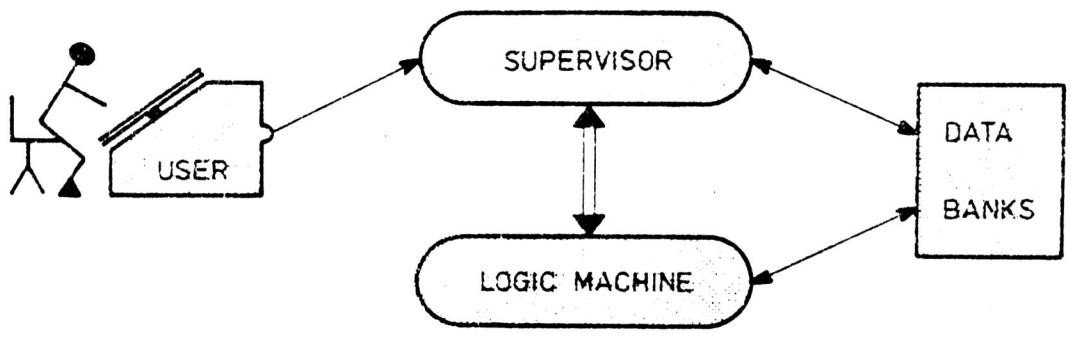


Figure 1

The SUPERVISOR, consists of several independent modules and has the complex task of generating an overall proposal (or several such) as to how the given theorem may best be proved, invoking the necessary knowledge that may be helpful in the course of the search for a proof and finally transforming both proposal and knowledge into technical information sufficient to guide the LOGIC MACHINE through the search space.
The LOGIC MACHINE is at heart a traditional theorem prover based on Kowalski's connection graph proof procedure [KO75], augmented by several components, which select the most appropriate deduction steps to be carried out.

The DATA BANKS consist of the factual knowledge of the particular mathematical field under investigation, i.e. the definitions, axioms, previously proved theorems and lemmata, augmented as far as possible by local knowledge about their potential use.

A view from a lesser Altitude

The diagram of Fig. 2 refines Fig. 1 in order to gain a feeling for the working of the system:

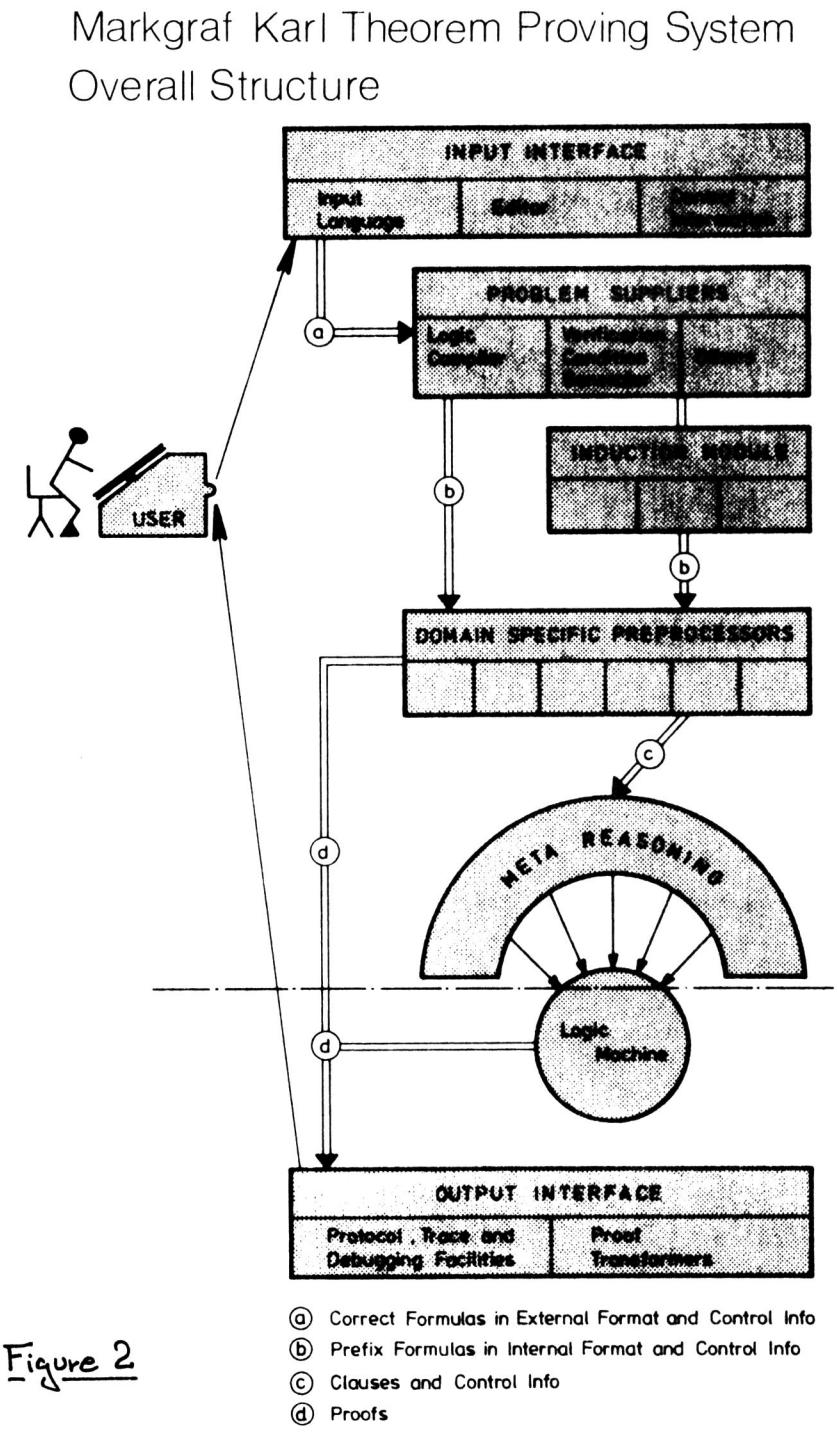


Figure 2

The user communicates with the system via the INPUT and the OUTPUT INTERFACE.

The numerous INPUT facilities essentially have the function to set the parameters and options governing the later behaviour of the system.

Some of the interactive facilities of this level were only designed for the intermediate stages of development and shall be taken over to an increasing degree by the SUPERVISOR as it develops, with the intention to move this interface with the user altogether to the outside and to make the SUPERVISOR take most of the low level decisions. Two sets of instructions however are to stay: the Editor is used to set up (and to read) the DATA BANKS in a way easily intelligible for the user. It also performs a syntactical and semantical analysis of the DATA BANK, which is of considerable practical importance in view of the fact that it eventaully contains a whole standard mathematical textbook.

The second set provides the user with several options to express control information to influence the later search for a proof.

The OUTPUT INTERFACE provides facilities for tracing the behaviour of the system at different degrees of abstraction using the available Protocol Options. The Proof Transformation modules are currently under development and transform a resolution style proof into a natural deduction proof and in a second stage translate this natural deduction proof into a proof stated in natural language.

The PROBLEM SUPPLIERS are the internal interface to the system, which either compile the theorems of the user or feed the output of a verification condition generator into the MKR-procedure. Other devices (e.g. we plan to prove certain hardware configurations and flow diagrams of a factory to be correct) may be plugged into the system at this point of entry.

Depending on the kind of the formulas the system may decide that the theorem is best be proved by structural induction, in which case the INDUCTION MODULE takes over control. This module, which is not yet fully developed, uses the MKR-Procedure in order to prove the base cases as well as the actual induction step and hence whether or not the theorem is to proved by induction, the information in (b) is now handed over to the two main components of the SUPERVISOR, the DOMAIN SPECIFIC PREPROCESSORS and the META REASONING components.

The PREPROCESSORS are domain specific experts that are called for various tasks depending on the kind of input formula.

The PREPROVER, one of these experts, consists of fast special purpose theorem proving techniques (like e.g. the Nelson-Oppen technique [NO77]) and elaborated simplification methods (like e.g. an evaluation of arithmetic terms and the simplification techniques of the King simplifier [KI69]), which are used to advantage for highly redundant or very special input formulas as for example in program verification tasks (see 6.3.12).

A second preprocessor tries to deal with the problem that the MKRP system has mainly been designed for the refutation of comparatively small sets of formulas (say fifty clauses) requiring however deep deductions (say proofs of up to two hundred steps). Although such a situation is generally the case in mathematics, for certain applications like proving verification conditions just the opposite is the case. There are in general rather large sets of formulas to be refuted requiring relatively few deduction steps. In such circumstances the SPLITTING allows to split the initial graph into a set of subgraphs such that if each of these (sometimes many hundreds) subgraphs has been refuted so is the original graph.

Another preprocessor, which is standard in every ATP system based on resolution, is concerned with the syntax of the input formulas

and translates these into clausal form. The elimination of equivalence and implication signs is also optimized [EW83].

Finally the resulting clause set is again simplified as much as possible (see 6.4.1.1) and the clause graph is constructed and again simplified.

These domain specific preprocessors are a main focus of our current developments and additional techniques will be implemented: for example a more elaborated definition expansion expert, an expert to transform special formulas automaticlly into attributes and so on.

The META REASONING components are the second major focus of our current developments, which however are not stable enough yet to be included into this report. The main job of these components is to "taylor" the input formula into a more appropriate format for the LOGIC MACHINE. For example the set of clauses can be reformulated by an extensive use of the sorted logic: instead of expressing certain facts at the clause level, they can be coded to advantage into the sort relationships. Also equalities may be better coded into the unification algorithms, clauses with two literals are often better used as literal rewriting rules and so on. Finally the transformation of the original formulas on the basis of their underlying meaning as well as the activation of additional knowledge (definitions, lemmata and theorems) that may be useful in the search for a proof is to be carried out by these components.

At this point, indicated by the line _._._ in Fig. 2, the SUPERVISOR has essentially finished its task and generated a set of clauses and control information hopefully appropriate for the task at hand: this information can be viewed as a proposal, which is now handed over to the LOGIC MACHINE, the actual deduction component of the MKRP system.

Up to this point the decisions and activities of the SUPERVISOR are to some extent based on the meaning of the theory under investigation and on knowledge about proofs in this theory (i) and its top goal may be formulated as: to be helpful in finding a proof by generating an appropriate clause set and appropriate control information. Once it has done so, the topgoal becomes "to derive a contradiction (the empty clause) and this goal implies that different kinds of information are now useful: the original information based on the semantics (which is by now coded into various parameters, priority values and activation modules) and in addition information based on the syntax of the formula under investigation as well as standard theorem proving techniques.

The achievement of this second goal is the task of the LOGIC MACHINE and hence it most closely corresponds to a traditional theorem prover. But its actual operation is again very different from a traditional system: since a traditional refinement does not specify which literal to resolve upon next, a classical resolution based theorem prover is not guided towards this goal in a step by step fashion. For example linear resolution reduces the search space as compared to binary resolution, but within the remaining space the search is as blind as ever.

(i) The proof techniques of human mathematicians developed for special problems in particular mathematical fields are sometimes usefully known by a machine too, but more often specially developed machine oriented techniques are more advantageous: as it so happend evolution did not provide us with a powerful deduction component built-in and hence a human mathematician can not rely on this useful device and is forced to develop rather different heuristics and techniques for himself.

The LOGIC MACHINE consists of two main components: the Logic Engine, which is an extended clause graph theorem prover based to a large extent on Kowalski's connection graph calculus [KO75] and the Selection Module.

Once the initial connection graphs are set up, the search for a proof commences by the selection of the next "most appropriate" deduction step. This selection process, which constitutes the "heuristic intelligence" of the MKR-Procedure turns the initial representation of clauses (the connection graph) into a proof procedure (based on connection graphs) as it defines a particular selection function mapping graphs to links. This selection function is situated in the Selection Module and is implemented as a production system [OH82]: each "production", which is called an operation block, consists of an activation condition, an update function and an executed function. The currently implemented operation blocks are summarized in Fig. 3 and are described in Sec. 6.4.

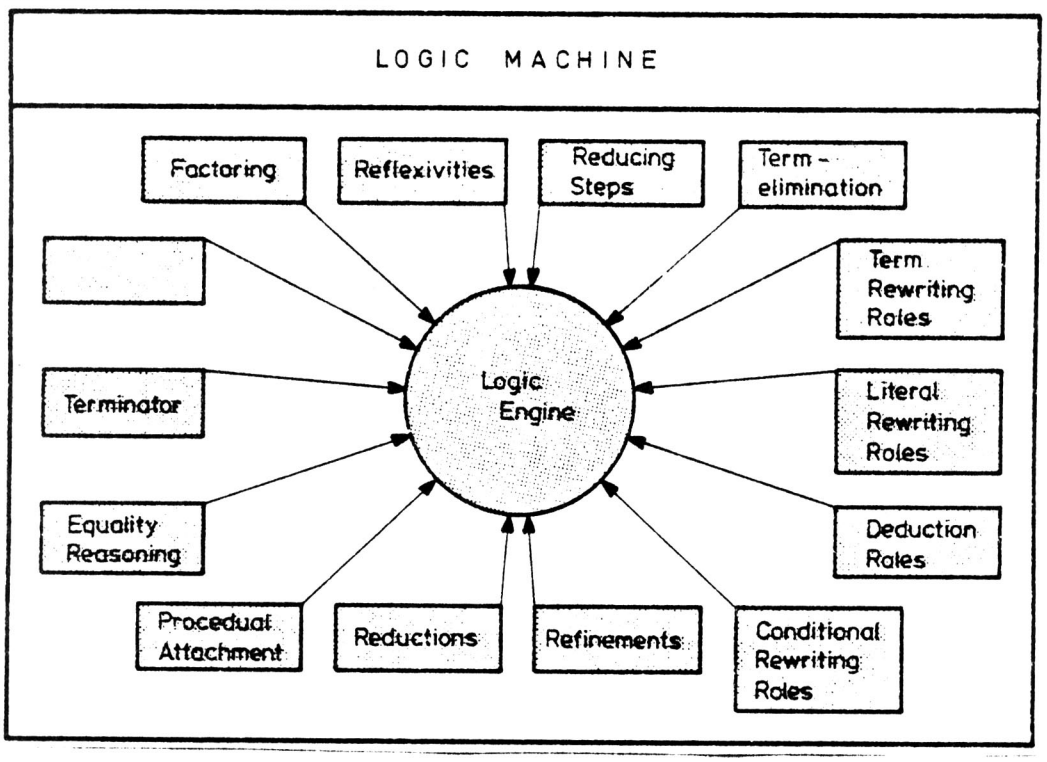


Fig.3

Each of these operation blocks corresponds to a particular task, which is usually to carry out a sequence of special deduction or

reduction steps. Once the activation condition that describes certain graph configurations, becomes true this task is executed and the local memory of the operation block is updated. In other words the flow of control - and hence the sequence of deduction and reduction steps actually carried out - is data driven by the current state of the clause graph and not preprogrammed. It should not be necessary to say that the complex interplay of these operation blocks, which 'suggest' which step to take next, prevents of course the overall deduction from being standard and the respective completeness results do not necessarily hold.

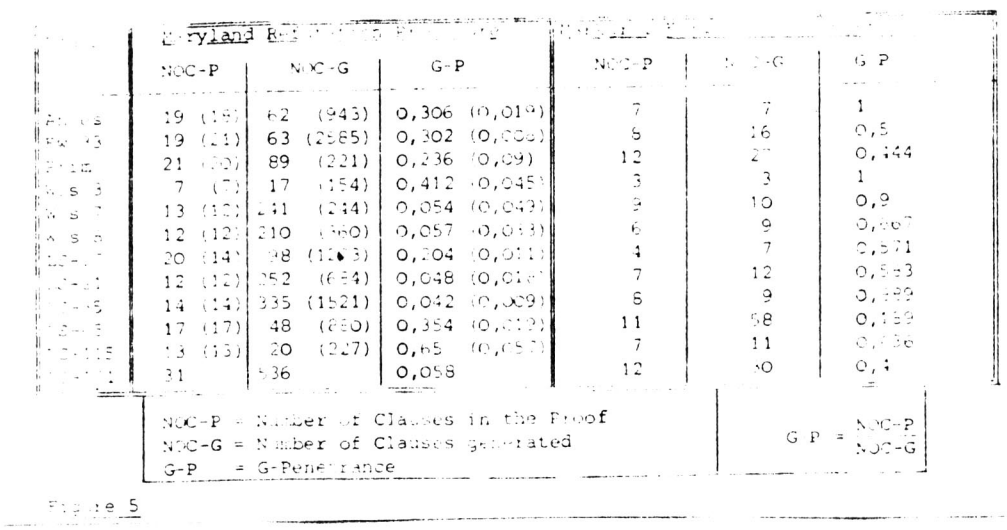
To summarize, five different levels may be distinguished within the MKRP system:

Level	Corresponding MKRP Component	Objects	Operations upon these Objects
ONE	PROBLEM SUPPLIERS	Formulas of the INPUT language	Syntactical analysis Semantical checks Syntax transformations
TWO	PREPROCESSORS	Wellformed formulas of the sorted calculus. Attributes of functions and predicates	Natural deduction. Splitting. Rewriting Special Decision Method Simplification Evaluation of terms
THREE	LOGIC MACHINE	Clause graphs	Graphoperations, like Resolution, Reduction, Deletion, Paramodulation, Rewriting, Symbolic Evaluation, etc.
FOUR	META REASONING	Graphoperations	Transformation of predicates into sort relationships. Transformation of clauses into deduction rules. Definition Expansion
FIVE	PROOF TRANSFORMERS	Proofs	Transformation of proofs Resolutionproof → Mating proof → Gentzenproof → Natural Language

Figure 4:

Performance Statistics

To gain a feeling for the improvement achieved up to now, Fig. 5 gives a sample of some test runs. In order to avoid one of the pitfalls of statistical data, which is to show the improvement achieved on certain examples and not showing the deterioration on others, the system has been tested on almost all of the main examples quoted in the ATP literature: [WM76], [RRYKU82]. Of all examples tested thus far, the examples of Fig. 5 are representative. They are taken from the extensive, comparative study undertaken at the University of Maryland [WM76], where eight different proof procedures were tested and statistically evaluated on a total of 152 examples.



	Maryland Bas					
	NOC-P	NOC-G	G-P	NOC-P	NOC-G	G-P
	19 (19)	62 (943)	0,306 (0,019)	7	7	1
	19 (21)	63 (2585)	0,302 (0,008)	6	16	0,5
	21 (30)	89 (221)	0,236 (0,09)	12	27	0,444
	7 (7)	17 (154)	0,412 (0,045)	3	3	1
	13 (13)	241 (2144)	0,054 (0,043)	9	10	0,9
	12 (12)	210 (960)	0,057 (0,013)	6	9	0,667
	20 (14)	98 (1103)	0,204 (0,011)	4	7	0,571
	12 (12)	252 (634)	0,048 (0,018)	7	12	0,583
	14 (14)	335 (1521)	0,042 (0,009)	6	9	0,667
	17 (17)	48 (880)	0,354 (0,019)	11	58	0,189
	13 (13)	20 (227)	0,65 (0,057)	7	11	0,636
	31	936	0,058	12	80	0,4

NOC-P = Number of Clauses in the Proof
NOC-G = Number of Clauses generated
G-P = G-Penetrance

G P = NOC-P / NOC-G

Figure 5

The table is to be understood as follows: the first column gives the name of the set of axioms in [WM76], e.g. LS-35 in line 9. The next three columns quote the findings of [WM76], where the figure in brackets gives the value for the worst proof procedure among the eight tested procedures and the other figure gives the

value for the proof procedure that performed best. The final three columns give the corresponding values for the Markgraf Karl Procedure. For example, in order to prove the axiom set LS-35 (line 9) the best proof procedure of [WMG76] had to generate 335 clauses in order to find the proof, which consisted of 14 clauses, and the worst proof procedure had to generate 1.521 clauses in order to find that proof. In contrast our system generated only 9 clauses in order to find an even shorter proof (of 8 clauses). As these figures are typical and hold uniformly for all cases, they are the statistical expression and justification for the first two claims put forward in the introduction.
The potential explosion of the number of links is the bottleneck of the connection graph proof procedure: the following 'challenge' proposed by P. Andrews, Carnegie Mellon at the 1979 deduction Workshop, provides a point of demonstration:

$$[(\exists x\ \forall y\ Px \equiv Py) \equiv ((\exists x\ Qx) \equiv (\forall y\ Py))]$$
$$\equiv [(\exists x\ \forall y\ Qx \equiv Qy) \equiv ((\exists x\ Px) \equiv (\forall y\ Qy))]$$

A straightforward translation of this formula into clausal normal form would result in up to 16 000 clauses (worst case). Using an improved translation algorithm the group at Argonne National Lab transformed it into a clauseset of 86 clauses with eight literals each and deduced 1052 clauses in order to find a proof [SIG76]. [SIG80].

The MKR-Procedure also uses an optimized translation algorithm, which generated 128 clauses of eight literals each. But this would result in an initial graph with more than 100 000 links and several thousand links would be added to the graph with each resolution step. If all these links were decleared "active" the computation of the selection functions would become intolerably expensive.

In actual fact the above formula is split and reduced into eight

parts of eight clauses each, where each clause has at most two literals. The initial eight graphs resulting from each split part never exceed 100 links and the system only deduced a total of 58 clauses before it easily found a proof. The G-penetrance varied from 0.8 to 1.0 during the eight runs.

The stipulated active and goal directed behavior of the MKRP-system, which finds its statistical expression in the very high G-penetrance, even holds for hard and extremely difficult theorems (judged by the standards of present day theorem proving systems). For example an open problem of modular lattice theory, which was first solved with the aid of a computer [GB69], has become well known in the ATP literature under the name of SAM's Lemma. The only system, so far capable of fully automatically finding a proof for this theorem, is that at Argonne National Lab [MOW76].

A protocol and description of how the MKRP-system found a proof also, is presented in Sec. 6.7 and the statistics there show the same hight G-penetrance.

Finally in [OW83] the proof found by the MKRP-system of a very difficult and until recently open problem taken from relevance logics is presented, which up to now none of the strong American systems could solve.

In [MOW76] some of the most difficult theorems so far proved by a TP system are presented and we have tested their examples also. Comparison with their reported results, shows that if the MKRP system finds a proof it is superior to the same degree as reported in Fig. 5 above.

However, there are still several difficult examples reported in [MOW76] which we can not prove at present. The strength of the system [MOW76] derives mainly from a successful technique to handle equality axioms and almost all the examples quoted in

[MOW76] rely on this technique. For that reason, as long as the equality reasoning modules of the MKRP-system are not fully developed, there is no fair comparison with respect to these examples.
Finally among the very large systems which presently dominate theorem proving research is the system developed by R. Boyer and J. S. Moore at SRI [BM78]. Their system relies on powerful induction techniques and although most of the examples quoted in [BM78] could be proved by the MKRP-system at present, a justifiable comparison is only possible once our induction modules are completed.

Kinship to other Deduction Systems

The advent of PLANNER [HE72] marked an important point in the history of automatic theorem proving research [AH72], and although none of the techniques proposed there are actually present in the MKRP system it is none the less the product of the shift of the research paradigm, of which PLANNER was an early hallmark.

More influential and directly relevant is the work of W. Bledsoe, University of Texas [BT75], [BB75], [BBH72], [BL71], [BL77]. However, in contrast to [BT75], we tried to separate as much as possible the logic within which the proofs are carried out from the heuristics which are helpful in finding the proof.

The strongest (resolution based) system up to now has been that of L. Wos and colleagues , Argonne National Lab and most of their techniques, like demodulation, heuristic weighting of terms, paramodulation and others have been adapted for the MKRP-system also. Although both systems are rather different in principle at present they vary little in strength.

The starting point for the Induction Modules was the theorem prover of Boyer and Moore [BM78]. The main difference between

their system and the MKR-Procedure is the use of a (slightly) different logic and the use of different theorem provers for the actual proofs of the base cases and the induction steps: wheras Boyer and Moore use a comperatively weak but specially tailored positive prover, the MKRP system uses the resolution (i.e. refutation) components of the Logic Machine.

6.3. METHODS AND TECHNIQUES

> "As a rule," said Holmes,"the more bizarre a thing is, the less mysterious it proves to be. It is your commonplace, featureless cases which are really puzzling."
>
> A.C. Doyle, The Red-Headed League

6.3.1. A many Sorted Calculus based on Resolution and Paramodulation

The MKR-Procedure is based on a sorted version of the first order predicate calculus. For example formulas like

(i) $\forall x{:}S.\ P(x)$ and $\exists x{:}S.\ P(x)$

are treated formally as abbreviations for

(ii) $\forall x.\ S(x) \vdash P(x)$ and $\exists x.\ S(x) \vdash P(x)$.

Well sorted formulas are frequently used in mathematics, because they provide a convenient shorthand notation for ordinary first-order formulas. But sorts also influence the deductions from a given set of well sorted formulas. For instance, if P is a predicate only defined on the sort **Z** of integers, we will never perform a deduction like $\forall x{:}\mathbf{Z}.\ P(x) \vdash P(\sqrt{2})$. Proofs are simplified, because a many-sorted calculus is more adapted to a many-sorted theory and hence not surprisingly deductions which respect sorts as well as the usage of well sorted formulas reflect the everyday usage of predicate logic.

A many-sorted (mehrsortig) calculus can be developed from a given (sound and complete) first-order one-sorted (einsortig) calculus as follows: Assume there is a set of sort symbols \$, ordered by a given subsort order $<_{\$}$ and variable and function symbols are associated with certain sort symbols. The sort of a term $\lceil t \rfloor$, which is different from a variable, is then determined by the sort of its outermost function symbol. Now in the construction of well sorted (sortenrecht) formulas for each argument position of

a function or predicate symbol only well sorted terms of a certain domainsort or of subsort of this domainsort are allowed.

The inference rules of the many-sorted calculus are the inference rules of the given calculus, but with the restriction that only well sorted formulas may be deduced. Starting with well sorted formulas this guarantees that only well sorted formulas are derived in a deduction. Now let $\vdash_\Sigma \Phi$ denote that Φ is a theorem of the many-sorted calculus and let AX $\vdash_\Sigma \Phi$ indicate that there is a deduction of Φ from the axioms AX. Further let us assume that there is a notion of truth for well sorted formulas and let $\models_\Sigma \Phi$ denote the validity of the well sorted formula Φ and let AX $\models_\Sigma \Phi$ denote the semantic implication. Obviously we are only interested in a many-sorted calculus which is sound and complete, i.e. we allow only definitions of $\vdash_\Sigma$ and $\models_\Sigma$ which guarantee

(1) $\models_\Sigma \Phi$ iff $\vdash_\Sigma \Phi$, for each well sorted formula Φ.

Assume our definitions satisfy (1): which formulas do we expect as theorems of the many-sorted calculus compared to its one-sorted counterpart? For a comparison, we let the relations between the function symbols and the sort symbols as well as the subsort order be represented by the set A^Σ of sort axioms (Sortenaxiome). For a well sorted formula Φ as e.g. (i) above, the relativization $\hat{\Phi}$ (Sortenbeschränkung, Relativierung) of Φ is the unabbreviated version of Φ e.g. (ii) above, where sort symbols are used as unary predicate symbols to express the sort of a variable. Now we can state what kind of theorems we expect in a many-sorted calculus: The definitions of $\vdash_\Sigma$ and $\models_\Sigma$ should ensure

(i) $\models_\Sigma \Phi$ iff $A^\Sigma \models \hat{\Phi}$ and

(2)

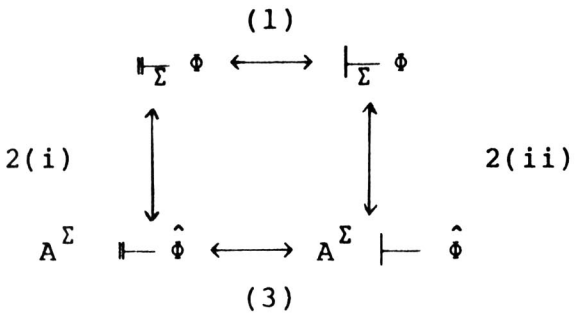
(ii) $\vdash_\Sigma \Phi$ iff $A^\Sigma \vdash \hat{\Phi}$, for each well sorted formula Φ.

Condition (2) is called the Sort-Theorem (Sortensatz), 2(i) is its modeltheoretic part and 2(ii) its prooftheoretic part.

The Sort-Theorem also shows the advantages of using a many-sorted calculus: shorter deductions with smaller formulas from a smaller set of hypotheses are obtained, when proving $\vdash_\Sigma \Phi$ instead of $A^\Sigma \vdash \hat{\Phi}$.

The reason is that deductions about sortrelationships, which are performed explicitly in the one-sorted calculus, are built into the inference mechanism in the many-sorted calculus.

The connection between a first-order one-sorted calculus and its many-sorted counterpart can be summarized as follows:



Suppose soundness and completeness of the given one-sorted calculus (3) are known. Then in order to show the commutativity of the above diagram either a proof of both parts of the Sort-Theorem 2(i) and 2(ii) is needed or a proof of one of its parts 2(i) or 2(ii) together with a proof of the soundness and completeness of the many-sorted calculus (1).

ooo

In his thesis, J.Herbrand presented a many-sorted version of his calculus and proved the prooftheoretic part of the Sort-Theorem [HER30]. However Herbrand's proof is inadequate, because he did not consider that certain deductions in his one-sorted calculus cannot be translated to deductions in the many-sorted calculus.

This was pointed out by A. Schmidt [SCH38], who proposed a many-sorted version of a Hilbert-Calculus without subsorts and proved the prooftheoretic part of the Sort-Theorem for this calculus [SCH38, SCH52].

H. Wang defined a many-sorted version of a Hilbert-Calculus without function symbols and subsorts [WAN52]. He proved the soundness and completeness of his calculus and the modeltheoretic part of the Sort-Theorem. Wang also gave an alternative proof of the prooftheoretic part of the Sort-Theorem by an application of the Herbrand-Theorem.

P.C. Gilmore pointed out that this proof is inadequate. He extended the many-sorted calculus of Wang by the introduction of subsorts and presented an improved version of the proof-theoretic part of the Sort-Theorem for this extended calculus [GIL58].

T. Hailperin presented a calculus which can be viewed as a generalization of Wangs's many-sorted calculus [HAI57]. In this calculus sortrelationships can be expressed by arbitrary first-order formulas instead of atomic formulas, i.e. unary predicates. Hailperin proved a theorem which corresponds to the prooftheoretic part of the Sort-Theorem.

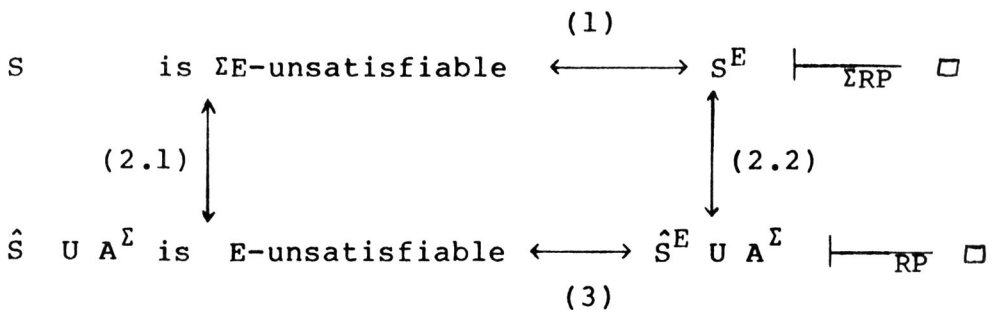
A. Oberschelp [OBE62] proposed several many-sorted versions of a calculus of Montague and Henkin [MH56]. In these calculi function symbols and subsorts are admitted. Oberschelp proved the soundness and completeness of his calculi and also gave the proofs for the modeltheoretic parts of the Sort-Theorems.
A.V. Idelson discussed forms of many-sorted calculi of constructive mathematical logic [IDE64], which are based on the calculus of natural deduction [GEN34].

ooo

The advantages of a many-sorted calculus are well recognized within the field of automated theorem proving e.g. [HAY71,HEN72] and several theorem proving programs successfully use some kind of many-sorted calculus, e.g. [WEY77, CHA78, BM79] (unfortunately without a sound theoretical foundation). Since most theorem proving programs are based on an RP-calculus, i.e. a first-order calculus whose inference rules are factorization, resolution and paramodulation [ROB65, WR73] and whose formulas (called clauses) are in skolemized conjunctive normal form, it would be useful to extend the results of the above quoted works to an RP-calculus enriched by sorts.

In [WA82] the ΣRP-calculus a many-sorted version of the RP-calculus is defined as outlined above and a notion of unsatisfiability of sets of well sorted clauses is introduced.

Soundness and completeness of the ΣRP-calculus as well as the modeltheoretic part of the Sort-Theorem are shown in [WA82], i.e. it is shown that the following diagram is commutative:



Here S^E denotes the extension of the set S of well sorted clauses by all functionally-reflexive axioms [WR73] and □ denotes the empty clause.
In particular [WA82] shows that the ΣRP-calculus is only complete provided the subsort order imposes a certain structure on the set of sort symbols. Moreover in the case of paramodulation the set of well sorted clauses to be refuted has to be in a certain format to ensure completeness.

It is also shown that these restrictions can be abandoned without loosing completeness, if the ΣRP-calculus is extended by an additional inference rule, the so called weakening rule, which derives a variant of a clause by renaming it with variables of a subsort. This rule is specific to a many-sorted calculus since it cannot be applied if only one sort is given and hence the RP-calculus is but a special case of the ΣRP-calculus.

The practical application of the ΣRP-calculus in the MKRP system, leads to a drastic reduction of the search space and to shorter refutations of smaller sets of shorter clauses as compared to the RP-calculus. However certain modifications are necessary to obtain a theorem prover based on the ΣRP-calculus. Essentially they concern:

- the input-language compiler
- the skolemization routine
- the unification algorithm and
- the computation of factors, resolvents and paramodulants.

The Compiler

The Compiler tests whether a given input string satisfies the rules of syntax and those of the ´static semantics`, for example that function symbols are used with a proper arity etc., and produces as ´code` a first-order formula in an internal representation.

The rules of the static semantics have to be extended such that only well sorted first-order formulas are accepted: For each atomic formula A in a formula given as input, the compiler has to determine whether A is a well sorted atomic formula.

This problem is the same as for programming languages with sorts (often called types, e.g. in PASCAL or ADA) and was solved using wellknown techniques of compiler construction.

In addition a device is required to define a set of sort symbols, a subsort order and some sorted signature (see Sec. 6.6.).

The Skolemization Routine

The skolemization of a first-order formula requires that each occurence of an existentially quantified variable symbol y is replaced by a skolem term t, and all existential quantifiers are removed.

For Σ-skolemization, i.e. skolemization under sorts, this process remains the same, but in addition the signature Σ has to be extended, yielding a signature Σ^* for the new function symbols introduced by the skolemization.

The Unification Algorithm

At the very heart of every unification algorithm, variable symbols have to be replaced by terms. The resulting substitution, represented as $\{x \leftarrow t\}$, is then composed with other substitutions of this kind, yielding the final unifier. Hence each unification algorithm contains a sequence of statements like

(1) **if** x = t **then return** { }
(2) **if** x ε vars ({t}) **then** stop/failure
(3) **return** {x←t}

The unification algorithm is modified to obtain a Σunification algorithm by replacing statement (3) by the sequence of statements:

(3.1) **if** $[t] <_S [x]$ **then return** {x←t}
(3.2) **if** t is not a variable **or** $[t] \sqcap_S [x] = \emptyset$ **then** stop/failure
(3.3) **if** $[x] <_S [t]$ **then return** {t←x}
(3.4) **let** $\{s_1 \ldots s_k\} = \max([t] \sqcap_S [x])$

(3.5) **let** $\{z_1,\ldots,z_k\}$ be a set of new variables and $[z_i] = s_i$
(3.6) **return** $\{\{x \leftarrow z_i,\ t \leftarrow z_1\},\ldots,\{x \leftarrow z_k,\ t \leftarrow z_k\}\}$

Here $s_1 \sqcap_\$ s_2 = \{s \varepsilon \$ \mid s <_\$ s_1 \text{ and } s <_\$ s_2\}$ and
$\max(\$) = \{s \varepsilon \$ \mid s <_\$ s' \text{ for each } s' \varepsilon \$\}$

For each Σ-unifiable set of Σ-terms the Σ-unification algorithm returns a set of Σ-unifiers (and not a single unifier as usual), because a unification problem may have several most general solutions under sorts.

Computation of Factors, Resolvents and Paramodulants
We outline an implementation, which avoids the explicit computation of weakened variants:

Let A be a clause in a Σ-deduction and let $B \subseteq A$ such that $|B| > 2$ and let U(B) be the Σ-mgus for the literals in B:

$$U(B) = \{\tau_1,\ldots,\tau_n\}.$$

Then every Σ-clause $\tau_i A$ has to be computed
each of which is a Σ-factor of a weakened variant of A.
Let A,B be clauses in a Σ-deduction, $L_A \varepsilon A$ and $L_B \varepsilon B$ such that L_A and L_B are complementary and

$$U(\{|L_A|, |L_B|\}) = \{\tau_1,\ldots,\tau_n\}.$$

Then every Σ-clause has to be computed

$$\tau_i(A - L_A) \ \cup \ \tau_i(B - L_B),$$

each of which is a Σ-resolvent of some weakened variant of A and B.

Let A, B be clauses and let $l \equiv r \varepsilon B$ and $L \varepsilon A$ be the literal to be paramodulated upon, i.e. l and a subterm in L are Σ-unifiable.
Now a Σ-paramodulant can only be derived if the sort of r is also compatible with the position of the subterm in L : this again may drastically reduce the search space.
For each weakening substitution every Σ-paramodulant of some weakend variant of A and B has to be computed.

After the computation of a Σ-factor, Σ-resolvent or Σ-paramodulant the variable symbols of these Σ-clauses have to be renamed using an appropriate **Σ-renaming substitution.**

The Markgraf Karl Refutation Procedure was adapted to the ΣRP-calculus according to the modifications stated above and the following is a proof protocol of the new system, proving a many-sorted version of the well known **monkey-banana-problem** [LOV78]:

```
*****************************************************************
*
* MARKGRAF KARL REFUTATION PROCEDURE, UNI KARLSRUHE, VERSION
* 12-OCT-82
* DATE: 2-Nov-82 16:46:27
*
*****************************************************************

AXIOMS GIVEN TO THE THEOREM PROVER:
  SORT ANIMAL, TALL: IN.ROOM
  TYPE BANANA, FLOOR: IN.ROOM
  TYPE CHAIR: TALL
  TYPE MONKEY: ANIMAL
  TYPE CAN.REACH (ANIMAL IN.ROOM)
  TYPE CLOSE.TO (IN.ROOM IN.ROOM)
  TYPE ON (IN.ROOM IN.ROOM)
  TYPE UNDER (IN.ROOM IN.ROOM)
  TYPE CAN.MOVE.NEAR (ANIMAL IN.ROOM IN.ROOM)
  TYPE CAN.CLIMB(ANIMAL TALL)
  AXM1 : ALL X:ANIMAL Y:IN.ROOM NOT CLOSE.TO(X Y) OR CAN.REACH(X
         Y)
  AXM2  : ALL X:ANIMAL Y:TALL NOT ON (X Y) OR NOT UNDER(Y BANANA)
         OR CLOSE.TO(X BANANA)
  AXM3  : ALL X:ANIMAL Y:IN.ROOM Z:IN.ROOM NOT CAN.MOVE.NEAR(X Y
         Z ) OR CLOSE.TO(Z FLOOR) OR UNDER (Y Z)
  AXM4  : ALL X:ANIMAL Y:TALL NOT CAN.CLIMB(X Y) OR ON(X Y)
  AXM5  : CAN.MOVE.NEAR(MONKEY CHAIR BANANA)
```

```
  AXM6   : NOT CLOSE.TO (BANANA FLOOR)
  AXM7   : CAN.CLIMB(MONKEY CHAIR)
THEOREM GIVEN TO THE THEOREM PROVER:
  THM8 : NOT CAN.REACH(MONKEY BANANA)

AXM2 + AXM3 --> RES1 : ALL X:ANIMAL Y:TALL Z:ANIMAL CLOSE.TO(X
                                    BANANA)
                       OR NOT ON(X Y) OR CLOSE.TO(BANANA FLOOR)
                       OR NOT CAN.MOVE.NEAR(Z Y BANANA)
RES1 + AXM5 --> RES2 : ALL X:ANIMAL CLOSE.TO(BANANA FLOOR) OR NOT
                      ON(X CHAIR)
                      OR CLOSE.TO (X BANANA)
AXM1 + RES2 --> RES3 : ALL X:ANIMAL CAN.REACH(X BABANA) OR NOT ON
                                    (X CHAIR)
                      OR CLOSE.TO(BANANA FLOOR)
ASM6 + RES3 --> RES 4 : ALL X:ANIMAL NOT ON(X CHAIR) OR
                                    CAN.REACH(X BANANA)
THM8 + RES4 --> RES5 : NOT ON(MONKEY CHAIR)
RES5 + AXM4 --> RES6 : NOT CAN.CLIMB(MONKEY CHAIR)
RES6 + AXM7 --> RES7 : EMPTY

THE FOLLOWING CLAUSES WERE USED IN THE PROOF:
AXM7 AXM4 AXM5 AXM3 AXM2 RES1 RES2 AXM1 RES3 AXM6 RES4 THM8 RES5
RES6 RES7.

THE THEOREM IS PROVED. END OF PROOF 2-NOV-82 16:47:22.
```

The following expressions (see Sec.6.6) are used:
SORT $s_1,...s_n$:s to denote $s_i <<_s s$,
TYPE $c_1,...,c_n$: s to denote that $c_i \varepsilon C$ has rangesort s
TYPE $P(s_1...s_n)$ to denote that P has the domainsorts $s_1,...$
ALL x:s to denote the universal quantification of a variable symbol x with rangesorts
The system also solved the monkey-banana-problem, using the one-sorted axiomatization from [LOV78]. The following diagram shows the proof statistics of both example runs:

CPU-SECONDS USED:	3.32	11.38	29%
NUMBER OF STEPS EXECUTED:	7	16	44%
NUMBER OF LINKS GENERATED:	22	99	22%
NUMBER OF LINKS IN INITIAL GRAPH:	8	23	35%
NUMBER OF CLAUSES GENERATED:	15	29	52%
INITIAL CLAUSES:	8	13	
RESOLVENTS:	7	12	
FACTORS:	0	4	
NUMBER OF LITERALS GENERATED:	28	75	37%
IN INITIAL CLAUSES:	14	24	
IN DEDUCED CLAUSES:	14	51	
LEVEL OF PROOF:	7	12	58%
NUMBER OF CLAUSES IN PROOF:	15	25	60%

G-PENETRANCE: 1.00 0.86(CLAUSES IN PROOF / CLAUSES GENERATED)
D-PENETRANCE: 1.00 0.75(DEDUCED CLAUSES IN PROOF / CLAUSES DEDUCED)

The first column lists the statistical values for the proof using the many-sorted calculus, the second column lists the values for the one-sorted calculus and the third column shows the ratio between the values of both example runs.

In the proof statistics, the value for ´number of links generated` corresponds to the size of the search space, the value for ´number of steps executed` is a measure of the expense of the actual search and ´level of proof` represents the search depth.

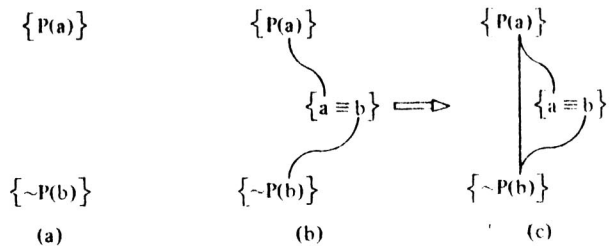
The comparison between the statistical values of both protocols immediately reveals the advantages of using an automated theorem prover based on the ΣRP-calculus. The values are typical for all examples (and of course for more complex ones) that have been proved by this system.

6.3.2. Paramodulated Connection Graphs

This section, which is based on [SW80] gives an account of how the connection graph proof procedure of [KO75] can be extended to the case of equality by the introduction of special links connecting those terms that can be paramodulated upon.

To gain a notion of the problem involved, assume that two one-literal ground clauses {P(a)} and {~P(b)} form a connection graph (Example 1(a)).

Example 1



As they stand these two clauses are not resolvable with each other. Now if it were known that a is equal to b and some way could be found of entering this information into the connection graph so as to be able to make a substitution into one of the literals then the two literals would become resolvable. The initial idea is to introduce this information in the form of an equality unit {a≡b} which is then connected to the clauses by two special links indicating possible paramodulations, either of b into {P(a)} of a into {~P(b)} (Example 1(b)). (These links will be called paramodulation links or P-links in the sequel, as opposed to the normal links connecting unifiable literals, hereafter referred to as R-links.)

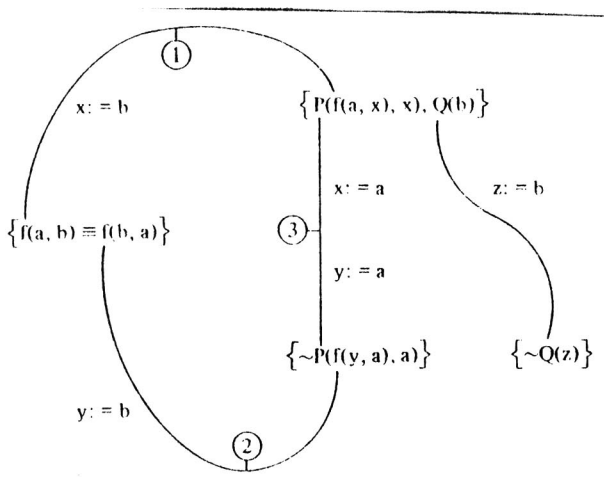
Now, to complete the idea, a link is included to indicate the potential resolvability of {P(a)} and {~P(b)} (Example 1(c)). But because these clauses are not unifiable as they stand, the link cannot be an ordinary R-link. Rather the link indicates that the

two literals connected can be made complementary under paramodulation: that is after certain paramodulation steps these two literals may be resolved upon. The information represented by such a link is absolutely essential, if the basic principle of the connection graph proof procedure is to be extended to the case of paramodulation. Unfortunately it is in general undecidable, whether such a link is to be set or not.

This fundamental problem will be discussed in the following paragraph, but before an example is given to demonstrate some of the advantages which would result, if the connection graph proof procedure could be extended by paramodulation links.

Example 2 illustrates the incompatibility deletion for paramodulation links. The graph consists of the four clauses {P(f(a,x),x),Q(b)}, {~Q(z)},{~P(f(y,a),a)} and {f(a,b)≡f(b,a)}. All possible P- and R-links are set and

Example 2



each link is labeled with it´s corresponding substitution, e.g. x<=b in P-link (1).

Paramodulation links (1) and (2) are incompatible with link (3) and may hence be deleted, since the respective paramodulants would contain pure literals. Note that in the case of incompatibility between a paramodulation link and a resolution link always the paramodulation link is erased.

The point of demonstration is, that the equality unit becomes pure after the incompatibility check and hence the whole (equality) clause is erased. This may lead to a snowball-effect of other clauses being erased as a consequence. This familiar effect from the original proof procedure is demonstrated here to hold also for R-link and P-links, hence apart from the obvious motivation to provide for equality rule in connection graphs, there is the additional chance that the graph enriched by P-links may reduce much more rapidly.

The Problem

In order to achieve the effect mentioned in the previous paragraph, we would like to have a connection graph as dense as possible. In particular we would like to have the following links:

(i) all complementary literals in different clauses which are unifiable are connected with a link called R-link;

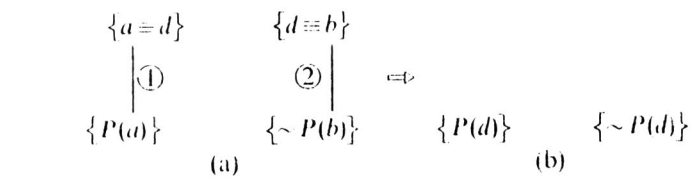
(ii) all equations permitting paramodulation into a particular term of some literal are connected with a link to that term, called P-link;

(iii) all literals which can be made complementary under paramodulation and unification are connected by a link.

The problem concerns the links of (iii): it is neither practically feasible nor (in general) theoretically possible to ever set all links of (iii). Consider a group G whose wordproblem

is unsolvable. Let S be a set of clauses containing all the equations defining group G. Let $P(w_1)$ and $\sim P(w_2)$ also be in S, where w_1 and w_2 are words in G and it is now impossible to decide whether or not $P(w_1)$ and $\sim P(w_2)$ should be connected. So let us try the following solution: Suppose we do not initially set all the links in (iii), then the resulting situation arises: Let S={{a≡d}, {b≡d}, {P(a)}, {~P(b)}} and the initial connection graph is given in example 4(a):

Example 4



Now, after paramodulating on links(1) and (2), we obtain the connection graph ⟨{P(d)}, {~P(d)}⟩ in example 4(b). But since P(a) and ~P(b) were not connected by a link, the paramodulants cannot inherit any links,i.e. P(d) and ~P(d) are <u>not connected</u> by a link and therefore cannot be resolved upon. The obvious solution then may be, after each paramodulation step, to search through the whole graph and compare the paramodulant with every other literal for unifiability.

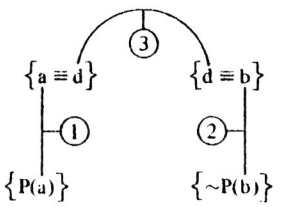
But to start searching for unifiable literals after each paramodulation step would destroy one of the main principles of the connection graph proof procedure, which is precisely to eliminate the unsuccessful search for unifiability.

A Solution to the Problem

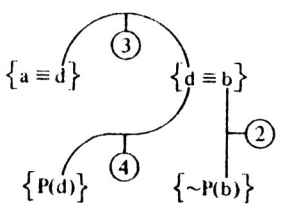
In order to solve this problem we demonstrate that P-links can have more than just the one function of recording possible

paramodulations. Their additional function is to store information.

Consider example 4(a) again in which we add another P-link connecting the d's of the two equality units i.e. the graph now becomes:

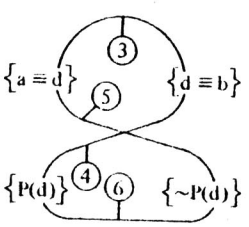


Suppose we select link (1)



where (4) is the link inherited from (3).

If we now select link (2)



where (5) is inherited from (3) and (6) is inherited from (4).

If links are inherited which eventually connect non-equality literals (link (6) above) a special process is initialized which checks whether the connected literals are resolvable and if so then the P-link is recoloured into an R-link. Otherwise it is erased.

That is, potentially unifiable literals are discovered (without exhaustive search) and the appropriate links are set, including connections between the right- and left-hand side of equality literals as e.g. in:

$$\{\overbrace{a=g(a)}\}$$

(such links are called ´auto-links` later on).

The Proof Procedure

The following proof procedure PROOF is essentially that of [KO75] augmented by additional steps to handle paramodulation. However we do not quite use the basic procedure of [KO75], but a refinement due to M. Bruynooghe, which avoids certain redundancies of the old procedure by employing ´internal links`. Such ´internal links` were also suggested by R.Kowalski in [KO75]. Note that a resolution step (paramodulation step) on such an ´internal link` is to be understood as a step using two copies of the same clause. Also we do not label each link with its associated most general unifier, as once the unifiability of two terms has been established the retention of the mgu is logically superfluous for the procedure presented below. An actual implementation however, may or may not retain this information for pragmatic reasons.

At the top level of the proof procedure PROOF initially sets up the connection graph $\langle S_1 \rangle$ for the set of clauses S with the help of CONSTRUCT-GRAPH and then calls PROVE.

Let S be a set of clauses.

```
PROOF(S)
  CONSTRUCT-GRAPH(S)→<S1>
  PROVE(<S1>)
END-OF-PROOF.
```

The recursive procedure PROVE terminates if the graph contains the empty clause or is empty itself and otherwise SELECT-A-LINK non-deterministically. After either RESOLVEing or PARAMODulating upon the link l in the graph $\langle S_1 \rangle$ (depending on the type of l) all tautologies and pure clauses are removed by DELETE, which then returns a new graph to PROVE.

Let $\langle S_1 \rangle$ be a connection graph.

```
PROVES(<S1>)
      if □∈<S1> then terminate with <S1> unsatisfiable,
  elseif <S1> is empty then terminate with <S1> satisfiable.
      SELECT-A-LINK(<S1>)→l
  if type(l)=R-link then RESOLVE(<S1>,l)→<S2>
  elseif type(l)=P-link  then PARAMOD(<S1>,l)→<S2>
END-OF-PROVE
```

Let S be a set of clauses.

CONSTRUCT-GRAPH(S)

(1) Generate and include in the graph all factors of clauses in S.

(2.1) For every pair(|L|, |K|) of unifiable (equality or nonequality) literals with opposite sign in distinct clauses in the graph, insert an R-link connecting the literals L and K.

(2.2) For every pair (|L|,|K|) of unifiable (renamed) literals with opposite sign in the same clause, insert an R-link

connecting the literals.

(3.1) For every pair of literals consisting of either a non-equality or equality literal and a second equality literal in distinct clauses, such that the left-hand side (or right-hand side) of the second equality is unifiable with some term in the first literal, insert a P-link connecting the terms in the literals.

(3.2) For every equality literal containing a term on one side which can be substituted into the term on the other side, insert a P-link connecting the terms in the literal.

END-OF-CONSTRUCT-GRAPH

Let $\langle S_1 \rangle$ be a connection graph and let l be an R-link in $\langle S_1 \rangle$. Let C and D be the clauses connected by l.

RESOLVE($\langle S_1 \rangle$,l)

Add the resolvent R of C and D associated with l to $\langle S_1 \rangle$.

INHERIT-R-LINKS($\langle S_1 \rangle$,C,D,R).

INHERIT-P-LINKS($\langle S_1 \rangle$,C,D,R).

For each factor F of R:

Add F to $\langle S_1 \rangle$

INHERIT-R-LINKS($\langle S_1 \rangle$,R,R,F)

INHERIT-P-LINKS($\langle S_1 \rangle$,R,R,F).

* Add all possible R-links and P-links between the factor of R and R.

Return the new graph

END-OF-RESOLVE.

Note: Step* should be avoided in an appropriate refinement.

Let $\langle S_1 \rangle$ be a connection graph and let l be a P-link in $\langle S_1 \rangle$. Let C be the clause containing an equality literal and let D be the clause paramodulated upon.

PARAMOD($\langle S_1 \rangle$,l)

Add the paramodulant P associated with l to $\langle S_1 \rangle$

INHERIT-R-LINKS($\langle S_1 \rangle$,C,D,P)

INHERIT-P-LINKS($\langle S_1 \rangle$,C,D,P)

If a new P-link connects a term in a literal L in P to some other term in a nonequality literal K
andif L and K are complementary unifiable
then erase the P-link and connect L and K by an R-link
else erase the P-link
For each factor F of P:
Add F to $\langle S_1 \rangle$
INHERIT-R-LINKS($\langle S_1 \rangle$,P,P,F)
INHERIT-P-LINKS($\langle S_1 \rangle$,P,P,F)
*Add all possible P-links and R-links between the factors of R and R
Return the new graph

END-OF-PARAMOD.

Note: Step* should be avoided in an appropriate refinement.

The INHERIT-procedures above connect the newly generated paramodulant and resolvent to the graph and the DELETE procedure erases all tautologies and pure clauses.

In [SW80] soundness and completeness of the paramodulated connection graph proof procedure are shown relative to the respective results for the original proof procedure; i.e. it is shown that if the proof procedure of Kowalski is sound and complete so is this one.
Recently it was shown that Kowalski's proof procedure is not complete - in fact it is not even consistent .

Equality Reasoning

The proof procedure as presented has certain advantages over ordinary paramodulation, for practical purposes however the search space is still too explosive: there are just too many links - i.e. paramodulation possibilities - in the graph.

In order to overcome this problem we originally experimented with certain refinements, but even then, the explosion in the number of P-links is unmanageable for any real application.
In 6.3.10 a feasable solution for this problem is presented.

6.3.3 Subsumption and Connection Graphs

One of the striking properties of the connection graph proof procedure is that the application of a deletion operation can result in the applicability of further deletion operations, thus potentially leading to a snowball effect which rapidly reduces the graph. The probability of this effect rises with the number of deletion rules available.

A very powerful deletion rule for resolution based systems is the subsumption rule ([LOV78]). Unfortunately a test for subsumption is very expensive and is usally implemented only for restricted cases. In this section a test for subsumption based on the principal idea of the connection graph proof procedure is developed, which in contrast to the standard test ([LOV78]) is sufficiently efficient to permit unrestricted subsumption in practical cases. Though not limited to it, the test is most naturally embedded into the connection graph proof procedure, but unrestricted combination of subsumption or tautology with purity deletion is shown in [EI81] to make a connection graph proof procedure inconsistent and hence incomplete.

The deletion rules contribute both to the practical attractivity and the theoretical difficulties of the connection graph proof procedure. The original rules of [KO75] are: delete a clause if it contains a pure literal and delete a clause which is a tautology. Further possible rules include: delete a link if its resolvent is a tautology (see 6.3.4.), delete a clause if it is subsumed by another clause in the graph, delete a link if its resolvent is subsumed by another clause in the graph. Note that each deletion may cause a purity to arise, thereby causing

further deletions. It is not yet known as to which combinations of these deletion rules preserve the completeness of the procedure.

Subsumption and the S-link Test

Let C and D be clauses. C θ-subsumes D, if $|C| \leq |D|$ and θ is a substitution such that $\theta(C) \subset D$.The standard test for θ-subsumption works as follows: given C and D, first make sure that $|C| \leq |D|$ and that D is not a tautology. Then negate D and change variables in D to constants, yielding a set D of ground unit clauses. C θ-subsumes D iff $\square$ is derivable from $\{C\} \cup D$. (Details can be found in [CL73] and [LOV78].

The positive aspect of this subsumption test is that it uses the same mechanism which underlies the entire deduction system, i.e. resolution. But from a practical point of view this turns out to be a disadvantage. Normally one has to check for subsumption as soon as a new clause is generated, i.e. after each resolution step. This means that each "major" resolution step is followed by several "minor" resolution steps for the subsumption test, thus multiplying the overall expense. Yet even worse, given a resolvent C there is no hint as to which clauses potentially subsume or are subsumed by C. So the test, already expensive in itself, has to be performed within an iteration over all elements of the given set of clauses. In practice, of course, one would first make sure that the predicates are in common, so that the test is not performed during each iteration step.

The resulting cost is such that for practical systems only restricted versions of subsumption are implemented, e.g. only for cases where the subsuming clause is a unit. Omitting subsumption, on the other hand, can cause considerable redundancies.

The central problem for subsumption test consists in efficiently finding out which literals in which clauses are unifiable.

Disregarding the signs of the literals this corresponds to the very same problem that arises when two clauses have to be selected for the next resolution step. In both cases comparing all literals of all clauses is a possible but inefficient solution.

In the resolution case the connection graph procedure provides for a more efficient alternative. The literals of a set of clauses are compared with each other once and forever when the initial graph is constructed. Subsequently the necessary information is directly available in the form of the links. Because of the inheritance mechanism for links the new literals in resolvents and factors need not go through any search process either. Thus the problem of finding two resolvable clauses is reduced to simply picking a link.

This basic idea can be applied to the subsumption problem by introducing links of a new type that connect unifiable literals. Formally we define a subsumption graph (S-graph) as a pair (C, S) such that

1) C is a set of clauses
2) Let LIT be the set of all literals occuring in the clauses of C. Then $S \subseteq C \times LIT \times \underline{C} \times LIT$ is a relation such that
   a) $(C, L, C', L') \in S \rightarrow C \neq C', L \in C, L' \in C'$, L and L' are unifiable
   b) $(C, L, C', L') \in \underline{S} \leftrightarrow (C', L', C, L) \in \underline{S}$

The S-graph is said to be S-total, if condition 2a) also holds in the opposite direction. A literal L in a clause C is S-pure, if there are no C', L' such that $(C, L, C', L') \in \underline{S}$. The elements of $\underline{S}$ are called S-links.

Given a set of clauses to be refuted, we initially compute all possible S-links between literals in these clauses. When a new resolvent or factor is derived, the S-links are inherited from the parent clauses in the same way as the resolution links in

the connection graph proof procedure. But in contrast to resolution links (R-links) S-links are never deleted, unless one of the parent clauses is removed from C it can be shown easily that under these circumstances the S-graph remains S-total throughout the entire computation (see [EI81]).

In order to develop a subsumption test using s-graphs we need some further definitions:
Let $(\underline{C},\underline{S})$ be a S-graph, $c\varepsilon\underline{C}$ a clause and $L\varepsilon c$ a literal.
We define $con(C,L) := \{D\varepsilon\underline{C}) \mid \exists K\varepsilon D\ (C,L,D,K)\varepsilon\underline{S}\}$ as the set of all clauses connected to L in C by S-links.

Further let $sub(C) := \bigcap con(C,L)$ be the set of all clauses connected to every literal in L by S-links

For $L\varepsilon C$ and another clause $D\varepsilon\underline{C}$ we define
$uni(C,L,D) := \{\sigma \mid \exists K\varepsilon D: (C,L,D,K)\varepsilon S \wedge \sigma(L) = K\}$ as the set of all matching substitutions mapping L onto some literal in D. Finally let $U_1,\ldots,U_n$ be sets of substitutions. Then $merge(U_1,\ldots,U_n) := \{(\sigma_1,\ldots,\sigma_n)\varepsilon U_1 x,\ldots x U_n \mid$ the σ_i are pairwise strongly compatible$\}$ is the subset of their cartesian product, for which the functional composition of the components yields a unique substitution regardless of their order.

The subsumption test is provided by the following theorems:

Theorem 1

Let $(\underline{C},\underline{S})$ be an S-total subsumption graph and $C = \{L_1,\ldots,L_n\}\varepsilon\underline{C}$ a clause, $n > 1$. Then for $D\varepsilon\underline{C}$
C σ-subsumes D iff $|C| \leq |D| \wedge D\varepsilon sub(C) \wedge$
$merge(uni(C,L_1,D),\ldots,uni(C,L_n,D)) \neq \phi$.

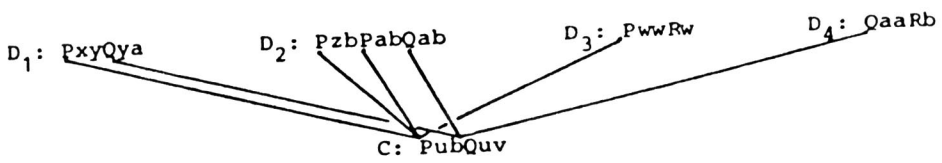
Theorem 2

Let $(\underline{C},\underline{S})$ be an S-total subsumption graph and $D = \{K_1,\ldots,K_m\}\varepsilon\underline{C}$ a clause, $m > 1$. Then for $C \neq \square$

$$C\ \sigma\text{-subsumes D only if } C\varepsilon\bigcup_{i=1}^{n} con(D,K_i).$$

Detailed proofs can be found in [EI81].

The following example illustrates the principle of a test based on Theorem 1. Assume the set of clauses $\underline{C}$ = {C, D_1,D_2,D_3,D_4} with C = {Pub,Quv}, D_1 = {Pxy,Qya}, D_2 = {Pzb,Pab,Qab}, D_3 = {Pww,Rw}, D_4 = {Qaa,Rb}. We want to find all clauses subsumed by C. In this case only S-links connected to C are relevant, so for reasons of clarity all other S-links are omitted in the S-graph for C:



Now |C| ≤ |D_i| for all i and sub(C) = {D_1,D_2}, because D_3 and D_4 are each connected to only one literal of C. We have to associate with each literal of C a set of matching substitutions for each clause in sub(C). In this case uni(C,Pub,D_1) = ∅ because there is no σ such that σ(Pub) = Pxy. Thus D_1 can be disregarded.

For D_2 we obtain uni(C,Pub,D_2) = {{u := z}, {u := a}} and uni(C,Quv,D_2) = {{u := a, v := b}}. The substitutions {u := a} and {u := a, v := b} are strongly compatible, thus C subsumes D_2 (but none of D_1,D_3,D_4).

The main point of the test is that the expensive unification is postponed until the function sub preselected a plausible subset of candidates for subsumption. In case we are looking for subsuming rather than subsumed clause, this preselection process is slightly more complicated. By Theorem 2 we first determine all clauses connected by at least one S-link to the given clause D. Then from this set we ascertain those C for which |C| ≤ |D| and

D∈sub(C) holds and only then the unification operations are performed.

In both cases this preselection and the fact that the literals to be unified are explicitly known, can save considerable time. On the other hand some effort has to be invested for the computation of all S-links in the initial graph. As with the connection graph proof procedure this advance cost can be higher than a possible gain, if the set of clauses is only small. For more complex examples however, there is certainly a pay off. But of course any gain in time has to be paid for by additional storage needed for the S-links.

Refinements of the S-link Test

The inheriting mechanism of S-links can be optimized in the same way as described in [B75] for resolution links. Here the proofs are very simple because S-graphs are always total.

Another refinement results from the observation that for a clause C contaning an S-pure literal sub(C) = Ø. That means that such a clause cannot subsume any other clause.

When computing the uni(C,L^i,D) we need to know which K∈D are unifiable with L^i. As the definition of uni shows, these are exactly those literals we already had to consider for the computation of con and sub. The information obtained during the computation of con and sub should be stored in an appropriate data structure to avoid having to recompute it for uni.

Thus far the S-link test was developed without considering the underlying inference mechanism. Since they are based on the same principal idea as connection graphs, S-graphs appear to be combined most naturally with this inference method. We can modify the definition of a connection graph to be a triple (C,R,S) such that (C,R) is a connection graph (in the hitherto sense) and

(C,S) is an S-graph. For such a graph we can define a new kind of subsumption: a clause C subsumes an R-link (D_1,K_1,D_2,K_2), if C subsumes the resolvent of (D_1,K_1,D_2,K_2).

This link subsumption rule is a very powerful technique, since deleting subsumed resolvents has the effect as if all resolutions leading to subsumed resolvents were performed prior to other steps. This results in a stronger reduction of the graph than in the usual case where subsumptions occur only randomly. The difference is similar to the one between deleting tautology clauses and deleting tautology links in a graph (see 6.3.4.).

Practical Results

On the average a graph has about the same number of S-links as it has R-links. This may seem an inappropriate increase in storage requirement. But in the actual implementation S-links need much less storage than R-links. Moreover, it is not the physical storage that is important, but the number of active R-links in the search space, and this number can be reduced considerably.

Practical tests indicate that the reduction of the graph caused by subsumption usually more than compensates for the storage used by the S-links. An example is P. Andrews 'challenge' proposed at the deduction workshop in Austin 1979:($\exists$ x Qx $\equiv$ $\forall$ y Qy) $\equiv$ ($\exists$x $\forall$ y Qx $\equiv$ Qy). Here subsumption reduces the initial graph by 89% of the clauses and by 99,5% of the R-links (which is however an extreme case).

In order to get some experience with more "natural" problems, a selection of examples from [MOW76] and [WM76] was run using the strategies basic resolution, set-off-support, and unit refutation, each with and without subsumption.

The results show (see [EI81] for the actual figures) that subsumption usually caused a considerable improvement of the

penetrances, i.e. fewer unnecessary steps were performed. This demonstrates the reduction of the search space. Sometimes the system even found a proof where it did not without subsumption. The incrase of the R-value indicates that subsumption infact has a very strong impact on the size of the graph.

6.3.4 Deletion of Redundant Links in Connection graphs

The connection graph proof procedure allows the removal of a clause from the graph, if it is a tautology or a pure clause.

This clause deletion rule can be transferred to links if we view a link as a potential clause: each link represents a potential factor, resolvent or paramodulant. Links which generate tautologies or pure clauses are called redundant links. Since non-redundant well as redundant links are copied (i.e. inherited) in the process of a derivation, the elimination of redundant links as early as possible (i.e. immediately after their generation) prohibits their occasional exponential growth. The connection graph proof procedure is extended by several new types of links in order to formulate necessary and sufficient criteria that links are redundant.

Multicoloured Graphs

A connection graph is a set of clauses together with several sets of links. The latter include so far:

› F-LINKS, the set of factorisation links

› R-LINKS, the set of resolution links

› P-LINKS, the set of paramodulation links

› IP-LINKS, the set of information links

› S-LINKS, the set of subsumption links

› T-LINKS, the set of tautology links which are subclassified into

TS-LINKS
TR-LINKS
TP-LINKS (see below)

› PC-LINKS, the set of parent connector links (see below)

Some of these links may again be classified as active, passive, inhibited or inheritance only and in addition are sorted according to various priorities as described in section 6.4.

A factorisation link (F-link) is a pair $\langle C,\sigma\rangle$, where C is a clause and σ is a most general unifier (mgu) for $\sigma L_0 = \sigma L_i$ $(1<i<n)$ where $\{L_0,L_1,\ldots,L_n\} \subset C$. If $l = \langle C,\sigma\rangle$ is an F-link, the clause σC is the factor of C, formed by l.

A resolution link (R-link) is a 5-tuple $\langle C,M,D,L,\sigma\rangle$ where C,D are clauses, $M \in C$ and $L \in D$ are literals such that $\sim\sigma M = \sigma L$ and σ is most general.
A paramodulation link (P-link) l is a 6-tuple $\langle C,M,\alpha,D,N,\sigma\rangle$ where Cand D are two distinct clauses, M is a non-equality literal in C, N is an equality literal of the form s = t in D, α is an admissible term access function for M and σ is a mgu for $\sigma\alpha(M) = \sigma s$.

A subsumption link (S-link) is a 4-tuple $\langle C,M,D,L\rangle$ where C,D are clauses, $M \in C$ and $L \in D$ are literals such that L and M are unifiable. These links are used to compute the subsumtion relation as discussed in section 6.3.3.

T-Links and PC-Links are defined below.

Tautologies

A clause is a tautology, iff

(1) it contains a literal of the form $t \stackrel{T}{=} t$, where T is some equational theory or
(2) it contains two complementary literals.

We classify tautologies according to the way they are generated:

Type 1:
A tautology is generated by an application of a substitution to a clause (this happens on factorisation, resolution and paramodulation).

Example:

The application of the factoring-substitution {x/a y/a} to the clause C = ⟨~Px, Py, Qxa, Qay⟩ results in a factor of C, ⟨~Pa, Pa, Qaa⟩ which is a tautology.

Type 2:
The two complementary literals in a tautology stem from distinct parent clauses and are made complementary by the application of a substitution to both parent clauses (this happens on resolution and paramodulation).

Example:
Resolution with the parent clauses ⟨Px, ~Qxa⟩ and ⟨Qay, Py⟩ results in the resolvent ⟨Pa, ~Pa⟩ which is a tautology.

Type 3:
The tautology is generated by replacing a subterm of a literal in a clause and by applying a substitution to the clause (this happens only on paramodulation).

Example:
Paramodulation with the parent clauses ‹~Pf(f(a)),Pa› and ‹f(f(x)) = x› results in the paramodulant ‹-Pa, Pa› which is a tautology.

Type 4:
The tautology is generated by inserting complementary equal literals in a clause which stem from two distinct clauses and are made complementary equal by subterm replacement and application of a substitution to both clauses (this happens only on paramodulation).

Example:
Paramodulation with the parent clauses ‹~Pf(f(a))› and ‹f(f(x))=x, Pa› results in the paramodulant ‹~Pa, Pa› which is a tautology.

Tautology detection

The different types of tautologies are now analyzed in order to prevent their generation during the search for a proof.

Type 1 - Tautologies:

Tautologies that are generated in this way are detected using socalled Tautology Substitution Links:
(DEF) A tautology substitution link (TS-link) is a triple ‹C,L,K› where C is a clause and L and K are complementary unifiable literals in C, or is a triple ‹C,s,t› where s and t are unifiable terms and P(s,t) is a literal in C and P is a reflexive predicate (or ~ P(s,t) is a literal in C and P is an irreflexive predicate).

Subsequently we assume connection graphs to be fully TS-connected, i.e. all TS-links are set.

(DEF) An F-link $\langle C,\sigma\rangle$ or a
R-link $\langle C,M,D,N,\sigma\rangle$ or a
P-link $\langle C,M,\alpha,D,N,\sigma\rangle$ or a
P-link $\langle D,N,\alpha,C,M,\sigma\rangle$

is τ_1-redundant iff there exists a

TS-link $\langle C,L,K\rangle$ or a TS-link $\langle C,s,t\rangle$
such that

$L \neq M$, $K \neq M$ $(Pst \neq M)$ or $\sim Pst \neq M$
and $\sigma|L| = \sigma|K|(\sigma s = \sigma t)$.

Type 2 - Tautologies

Tautologies of this type are detected on the basis of socalled Tautology Resolvent Links:

(DEF) A tautology resolvent link (TR-link) is a quadruple $\langle C,L,D,K\rangle$ where C and D are distinct clauses and $L \in C$ and $K \in D$ are complementary unifiable literals.
In a fully TR-connected connection graph each pair of complementary unifiable literals in distinct clauses is connected with exactly one TR- or R-link.

Subsequently we assume connection graphs to be fully TR-connected.

(DEF) An R-link $\langle C,M,D,N,\sigma\rangle$ or a
P-link $\langle C,M,\alpha,D,N,\sigma\rangle$ or a
P-link $\langle D,N,\alpha,C,M,\sigma\rangle$

in a connection graph is τ_2-redundant, iff there is an

R-link $\langle C,L,D,K,\Theta\rangle$ or a
TR-link $\langle C,L,D,K\rangle$

such that

$$L \neq M, \quad K \neq N \quad \text{and} \quad \sigma|L| = \sigma|K|.$$

Type 3 - Tautologies

We introduce another new link type to detect tautologies of this type:

(DEF) A tautology paramodulant link (TP-link) is a 5-tupel $\langle C, M, \alpha, D, s=t \rangle$, where C and D are distinct clauses, M is a non-equality literal in C, s = t is an equality literal in D and α is an admissible term access function for M, such that $\alpha(M)$ and s are unifiable terms.

In a fully TP-connected connection graph each side of an equality literal is connected to a subterm of a non-equality literal in another clause by exactly one P- or TP-link, whenever the side of the equality and the subterm are unifiable.

Subsequently we assume connection graphs to be fully TP-connected.

(DEF) A P-link $\langle C,M,\sigma,D,s = t,\theta \rangle$ in a connection graph is τ_3-redundant, iff there is a P-link $\langle C,N,\alpha,D,t = s,\theta \rangle$ or a TP-link $\langle C,N,\alpha,D,t = s \rangle$ such that M and N have opposite signs and the same predicate symbol, $\sigma|M| \overset{\alpha}{=} \sigma|N|$ and $\sigma t = \sigma\alpha(N)$.

$K \overset{\alpha}{=} L$ for some literals K and L, iff L = K except for the subterms of L and K given by α.

Type 4 - Tautologies

It is characteristic for tautologies of this type that the two complementary equal literals are the paramodulated literal and a literal stemming from the equality clause.

(DEF) A P-link $\langle C,M,\alpha,D,s = t,\sigma\rangle$ in a connection graph is τ_4-redundant, iff there exists a literal $N \varepsilon D$ different from $s = t$ such that M and N have opposite signs and the same predicate symbol, $\sigma|M| \overset{\alpha}{=} \sigma|N|$ and $\sigma t = \sigma\alpha(N)$.

Remark

The detection of type-4 tautologies involves a search for the literal $N \varepsilon D$. This problem could be solved by the introduction of a new class of links (similar to P-links), which connect one side of an equality with a subterm of a literal in the same clause. Using these links the literal $N \varepsilon D$ could be found without search. We reject this solution: As in the case of P-links, the number of those links in a connection graph would be very large, so that the cost of generating, inheriting and storing these links exceeds the cost of the search for a specific literal in a clause. Note that this argument does not hold for TP-links, since the number of TP-links compared to the number of P-links in a connection graph is very small indeed.

Tautology Redundant Links

Now in order to avoid the generation of tautologies, we say:

An F-, R- or P-link l in a fully TS-, TR- and TP-connected connection graph is tautology redundant (τ-redundant), iff l is τ_i-redundant for some $i \varepsilon \{1,2,3,4\}$.

Theorem 1
A clause generated by an F-, R- or P-link in a fully TS-, TR- and TP-connected connection graph is a tautology, iff l is a τ-redundant link.

Theorem 1 gives a necessary and sufficient condition to detect links which generate tautologies by means of TS-, TR- and TP-links.

The actual implementation uses this information to avoid the generation of tautologies and extends the Connection Graph Proof Procedure by a τ-link reduction rule:

(1) For each clause in the initial graph create all TS-links to guarantee the fully TS-connected property of the graph.

(2) After creation of the initial graph, remove each τ-redundant F-link from the graph. Recolour each τ-redundant R- or P-link to a TR- or TP-link to guarantee that the graph is still fully TR- or TP-connected.

(3) After generation of a factor, remove the associated F-link from the graph. After generation of a resolvent or paramodulant, recolour the associated R- or P-link to a TR- of TP-link (R- and P-links must not be removed to guarantee the fully TR- and TP-connected property of the graph). In addition generate all TS-links for the factor, resolvent or paramodulant.

(4) After generation of a resolvent or paramodulant, remove each τ-redundant F-link from the graph, which is attached to the resolvent of paramodulant. After generation of a factor, resolvent or paramodulant, recolour each τ-redundant R- or P-link to a TR- or TP-link, which is connected to a literal in the factor, resolvent or paramodulant.

(5) After recolouring an R- or P-link, check the link's parent clauses for purity.

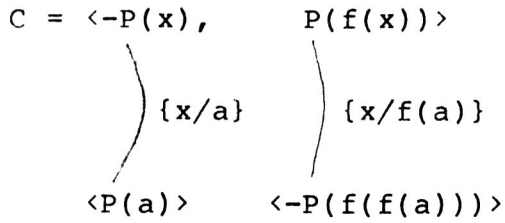
Pure Clause Generation

A clause is pure, iff it contains a literal which is not connected by an R- or P-link.
A link connecting a literal in a factor or resolvent is either inherited from a link connecting its parent literal or is newly created and connects a literal in its parent clause.

In order to check for purity with a given literal L in clause C and a substitution σ of an F- or R-link, we have to search C for literals K_i, which are unifiable with σL.

Given a literal L, we find some of the literals K_i with the help of the TS-links. Unfortunately not all literals K_i are given by these links.

Example:



Since -P(x) and P(f(x)) are not unifiable, we have no TS-link between these literals. But we can connect the descendant P(f(a)) of P(f(x)) in C to -P(x) in C. The problem is to take into account the renaming of variables in factors and resolvents, which is done by introducing socalled parent connector links.

Pure Clause Detection

(DEF) A parent connector link (PC-link) is a triple <C,L,K> where C is a clause, L and K are literals in C which are unifiable only after renaming, i.e.

$$\begin{aligned} &\text{ALL } \theta.\ \theta|L| \neq \theta|K| \quad \text{and} \\ \text{EX}\mu\ \text{EX}\nu.\ &\mu \text{ is variable renaming of } K \\ &\text{and } \nu|L| = \nu\mu|K| \end{aligned}$$

We assume connection graphs to be fully PC-connected and the TS-links and the PC-links denote the literals K in a clause C which are unifiable (at least after renaming) with a given literal in C.

Purity Redundant Links

Purity redundant links are defined as:

(DEF) An R-link $\langle C,M,D,M,\sigma\rangle$ or an F-link $\langle C,\sigma\rangle$ is purity redundant (π-redundant), iff

EX $L \varepsilon C-\{M\}$
(resp. $L\varepsilon C$ in case of an F-link)
such that
(1) ALL R-LINKS $\langle C,L,C_i,L_i,\sigma_i\rangle$
$\sim$EX θ. $\theta\mu\sigma|L| = \theta|L_i|$
and
(2) ALL TS-LINKS and PC-LINKS $\langle C,L,K\rangle$
$\sim$EX θ. $\theta\mu\sigma|L| = \theta|K|$
where μ is a variable renaming for C.

Theorem 2
A clause generated by an F- or R-LINK l in a fully TS- and PC-connected connection graph is pure, iff l is a π-redundant link.

Purity Link Reduction

Theorem 2 gives a necessary and sufficient condition which is used in the MKR-Procedure to avoid the generation of pure clauses. The Proof procedure is extended by a π-link reduction rule:

(1) For each clause in the initial graph and for each resolvent or factor create all TS- and PC-links to ensure that the graph is fully TS- and PC-connected.

(2) After creation of the initial graph and after each creation of a factor or resolvent, remove from the graph each τ-redundant link, which is connected to a literal in a clause of the initial graph or in the factor of resolvent.

If τ-link reduction is applied, instead of removing π-redundant R-links from the graph, "recolour" these links to TR-links to ensure that the graph is still fully TR-connected.

(3) After removing or recolouring an R-link, remove or recolour each π-redundant link, which is connected to a literal in a parent clause of the removed or recoloured R-link.

Pure Clauses under Paramodulation

The extension of π-link reduction to P-links is of little practical value, since the purity principle is so rarely applicable in paramodulated connection graphs. Consider for instance axiom clauses like ⟨~x>0, successor (predecessor (x)) = x⟩ or ⟨~f(x) = f(y), x = y⟩.

In paramodulated connection graphs each variable representing one side of an unnegated equality in a clause C is connected to each term in a clause different from C. Hence clauses like those given in the example prevent each other from becoming pure. These examples do not represent artificial problems but types of axioms which occur very often in axiomatizations with equality. The restricted success compared to its great expense does not seem to justify extending π-link reduction to paramodulated connection graphs.

However, these arguments are not specific to paramodulation: If we incorporate equality by a predicate symbol E and do not use paramodulation, the substitution axioms for predicates (for instance for a unary predicate P:

⟨~E(xy), ~P(x), P(y)⟩)

guarantees that no literal with predicate symbol P becomes pure unless the substitution axiom itself become pure, which is prevented by the axiom of symmetry ⟨~E(xy), E(yx)⟩.

Eliminating the Search for R-link Inheritance

A further usage of TS- and PC-links is the following: After forming a resolvent or factor it is necessary to search for literals in the resolvent (or factor), which are unifiable with a literal in the parent clauses. For such pairs of literals an R-link is added to the graph. We call such an R-link a parent connection. (The original procedure of [KO75] requires a search and connection for both parent clauses. It is easy to verify, that this approach results in the multiple inheritance of R-links or - even worse - in the inheritance of R-links already removed from the graph).

The problem consists in the fact that the search for parent connections can be very expensive because we have to perform (x + y) * (x + y) tests for unifiability, where x and y are the number of literals in a resolvents parent clauses. This expensive search can be dispensed with in fully TS- and PC-connected graphs since for each literal L in a resolvent or factor the unifiable literal K in its parent clause C is given by a TS- or PC-link ‹C,L,K›.

Generation Rules for Links in Extended Connection Graphs

During the construction of an initial connection graph, each time two literals L and K in a clause C are unifiable, a TS-link ‹C,L,K› is added to the graph. Each time two literals L and K in a clause C are unifiable after renaming, a PC-link ‹C,L,K› is inserted into the graph. TR- and TP-links are generated in the initial graph by application of the τ-link reduction rule to the R- and P-links of the initial graph.

6.3.5 Terminator

Todays theorem proving calculi may be classified into those that start with a given set of logical formulas and create new formulas by the application of certain deduction rules like resolution [RO65], paramodulation [WR73], natural deduction rules [BL77], [NE74] etc, until the theorem (forward reasoning) or a refutation (backward reasoning) has been derived.

Recently new calculi of a different kind like Andrews' mating calculus [AN81] or Bibel's matrix calculus [BI81] have been developed which initially do not deduce any new formulas, but only test certain path conditions ensuring satisfiability or unsatisfiability of the initial formula set. Only if this test fails may the need arise to copy certain formulas. Kowalski`s connection graph proof procedure [KO75] in its original formulation is of the first kind: A connection graph consists of the nodes labeled by clauses in conjunctive normalform and links (connections) between complementary unifiable literals representing possible resolution steps. A deduction is performed by the selection of a link, creating the corresponding resolvent, inserting it into the graph and deleting the selected link, potentially causing further deletions of links and clauses.
This proof procedure can be transformed into a calculus of the second kind using an idea originally proposed by S.Sickel [SI76]: instead of adding resolvents to the graph, the search for a proof is essentially done on the initial graph by "walking along" the links until a refutation has been found, thus "unrolling the graph" [SI76].
The method proposed here is very much in that spirit but used for a very special - albeit important - case only: If the clause set is unit refutable, i.e. the empty clause can be derived by successive resolution steps with one-literal clauses, the clause graph has to contain a special subtree (refutation tree or terminator situation) which just represents this chain of unit resolutions.

Examples:

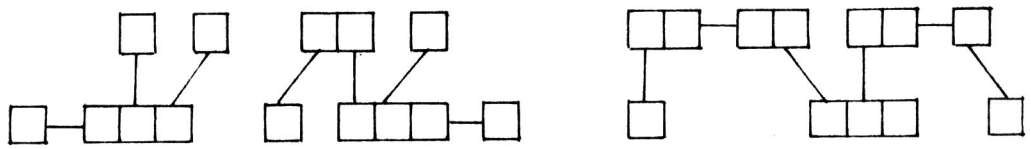


Fig.1 a) b) c)

Every box in fig.1 represents a literal, a string of boxes is a clause and complementary unifiable boxes (literals) are connected by a link. If all unifiers attached to the link in fig. 1a) are compatible, this represents a one-level terminator situation, since it immediately allows for the derivation of the empty clause. Similary fig. 1b) represents a two-level terminator situation (a kernel which is connected to a one-level terminator situation) and fig. 1c) a 3-level terminator situation if all unifiers are compatible.

The terminator component of the MKRP system, which detects configurations of the above kind, is used in two ways: first it acts like a simple and fast theorem prover and is activated on the initial connection graph. If it fails, the full machinery of the Logic Machine is activated. Secondly it is used to overcome the problem that the heuristic selection functions have the very limited horizon of one step ahead, since the computation of a further n-level look ahead for n>2, is so prohibitively expensive that it outweighs the advantage. For that reason the Terminator is used as a different n-level look ahead technique, which checks at tolerable costs if there is a proof within a predefined complexity bound. This use is the no-loop-requirement of [SI76] and is akin to the n-level-look-ahead heuristic proposed by [KO75]. It is one of the main sources for the success of the current system.

A clause graph is a refutation tree (a terminator) if:

- Every literal of a clause is attached to exactly one link.
- The unifiers of the links are compatible.
- The graph is cycle free.

The following important result is known about refutation trees [HR78]:

Theorem:
A unit refutable clause set S is unsatisfiable if there exists a refutation tree for the factored set S.
Of course the knowledge of the existence of such a refutation tree is of little practical use unless a fast method for extracting it from a given graph is known. An exhaustive and unsophisticated search for such a terminator configuration is prohibitively expensive in large graphs, hence an efficient extraction of a refutation tree (if it exists) from a given graph is the task of the presented TERMINATOR algorithm.

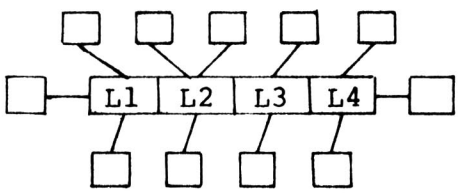
The Algorithm

The first attempt to implement an N-level terminator algorithm used a recursive technique [SS81]: A non-unitclause C was selected and examined for a one-level situation. If the first test failed, the algorithm was called recursively for other non-unitclauses connected to C, trying to find a chain of unit resolutions which resolves away all literals exept that one connected to C, and the one-level test for C was performed once again taking advantage of this new information. This algorithm considerably increased the overall strength of the system, since the interaction between the TERMINATOR and the MKR-Procedure essentially amounts to a bidirectional search for a refutation. Unfortunately the algorithm itself was rather inefficient, because no results of former TERMINATOR calls had been stored,

and therefore the same part of the clause graph had to be examined again and again. Hence only the N = 1 case was ever within practical limits.

Whereas the old algorithm worked from inside the refutation tree to the leaf nodes (unitclauses), the new algorithm works just in the opposite direction and stores every information it has onces generated for later use. We shall present the working of the algorithm using a few examples.

The input clauses are divided into the set of unitclauses (UNITS) and the set of non-unitclauses (NON-UNITS). NON-UNITS are sorted according to increasing literal number. For each element C of NON-UNITS of which every literal except at most one is connected to at least one element of UNITS a compability test is performed. An example will illustrate this test:
Suppose C has four literals: C = ‹L1 L2 L3 L4›. Each literal Li may be connected to a set of unitclauses complementary to Li and let U be the set of unifiers associated with these connections (there may be more than one unifier associated with each link):



We start with the first literal and compute the set of "merged" unifiers:

$U_{12} = U_1 * U_2 = \{\sigma : \sigma = \sigma_1 * \sigma_2,\ \sigma_1 \varepsilon U_1,\ \sigma_2 \varepsilon U_2\}$

where $\sigma_1 * \sigma_2$ is a most general merge substitution (resp.unifier, see below) of σ_1 and σ_2 . Similarly we calculate

$U_{123} = U_{12} * U_3$ and $U_{1234} = U_{123} * U_4$.

If $U_{1234} \neq \phi$ then every element of U_{1234} allows to resolve away the whole clause, i.e. a proof is found. If $U_{1234} = \phi$, but $U_{123} \neq \phi$ we apply every substitution of U_{123} to literal L4 thus generating new unitclauses and insert them into the UNITS-List. With this step we deduce in fact new formulas, but we only use them as a compact representation of the merged unifiers with two additional advantages:

1. A very simple and fast subsumption test between the unitclauses allows to detect if the result of a unit resolution chain is an instance of the result of another chain. In general the deletion of subsumed unitclauses prunes the search space considerably.
2. The treatment of the input unitclauses is just the same as the treatment of the deduced onces, which simplifies the data- and control structures.

If $U_{1234} \neq \phi$ we create new unitclauses in the same way applying $U_{12}{}^*U_4$ to L3, $U_1{}^*U_{34}$ to L2 and U_{234} to L1. The links of the initial graph provide now the information to attach all the new unitclauses to complementary literals in other clauses, (respectively other literals in the same clause, if it is selfresolving).

The clauses in NON-UNITS are examined several times, constantly producing new unitclauses, either until the proof is found or until a certain boundery value is exceeded. If a terminator situation is found, the datastructures we use to represent the units and the link allow for an immediate extraction of the refutation tree, which is then used to generate an ordinary resolution proof.

The TERMINATOR is called from the MKR-Procedure at the beginning of the search for a proof and, if it does not find an immediate proof it is called again each time a certain number of steps has been performed by other components of the system, hence the overall behaviour amounts to a tightly controlled and

sophisticated bidirectional search. In particular the other link selection mechanism in the system prefer those resolution and paramodulation steps which are expected to transform a non-unit refutable clause graph into an unit refutable one.

A second application of the TERMINATOR is the generation of unitclauses, for instance the rewrite component of the system can decide that a specific unit equation might be usefull as a rewrite rule and charges the TERMINATOR to deduce, if possible, the desired equation from a conditional equation.

The Compatibility Test

The bottleneck of the terminator algorithm is the compatibility test:
We have a set A with n substitutions and a set B with m substitutions. Now the task is to unify each substitution in A with each substitution in B, i.e. to compute the most general merge substitution for each such pair.

A unifier for two substitutions σ,τ and λ is a substitution such that

$$\lambda.\sigma = \lambda.\tau$$

$\sigma*\tau := \lambda.\sigma = \lambda.\tau$ is called the merge substitution of σ and τ.
There are fast unification algorithm known for the unification of terms [RO71], [BA73], [MM79], the most recent are even linear [PW78], [KK82]. The unification of substitutions was first investigated in [VV75], better and more efficient algorithms are surveyed in [HE83]. But for $|A| = n$ and $|B| = m$ they still have to perform n*m such operations and in many applications this number may be greater than 10^5, i.e. the known methods are out of the question. For that reason a very fast merging algorithm for large sets of substitutions was developed which exploids special conditions present in the terminator situation. The problem is to

find a representation for the substitutions such that there is a direct access from each substitution σ in set A to those which are compatible with σ in set B.
The working of the algorithm will be illustrated with the following examples.

Example 1

```
<~Pxy   ,   ~Pyz   ,   Pxz>
  |           |          v,x,y,z are variables
<Paf(v)>    <Pf(b)c>
```

set A: {(x/a, y/f(v))} n = 1
set B: {(y/f(b), z/c)} m = 1

Since only links between unitclauses and non-unitclauses are considered, we know that the single literal of the unitclause is not present in the resolvent after resolution upon such a link. Therefore only the substitution components for variables of the non-unitclause are relevant for the resolvent, provided the unifier is in normal form [HE83] and the variables in different clauses are disjoint. Since these conditions can be ensured, it is admissible to discard all other substitution components. Because of this it is possible to represent the remaining components in a table using a fixed ordering of the variables occurring in the non-unitclauses. This representation will be called a T-representation.

For example 1 we obtain:

	x	y	z
A	a	f(v)	z
B	x	f(b)	c

In this case it is easy to see that the unification of the corresponding terms in the table is sufficient to get a most general instance of the substitutions.

In this example it is:

x	y	z	or in standard representation:
a	f(b)	c	(x ← a, y ← f(b), z ← c)

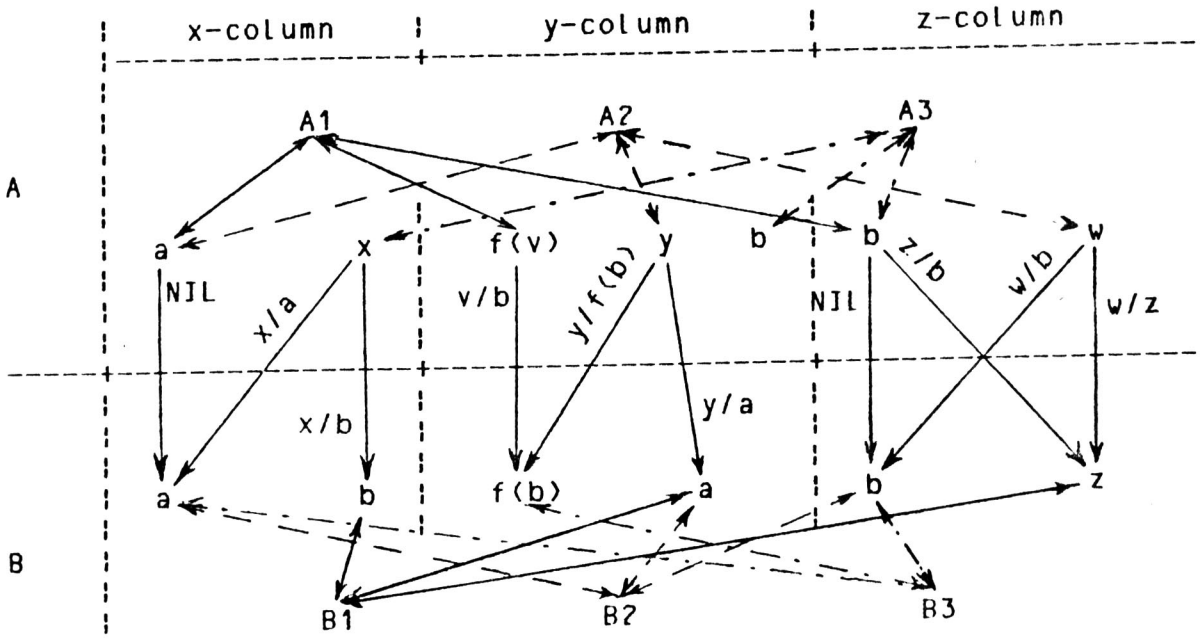
The main problem, however is the merging of two non-singleton sets A and B of substitutions.
Suppose we have two lists A and B with three elements in each list:

Example 2 T-representation

A :	x	y	z	B :	x	y	z	v,w,x,y,z are variables
A1 :	a	f(v)	b	B1 :	b	a	z	
A2 :	a	y	w	B2 :	a	a	b	n = 3
A3 :	x	b	b	B3 :	a	f(b)	b	m = 3

The representation is then changed a second time: Equal terms occurring in different substitutions but in the same column of the T-representation are stored only once. The substitutions themselves are represented by pointers to their terms. These terms have also pointers back to the substitutions in which they occur; and from the terms of group A links to every unifiable term in the same position of group B are generated. In addition the unifiers for these single terms are attached to the links. We call this representation the P-representation.

P-representation of example 2:



The complexity of the merging algorithm is a function of the task of generating the links between unifiable terms, which can be further improved by grouping the terms into lists of variables, constants, ground terms (without variables) and composed terms to exploit the fact that some single-term-unifiers are trivial (variables with constants etc.) whereas others are impossible (constants with ground terms).

On the basis of the P-representation it is now easy to extract for a substitution σ in group A all compatible ones in group B. For every term of σ and every link attached to this term: attach the single-term-unifier corresponding to this link at every substitution of group B in which the term pointed to by the link occurs:

For example 2 we obtain:

	B1	B2	B3
from A1:	(z/b)	(NIL) (NIL)	(NIL (v/b) (NIL)
from A2:	(y/a) (w/z)	(NIL) (y/a) (w/b)	(NIL) (y/f(b)) (w/b)
from A3:	(z/b)	(x/a) (NIL)	(x/a) (NIL)

All those substitutions in B with exactly p compatible single-term-unifiers, where p is the length of the T-representation (=number of variables in the non-unit clause), are compatible with σ. For example 2 we obtain three compatible pairs of substitutions:

	x	y	z
A1 * B3	a	f(b)	b
A2 * B2	a	a	b
A3 * B3	a	f(b)	b

Although the generation of the single-term links is not linear in general, the algorithm shows a linear (in n) behaviour in most practical examples, i.e. instead of n * m operations only n operations are performed on average.

Summary

With the terminator algorithm two main goals have been achieved:

1. Actually nothing has to be done twice, because all relevant intermediate results (merged substitutions) can be stored without requiring too much space.
2. The compatibility test is enormously reduced in complexity, allowing the fast unification of millions of pairs of substitutions.

These are the main advantages compared to similar methods for the extraction of refutation trees (respectively graphs), like S. Sickel's graph unrolling [SI76] or Chang and Slagle's usage of rewrite rules [CS79]. Although their calculi are complete for non-unit refutable clause sets as well, their practical usefullness is questionable, unless their efficiency problems are solved.

The implementation of the terminator algorithm has considerably increased the performance of the MKR-Procedure. For instance we have solved SAM's Lemma, a famous problem in automated theorem proving, which was first proved with an interactive theorem prover [GOSB69], and later by the automated theorem prover at Argonne National Lab. It had to deduce 22000 clauses before finding a proof [COW76]. The TERMINATOR generated only 190 unitclauses until the proof was found [OH82].

6.3.6. Heuristic Selection Criteria

In contrast to the global search strategies and global heuristics some heuristic selection functions are based on local syntactical information about the graph or the resolvent (paramodulant) respectively.

Initially we experimented with about 20 different heuristic features, where each feature attaches a certain value to every link k in G_i. G_i is the present graph, G_{i+1} is the resulting graph after resolution upon link k and Res is the resolvent resulting from this step:

(i) Sum of literals in G_{i+1}
(ii) Sum of clauses in G_{i+1}
(iii) Sum of links in G_{i+1}
(iv) Average length of clauses in G_{i+1}
(v) Average sum of links on literals in G_{i+1}
(vi) Sum (resp. average sum) of constant symbols in G_{i+1}

(vii) Number of distinct predicate symbols in G_{i+1}

(viii) Number of distinct variables in G_{i+1}

(ix) Sum of literals of Res

(x) Sum of links of Res

(xi) Sum of constant symbols in Res

(xii) Sum of distinct variables in Res

(xiii) Number of distinct predicate symbols in Res

(xvi) Term complexity of Res

(xv) Minimum of links on literals in Res

(xvi) Complexity of the most general unifier σ attached to link k

(xvii) Age of Res

(xviii) Degree of isolation of Res

(xix) Degree of isolation of the parents of Res.

The problem is that although each heuristic feature has a certain worth, the cost of its computation can by far outweigh its potential contribution. Also it may not be independent of the other heuristic features; for example features (xi) and (xii) both measure the "degree of groundness of Res", but in a different way. Similarly the values for Res and for G_{i+1} are not independent for certain features (e.g. xiii and vii). Also there are the two problems of finding an appropriate metric for each feature and to decide upon their relative worth in case of conflict with other features.

Originally the information contained in the heuristic features was entered in two different ways: certain facts (e.g. decreasing size of the graph) had absolute priority over all other information (see also the merge feature of TT in [DA78]). Most of the information of the other features however was expressed as a real number in [0,1], where we experimented with several (linear, nonlinear) metrics [SS81]. This information was then entered in a weighted polynomial and the resulting real number (the priority value) expressed the relative worth of the particular link and was attached to each link.

The system has been designed such that heuristic features can easily be added and deleted and after more than two years of experimentation the system stabilized with the following set of features.

1. Complexity of the Graph
1.1 FCLSUM = (Σ of clauses of G_{i+1}) minus (Σ of clauses of G_i)
1.2 FLINKSUM = (Σ of links of G_{i+1}) minus (Σ of links of G_i)
1.3 FCANCEL = {P/P is predicate symbol occurring in G_{i+1}}

2. Complexity of the Resolvent
2.1 FAGE = Age of Res
2.2 FLITSUM = Sum of literals in Res
2.3 FTERM = Term complexity of Res
2.4 FRESISO = Degree of isolation of Res

3. Complexity of the Parents of Res
3.1 FPARISO = Degree of isolation of the parents

These features are used to influence the actual derivation in the following way: all steps that lead to a reduction in the size of the graph have absolute priority and are immediately executed. That is, every link which leads to a graph with fewer clauses or fewer links or both is put into a special class, which is executed before any further evaluation takes place. The decision whether or not a link leads to a reduction is based on information from the reduction module and is optionally taken for every link or for the active links only. Note that the reduction in the size of the graph may lead to further deletions, hence a potential snowball effect of deletions is carried out immediately which accounts for one of the main sources of the strength of the system.

The realisation of the paramount importance of those features having absolute priority (and their former costly computation) has led to a different implementation now: we no longer compute a

weighted polynomial as in the original heuristic module [BES81]. Instead the features 1.1 and 1.2 are removed from this module and the selection module selects and executes the corresponding steps prior to everything else (see section 6.4.).
The heuristics 2.4. and 3.1. are completely abandoned, whereas the remaining features 2.1., 2.2. and 2.3. are now realised in the following way: each list of links to be executed (see section 6.4.) is sorted such that for the resolvent Res(i) resulting from the ith link in that list:

(i) either the number of literals of Res(i) is smaller than the number of literals of Res(i+1) (unit preference)
(ii) or (number of literals of Res(i)) = (number of literals of Res(i+1) and the term complexity of Res(i+1) is less than the term complexity of Res(i).
(iii) or term complexity of Res(i) = (term complexity of Res(i+1)) and the age of Res(i) is less than the age of Res(i+1).

Now always the first element of a list is executed first.
The age of a clause is computed by the following formula:

$$FAGE := (1 - AGE/D_{max})^{3/2}$$

where Age: max{Age(Parent1), Age(Parent2)} + 1
D_{max}: user defined maximally admitted depth of derivation

This feature is mainly used to avoid "infinte holes" in the search space, since after too many steps in one direction, the age of the resolvents become too high and different steps are tried, i.e. after a while a "breadth-first" strategy is enforced again.

The term complexity of a term is computed as:

$$\text{FTERM} := \begin{cases} 1 & \text{if no nested terms in Res} \\ 1/2\left[1-\left(1/n \sum_{i=1}^{n} (s_i)^{3/2}\right)/S_{max}^{3/2}\right] & \text{otherwise} \end{cases}$$

where: s_i: maximal nesting depth of i^{th} term in Res
S_{max}: user defined maximally admitted depth of nesting
n : number of terms in Res

The term complexity of a clause is the average of the term complexity of the terms occurring in it.

6.3.7. Refinements

Although the traditional research paradigm of automated theorem proving was strongly rejected right from the very start of this project, some of the traditional results turned out to be not entirely without value:

A link in a connection graph represents a potential resolvent and the complex and expensively computed selection functions, which pick the most promising candidate, have to examine every link: in a realistic setting there may be several thousands in a graph to be examined again and again for each step.

But suppose the system is to simulate a unit-refutation, i.e. a derivation of the empty clause, where at least one of the parents of every resolution step is a unit clause. In that case the selection functions are allowed to pick only those links that connect at least one unit clause.
If by appropriate on-off switches only those links emanating from unit clauses are declared active and all other links passive the selection functions have to be computed only for the small fraction of the graph that is declared active.
It should not be too difficult to see that by an appropriate

setting of the on-off switches every traditional refinement [LOV78] can be simulated.
Seen from this point of view a traditional refinement acts like a cone of light that illuminates certain parts of the graph and shades those that are (hopefully) irrelevant for a proof - and it is a comforting fact to know that if the refinement is complete, the shaded parts are at least theoretically irrelevant.

On the other hand it is a wellknown fact that the "shaded parts" of blocked resolutions represent "garbage and gold" alike: hence the strategic information of the system overrides the information of every particular refinement. Only if nothing better in known does it take the refinement information into account and it should not be necessary to say that the complex interplay of the many sources of information that are taken into account (see 6.4.) prevent the overall deduction from being standard.

6.3.8. Clause Reduction

Deduction steps, which in a certain sence reduce the complexity of a clause, are called clause reduction.

Term Rewriting

A given equation l=r can often be turned into a directed equation l=>r such that the complexity of the term l is greater than that of r. For example x * 0 = 0 is such an equation. Directed equations are computationally very useful: if $\hat{l}$ is some subterm occurring in a literal L_0 and $\hat{l}$ is an instance of l, i.e. $\exists\sigma$ such that $\hat{l} = \sigma l$, then replacing $\hat{l}$ in L by σr does not change the truth value of L_0 if l=r is assumed valid. If $L_0 \rightarrow L_1$ denotes such a replacement, we say a given set of directed equations is Nötherian if there is no infinite sequence $L_0 \rightarrow L_1 \rightarrow L_2 \rightarrow \ldots$ The set of equations is confluent if for every L_i, L_k with $L_0 * L_k$ there exists L_k such that $L_i * L_n$ and $L_k * L_n$ (see [HO80] for a survey).

If the initial set of clauses contains a set of equational unit clauses, which are confluent and Notherian, they can be used to compute normal forms and hence are removed from the initial clause set: they are only used to compute the normal forms in the initial set and subsequently in every deduced clause.

Unfortunately most sets of equational unit clauses are not confluent and Notherian, but they may still be used for rewriting purposes (see 6.4.).

The essential difference between a rewriting step and a paramodulation step is the unification requirement: in order to rewrite the term $\hat{l}$ in a literal L using the equation $l>r$ we require $\hat{l}=\sigma l$ and replace $\hat{l}$ by σr. But in order to paramodulate the term $\hat{l}$ in L using $l>r$ we have the weaker requirement of $\sigma\hat{l} = \sigma l$ in order to replace $\hat{l}$ by σr in L.

For example P(f(x,y),z) can not be rewritten with the equation f(x,0)=>0, but can be paramodulated to P(0,z). P(f(1,0),z) can be rewritten to P(0,z), hence rewriting does not change the value of a literal, but is only used to manipulate its syntax.

Conditional Term Rewriting

Single equations (units) do not occur too often in clause sets: usually an equation requires some condition to be true before it can be applied. For example for all x,y: if P(x) and Q(f(y)) then g(x,y)=C where P and Q state some properties of x and y such that g(x,y)=C holds.

The clause form is

$$\langle \sim Px,\ \sim Q(f(y)),\ g(x,y)=C \rangle$$

and such clauses are treated in a special way:
if the literals preceeding the equation can be resolved away with at most a predefined number of steps then these steps are executed prior to anything else and $\sigma[g(x,y)=C]$ is used as a

rewriting equation. Here σ is the substitution resulting from these resolution steps(see 6.4.2.9).

Literal Rewriting

A very similar kind of "rewriting" can be done with two-literal clauses (instead of equations). For example:

for all x: P(f(x)) implies P(x)

can be used to reduce a literal like

P(f(f(f(0)))) to P(0).

Such two-literal clauses are marked by the user (see 6.4.2.7) and all possible reduction steps are performed with a high priority.

Conditional Literal Rewriting

An assertion

for all x,y: P(x) and Q(f(x)) and P(f(x)) implies P(x)

has the clausal form

‹~Px, ~Qf(x), ~Pf(x), Px›

An actual implementation will make sure that these literals are not randomly resolved upon, but that the first two literals are resolved away prior to everything else and the remaining two literals are then used as a literal rewriting rule.

Demodulation

Suppose several rewriting steps can be sequentially applied to a literal L in a clause C (with some abuse of notation):

$C_0 \rightarrow C_1 \rightarrow C_2 \rightarrow \ldots \rightarrow C_n$

Although not complete theoretically and often dangerous practically it may nevertheless be useful sometimes to not save the intermediate results $C_1, C_2, \ldots, C_{n-1}$.

If these clauses are erased once C has been deduced, the replacement of C_0 by C_n is called demodulation [WR67], or C_0 is demodulated to C_n.

Demodulation is user controlled by appropriate options (see 6.4.).

Replacement Resolution

Suppose we have a unit clause ‹Pa› and a clause ‹~Pa,Qx›. The resolvent is ‹Qx›, which subsumes its second parent clause. The result of the resolution step with the subsequent subsumption could also be obtained by simply erasing ~Pa in ‹~Pa,Qx›. This observation, called replacement subsumption in [RO65] can advantageously be generalized in several ways:

Firstly possible merging of literals can be taken into account, as for example in:
‹Pa,Qb› ‹~Pa,Qb›, i.e. we still may just erase ~Pa.

Secondly the instantiation process can be taken into accont, as for example in:
‹Pa,~Px› and ‹~Pa,Qy›, where the second clause is to be replaced by ‹~Px,Qy› (resolution plus subsequent subsumption).
Finally elaborate additional techniques can be employed, as for example in:
‹Pa,Qy› and ‹~Px,Qa,x≠a›. The resolvent is ‹Qy,Qa,a≠a›. The literal a≠a is false in any interpretation and may hence be deleted, i.e. a factor of the final resolvent is ‹Qa›, which could be obtained immediately by erasing Px and x=a in the parent.
The advantage of these replacement rules is that all the intermediate steps are discarded and most importantly: these reduction steps can be carried out prior to everything else, thus taking the burden off the deductive machinery.

This is but one instance of a more general aim: a traditional theorem proving system is confronted with a bewildering number of possible deduction steps, most of which result in trivial symbol manipulations and although these steps are necessary, they

clutter the view for the "essential logical" steps that actually constitue the final proof. The general aim is to remove as many trivial deduction possibilities from the view of the selection mechanism of the Logic Machine and to build them into its hard wired core.

Replacement Factoring

Just as a resolvent may subsume one of its parents, a factor may subsume its parent and can be obtained by simply erasing the appropriate literal of the parent in the first place. Similar generalizations as above are possible.

Merge Resolution

If a clause contains two identical literals they may be merged into one literal without effecting the truth value of that clause. For example ‹Qa,Qa,Pf(x)› becomes ‹Qa,Pf(x)›.

The importance of merging can be seen more clearly in combination with a resolution step:
For example the two clauses ‹~Py,Qa› and ‹Pa,Qa› resolve and merge to the unit clause ‹Qa› (i.e. a shorter clause!). Such a step is called a merge resolution and the realization of the importance of merge resolution steps is - besides the realization of the importance of unit clauses [WR64] - the second major observation in the history of the field [AN68]: merging is the "golden key" in the search for a proof.

6.3.9. T-Unification

Although the problem was mentioned earlier in the theorem proving literature; it has been particularly clear since Robinson's paper in 1967 [ROB67], that substantial progress would be achieved - in fact "a new plateau" - if certain troublesome axioms could be taken out of the database and "built into" the rules of

inference.

These axioms include the equality axioms, partial ordering, (naive) set theory; or they may be simply the associativity, commutativity or idempotency laws.

For certain axioms the resolution process yields a large number of resolvents all of which - although they are syntactically different - have the same meaning in the following sense:

If for example the axioms define multiplication and one axiom states that multiplication is associative

```
f(f(x,y),z) = f(x,f(y,z))
```

the following situation might arise. Suppose some axioms contain strings of multiplied constants, e.g.

```
f(f(f(f(a,b),d)e)g)
```

then resolution with the above formula will eventually produce all possible combinations:

```
f(f(f(a,b),f(d,e)),g)
f(a,f(f(b,d),f(e,g)))
f(f(f(a,b),d),f(e,g)) ... etc.
```

Since they are syntactically different the TP regards them as different formulas, although semantically they denote the same object in any interpretation.

A similar situation can arise for other axioms, and the storage will be gradually filled with redundant clauses of this type. This phenomenon has been referred to as "semantic noise".

However apart from the problem of the "semantic noise", i.e. the fact that a whole equivalence class is unnecessarily computed, there is also the problem that these axioms considerably enlarge the search space, which is demonstrated in [SI75] using again the case of associativity as an example.

Two different approaches towards a solution aiming at the "new plateau" in ATP have emerged so far: either to build the axioms into different inference rules [SL72] or to build the axioms into the unification algorithm [PL72].

The first approach - adopted e.g. by Slagle in [SL72] - appears to be rather weak, since the search space seems to be of the same complexity as before and only a reduction in actual computation time - the least important of all costs - is claimed.

The second approach has been adopted by many workers in ATP, and provided some motivation for the growth of a theoretical field of its own, concerned mainly with equational axioms (see [SS82] for a survey).

Given two terms and an equational theory the unification problem is to find substitutions for the variables such that these terms become equal in the given theory. Suppose now that we have a T-unification algorithm for a given theory T, which solves this problem, then the axioms in T may be removed from the data base provided the set of unifiers has the following property:
- all elements of the set are unifiers (correctness)
- all unifiers are represented by this set (completeness)
- the set satisfies a minimality condition (minimality)

Under these conditions the theorem prover is complete [PL72] and based on this result many unification algorithms have been developed. The following table summerizes the major results that have been obtained for special theories which consist of combinations of the following equations:

A (associativity) $f(f(x,y),z) = f(x,f(y,z))$

C (commutativity) $f(x,y) = f(y,x)$

D (distributivity) D: $f(x,g(y,z)) = f(f(x,y),f(x,z))$

D: $f(g(x,y),z) = g(f(x,z),f(y,z))$

H,E (homomorphism, endomorphism) $\psi(x \circ y) = \psi(x)\psi(y)$

I idempotence $f(x,x) = x$

Abbreviations:

FPA: Finitely Presented Algebras

QG: Quasigroups

AG: Abelian Groups

Hl0: Hilbert's 10 Problem

Sot: Second order terms

Hot: Higher order terms (i.e. > 3 order)

Theory T	Type of T	Unification decidable	recursive	A
∅	1	Yes	Yes	Yes
A	∞	Yes	Yes	Yes
C	ω	Yes	Yes	Yes
I	ω	Yes	Yes	Yes
A+C	ω	Yes	Yes	Yes
A+I	?	Yes	?	No
C+I	ω	Yes	Yes	Yes
A+C+I	ω	Yes	Yes	Yes
D	∞	?	Yes	Yes
D+A	∞	No	Yes	Yes
D+C	∞	?	Yes	Yes
D+A+C	∞	No	Yes	Yes
D+A+I	?	Yes	?	No
H,E	1	Yes	Yes	Yes
H+A	∞	Yes	Yes	Yes
H+A+C	ω	Yes	Yes	Yes
E+A+C	∞	?	?	No

QG	ω	Yes	Yes	Yes
AG	ω	Yes	Yes	Yes
Hl0	?	No	?	No
FPA	ω	Yes	Yes	Yes
Sot, $T=\emptyset$	?	No	-	-
Hot, $T=\emptyset$	0	No	-	-

The column "type of a theory" indicates the cardinality of a minimal set of unifiers and the column "A" says whether or not a type conformal algorithm is known. The condition type conformal is a slight weakening of the minimality condition.
For details see [SS82].
But two main problems remain when building these algorithms into an automated theorem prover. The first one is that these algorithms are only designed for one function symbol (which defines the theory) and constants and variables. For example let f be a commutative function, i.e. f(x,y) = f(y,x) then terms like f(g(a),h(e,y)) are forbidden.
The second unsolved problem is the combination of several such algorithms. The immediate intuitive approach of iteratively invoking does not solve the problem as the following example shows: let f be idempotent and g commutative. Given the terms s = g(f(a,y),g(x,y)) and t = g(g(c,a)y), the s and t are unifiable with σ = {y/a,x/c} but it is not apparent which algorithm is to be invoked.

The following is a short report of what is implemented at the moment and what is planned for the near future:
The unification module contains an algorithm for ordinary term-unification and a fully integrated unification algorithm for associative terms. In addition there are special algorithms and data structures used for the computation in the connection graph.

We also have implemented but not integrated algorithms for the following theories:
associality + commutativity, commutativity, idempotency, commutativity + idempotency and assoc. + comm. + idemp.

The incorporation of the A+C-case into the Markgraf-Karl-Refutation-Procedure is the most pressing task in the future, since many algebraic structures like abelian groups, rings, fields, lattices, sets and so on are defined with these two axioms.

A special study is under way to compare the known R-unification algorithms (i.e. for tree terms) with respect to the following problems: (i) effeciency, (ii) applicability for the MRKP-system and (iii) possible hardware realization.

Finally we intend to develop and incorporate heuristically motivated unification algorithms that are not complete but sufficiently efficient: some theories (like e.g. associativity) are infinite, i.e. there exists in general an infinite set of most general unifiers for two given terms. But even finite sets of unifiers may be far too large to be practically computable or else even if the finite set is reasonably small (as in the A+C-case) the computation of this set is far too expensive. In all these cases efficient methods to compute a heuristically motivated subset of the whole set of unifiers have to be found. The difficult part is not so much to find such methods, but to gain enough experience for a practical evaluation, i.e. if the increase in efficiency outweighs the loss of completeness.

6.3.10 Equality Reasoning

Introduction

The use of the equality axioms in a theorem prover based on resolution has turned out to be very inefficient, since too many additional resolution operations involving the equality axioms are possible. This problem is well recognized whithin the field (see [WR67], [RW69], [SI69], [MO69], [BR75], [SH78], [HR78], [DI79]).
A way out is to directly incorporate equality into the proof procedure. One of the various methods proposed with this aim in mind is paramodulation, as described in 6.3.2: with one additional rule of inference, the paramodulation rule, the equaliy axioms become superfluous except for the reflexivity axiom.
But paramodulation, i.e. the replacement of terms by equal terms, can be applied almost everywhere in a clauseset and therefore paramodulation alone still does not solve the problem of "how to handle equality in an automatic theorem proving system".
Strategies or methods are required to control the enormous amount of potential steps and to make sensible use of the paramodulation rule.
A promising control mechanism may result from the "paramodulation if needed" idea, which states that the paramodulation rule should only be used to reduce differences between potentially complementary literals, such that an inference step (by resolution) becomes possible.

There are several methods known to realize the if needed idea (e.g [SH78], [HR78], [DI79]. The most explicit realization of which is Morris' E-resolution [MO69].
An E-resolution step can be viewed as a sequence of paramodulation steps such that two potentially complementary literals become unifiable, followed by the appropriate resolution step. This could be an optimal realization of the if needed idea

and potentially one of the best ways to handle equality in an ATP, because equations are only used when needed, i.e. to remove the difference between terms, which prevent their unification. Furthermore the equations are only used if it is possible to remove such differences completely.

An implementation of a proof procedure based on E-resolution however is unfeasable without additional search and control mechanisms, because two major problems remain:

(i) Equality of two terms with respect to the given set of equations is undecidable, and therefore in general it is impossible to continue searching for equations until the potentially complementary literals under consideration are unifiable (or definitely not unifiable). Hence the first problem is to organize the search for equations and the application of E-resolution in such a way, that the proof procedure is efficient and complete. The proof procedure based on E-resolution presented by Morris [MO69] is designed to ensure completeness [AN70] rather than efficiency.

(ii) Before an E-resolution step can be executed, the necessary equations have to be found and because of the enormous search space an unsophisticated and exhaustive search for such equations is prohibitively expensive.

In the following the essential ideas of the paramodulation-if-needed paradigm and the ideas of the paramodulated clause graph procedure (PCG-procedure) are combined by incorporating E-resolution into the PCG-procedure. This combination provides an efficient search procedure for possible E-resolutions, and a control mechanism to direct the search for E-resolutions solving the two problems (i) and (ii) mentioned above.

Paramodulation and resolution concern at most two clauses, whereas E-resolution is a generalization of resolution which

involves many clauses, therefore new structures (called paths) will be used to represent such macro-operations. A path repesents a sequence of P-links and consists of the necessary information to perform the respective macro-operation. During the search for a proof it is possible to select links as well as paths for the derivation of new clauses. ER-paths (E-resolution-paths) representing possible E-resolution steps are defined such that an ER-path connects two complementary literals and consists of the equations which make both literals unifiable.

Using this definition, the above stated problems of E-resolution may then be rephrased as:
(i) the integration of ER-paths into the proof procedure
(ii) the problem to search for ER-paths

Integration of Paths into the Proof Procedure

Just like R-links and P-links paths should be searched for and created right at the beginning, when the initial graph is formed, and the information contained in a path should then be inherited during the subsequent search for a proof.
But since the equality of two terms is undecidable, not every ER-path can be found in the initial graph. For that reason a new link type, PER-link (potential E-resolution link) connecting potentially complementary literals is introduced. These PER-links provide the toplevel information for the proof procedure to search for the corresponding ER-paths, and during this search three cases can occur:

a) All possible ER-paths corresponding to a PER-link are found.
b) It is detected that there does not exist an ER-path for a given PER-link.
c) The search for ER-paths has to be terminated because of space or time limitations.

For each of these three cases different operations are performed:

a) If all ER-paths are found, the PER-link is erased and replaced by the ER-paths. All possible ways to remove the differences between the literals connected by the PER-link are known now and given by the ER-paths. If an ER-path is selected, the E-resolution step corresponding to the ER-path is executed by the proof procedure.
b) There exists no ER-path between the literals connected by the PER-link, i.e. the differences between the respective terms cannot be completely removed and therefore the PER-link is deleted.
c) If the search is terminated and no (or not all) ER-paths are found, the PER-link remains in the graph. If nevertheless a PER-link is selected by the proof procedure, then an operation is executed to reduce the difference between the literals connected by the PER-link using the information obtained from the incomplete search.

PER-links are treated as a special case and are only used by the global strategies as auxiliary links to control the search for paths.

During the actual search for a proof ER-paths as well as PER-links are inherited and used to compute new ER-paths and new PER-links. Hence each ER-path used in the proof was already present in the initial graph or has been inherited from an initial ER-path.

This method of using ER-paths has three main advantages:
(i) ER-paths and PER-links contain important information for global strategies which plan and control the whole search for a proof. This information is particularily useful, because it is already available at the beginning of the proof.
(ii) An efficient handling of equality is possible, because the search is done in the small initial graph only. Furthermore only a few paths are found by inheritance.

(iii) Strong additional deletion rules are available, which greatly reduce the potential number of paths as well as the size of the graph. For example in case b) above the deletion of a PER-link may lead to a pure clause (just as in the usual clause graph proof procedure) causing the wellknown snowball effect of deletions to start.

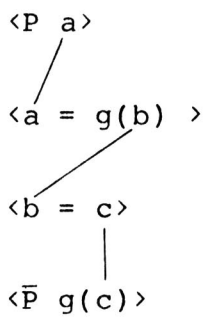
The Search for Paths Using Constraints

An unsophisticated and exhaustive search for ER-paths (i.e. for E-resolution steps) is prohibitively expensive, because of the very large search space, the size of which essentially depends on the number of equations (ne) in the clauseset, on the average number of subterms (ns) in a literal, and on the ratio (r) of positive to total (i.e. positive plus negative) unification tests. Finally it depends on the search depth (sd). A breadth first search procedure, like that of Morris [MO69] must execute ((2 * ne * ns * r)sd * 1/r) calls of the unification algorithm. Also (2 * ne * ns * r)sd equality replacements have to be carried out in order to get all possible E-resolvents for one PER-link. For example if a clauseset has ten equations, on average five subterms per literal and 10% of the unification attempts are successful, then for a search depth of ten steps there are 10^{11} calls of the unification algorithm and 10^{10} replacement operations necessary. To worsen the situation, Morris [MO69] observed that in a usual clauseset most PER-links do not have an E-resolvent, thus most effort is spent in vain.
In order to obtain a practical search procedure, the enormous search space has to be reduced somehow.

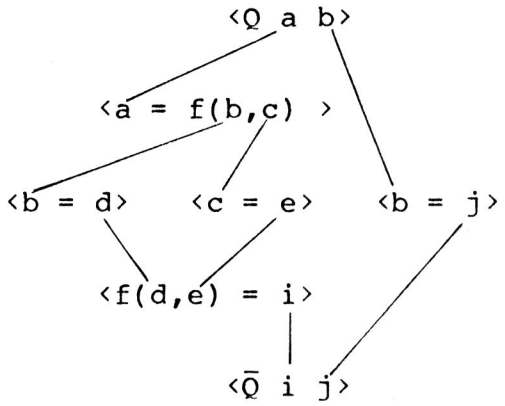
In the following we propose a reduction of the search space by exploiting constraints, where the constraints use the information contained in the P-links.

For a given clausegraph and a PER-link there are usually many P-links connecting the effective equations and the two literals. If

these P-links are visualized graphically, certain structures emerge, e.g. an equality-chain as in the simple example 1. In most cases the structure is a more complex equality-net as example 2 demonstrates:



Example 1

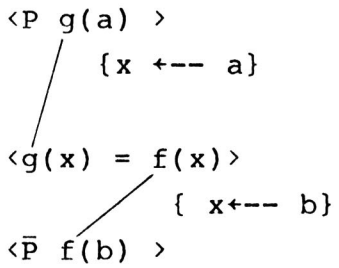


Example 2

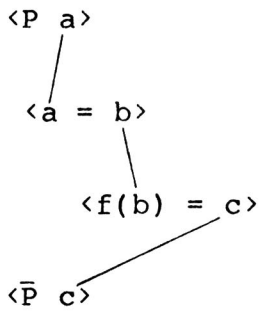
An equality-net (EN) and a compatible equality-net are defined as special graph structures of P-links. Each compatible equality-net is an ER-path and represents a possible E-resolution step. Vice versa for each possible E-resolution step there exists a compatible EN in the graph, which builds the appropriate ER-path. These conditions hold, because the PCG-procedure is complete [SW79]. In order to find the ER-paths, the compatible equality-nets must now be searched for.

To this end several conditions can be stated which are necessary but not sufficient for the compatibility of an EN. Such constraints are checked step by step drastically reducing the search space. This method is known as exploiting constraints in other fields of artificial intelligence.

The following examples show two reasons for the incompatibility of equality-nets:



Example 3



Example 4

In example 3 the combination of P-link 1 and P-link 2 is incompatible, because their unifiers {x +-- a} and {x +-- b} are incompatible. In example 4 P-link 1 and P-link 2 are incompatible: after paramodulation on link 1, link 2 cannot be inherited to the paramodulant <P b>, since the access depth does not coincide.

Some of the constraints to be used are:

(i) In a compatible EN all unifiers of the P-links concerned must merge to one mgu (i.e. the unifiers must be compatible, provided a proper variable renaming).

(ii) In a compatible EN: for each maximal equality-chain in the EN the sum of all access depths must be equal to zero and each partial sum must be less or equal to zero.

(iii) In a compatible EN, no chain of two links in the EN is compatible, i.e. if an equality-chain of length 2 is incompatible. (Note: incompatible is different from not compatible).

The search procedure can be improved considerably by additional constraints.
Although such constraints are very usefull, they are not expensive to compute, for example only some integers must be

added to detect the incompatibility of an EN caused by condition 2.

After the reduction of the search space the most promising ENs are selected for a full compatibility test, and the surviving candidates are used to build the ER-paths.

Summary and Future Plans

Paramodulation based on P-links is already incorporated into the MKRP-system. The method to control the paramodulation steps as outlined above is currently being implemented and evaluated on the basis of the present findings. It is still to early however to draw any definite conclusions.

6.3.11. Proofs by Induction

A survey of the Induction-Proof System is presented which is currently under developement and partely implemented.

The System Language

For a system designed to prove theorems by induction, i.e. proving theorems in constructives theories, it is a necessary prerequisite that the objects under investigation are defined in a constructive way. Hence each induction theorem proving system must provide a system language which contains some kind of a programming language, called the algorithmic sublanguage, to allow

- the definition of well-founded sets, and
- the definition of functions and predicates (which operate on these well-founded sets).

To express properties of the functions and predicates defined using the algorithmic sublanguage, the system language has to contain another sublanguage, called the logic sublanguage. The algorithmic and the logic sublanguage constitute the system language of an induction theorem proving system.

For the Karlsruhe induction theorem proving system we have chosen the many-sorted first-order language PLL (see 6.3.1 and 6.6.2) as the logic sublanguage, because the theories under investigation are many-sorted and hence properties of functions and predicates are most conveniently expressed in a many-sorted first-order language [WA82].

The logic sublanguage PLL is extended by the algorithmic sublanguage, yielding the Predicate Logic Programming Language (PL2), which is the system language of the Karlsruhe induction theorem prover [WA83]. The algorithmic sublanguage consists of a definition principle for well-founded sets and for functions and predicates.

Well-founded sets are represented by structured sorts or structures for sort. Structures are introduced by a so called structure declaration.

Functions and predicates defined on the well-founded sets represented by structures are introduced by a function declaration or a predicate declaration respectively, i.e. a definition principle which uses the techniques of functional composition, definition by cases and definition by recursion.

The semantic of PL2 is defined by a mapping of PL2 expressions into formulas of the logic sublanguage PLL, yielding the so called definition formulas: Formulas of the logic sublanguage are mapped into themselves. Expressions of the algorithmic sublanguage are treated as abbreviations for the definition formula of the respective expression.

Additionally each function and predicate declaration is associated with a PLL formula, called the uniqueness formula, which expresses that the intended function or predicate is unique for each given input.

The recursion formula for a function or a predicate declaration, is a PLL formula, which expresses that for a given well-founded order relation the intended function or predicate will terminate.

Specification of the System

The induction theorem proving system can be viewed as a system which maintains a sequence of PLL formulas, called the database, whose conjunction represents a certain mathematical theory.

There are only two operations defined by the system:

- extend the data base by a new PLL formula
- remove the PLL formula most recently inserted from the data base

However the system will not accept just any formula for extension of its data base. The expressions given to the system have to be admissible PL2 expressions:

- a structure declaration is always an admissible expression
- a function or predicate declaration is an admissible expression if the associated uniqueness formula and (for some well-founded order relation) the associated recursion formula can be proved by the system using the conjunction of formula in the data base as hypotheses.
- a PLL formula is an admissible PL2 expression if it can be proved by the system using the conjunction of formula in the data base as hypotheses.

For each PL2 expression given as input, the system has to improve the admissibility of the expression. For each admissible expression, its definition formula will be used to extend the data base.

It can be shown, that by accepting only admissible expressions, the conjunction of formulae in the data base is always satisfiable.

Practical Implications

Using the specification of the last section, the induction theorem proving system is used to define theories by extending its data base by formulae which represent well-founded sets, by function and predicates, which operate on these well-founded sets and by formulae, which state properties about this function and predicates.

The constraint to admissible expressions as input for the system guarantee that the theory represented by the conjunction of the formulae in the data base is always consistent.

To check the admissibility of a given input, two main problems have to be solved by the system:

(1) find a well-founded order relation (if possible), such that the recursion formula for a given function can be proved, and

(2) prove the formulas involved with the admissibility tests.

Problem (1) is called recursion analysis: Using special heuristics the system attempts to find a well-founded order relation which guarantees the termination of the given function or predicate.

Problem (2) splits into the following subproblems:

(2.1) Find adequate induction schemes such that the formula obtained from the given formula by instantiations suggested by the induction scheme can be proved without induction, and

(2.2) influence the control of a resolution theorem prover such that the formula thus obtained can be proved.

Conclusion

Presently the following problems have been solved:

- definition of the system language PL2 of the induction theorem proving system

- implementation of a PL2 compiler

- developement and implementation of a ´simplification´ module which transform PLL formulas to equivalent but simpler PLL formulas. (This module is in particular useful to prove uniqueness formulas)

The following modules are presently under development:

- the recursion-analysis-module,

- a module which eliminates existential quantifiers by skolem functions for which constructive definitions are obtained using techniques of program synthesis.

- a control-module which performs the necessary interactions between the induction system and the resolution theorem prover, i.e. the logic machine of the MKRP-system.

6.3.12 Preprover

The preprover is part of the Domain Specific Preprocessors mentioned in section 6.2 (see figure 2) and consists of simplification techniques and fast decision methods.

The Nelson-Oppen Method for Combining Decision Procedures

In 1979 G.Nelson and D.Oppen [NO79] have developed a method for combining decision procedures for several (disjoint) theories into a decision procedure for the combination of the theories. The method , which is applicable to quantifier-free first-order theories, is particularly useful for applications in program verification. We have implemented a special version of the method, which contains decision procedures for the following quantifier-free theories: real arithmetic under addition and order, the theory of equality with uninterpreted function symbols and the theory of list structures under cons, car, cdr and atom. The implementation thus constitutes a special theorem prover for the combination of the above theories. Besides implementing the prover we have also improved some theoretical results which are relevant to the method and have given a rigorous proof of correctness of the method, see [BA82a], [BA82b].

The Nelson-Oppen method is based on a specific kind of interaction between the individual decision procedures which have to be combined. Roughly speaking, interaction takes place by "propagating equalities", that is, all equalities between variables which are found to be a logical consequence of a given formula by one decision procedure are transmitted to all other decision procedures. Practical experience has shown that the capability of the individual decision procedures to detect the relevant equalities in an efficient way, heavily influences the efficiency of the overall method. Therefore we have extended some of the decision procedures, which are contained in our prover, in an appropriate way in order to achieve a better performance of

the general proving procedure.

Work has now been undertaken in implementing and including a decision procedure for the theory of arrays under store and select. We also plan to include further decision procedures for theories which are useful for application in program verification. Consideration will also be given to developing methods for using the theorem prover as a simplifier.

The Simplifier of King

In order to prove theorems that are generated mechanically by a VCG, King developed a powerfull special purpose simplifier for arithmetics [KI69].

The simplifier consists of four parts:

(i) the computation of canonical arithmetical terms and of standarized equality and unequality relations.

(ii) a satisfiability test for systems of equality and unequality relations which also reduces the number of relations in the system.

(iii) logical reductions like subsumption working on the conjuncts of the disjunctive normal form of the theorem.

(iv) a linear solver working on linear relations using methods of linear algebra.

A handsimulation of the simplifier together with the truthfunctional reduction method to be described below for the case of a sorting program (and others), has demonstrated the extrem usefulness of this combination: Allmost all verification

conditions were proved and the remaining ones were substantially simplified. Furthermore the transformation of arithmetical expressions and relations into normal forms eases the work of other theorem provers to be carried out afterwards.

Truthfunctial Reduction

In order to prove a theorem of the form A∧B → A∧C we may simply prove A∧B → C instead. Calling reductions of this kind truthfunctional reductions, they may be extended to quantified subformulae also:

For example in

$$(ALL\ x\ A) \rightarrow B$$

A may be reduced to A´ and if x does not occur in A´ then

$$A' \rightarrow B$$

may be further reduced.
In combination with the King simplifier of the previous paragraph these truthfunctional reductions become a powerfull technique for the simplification of mechanically generated theorems.

At the moment there exists a simplifier for arithmetical relations and expressions, it reduces expressions and relations with constant arguments and tries to simplify arithmetical expressions and relations containing variables as much as possible.

The implementation of the King simplifier and the truthfunctional reduction method is under way.

A Production System for the Control of the Simplifiers and Fast Decision Procedures

Simplifiers or fast decision procedures work satisfactorily only if they are applied to problems of appropriate kind. It is

therefore desirable to control the order of their application automatically. For that reasons the call of the most appropriate decision or simplification component is to be carried out by a production system (similar to the selection module, see 6.4), which states the criteria of applicability in its conditional part.

6.4. The Logic Machine

The logic machine (LM) of the MKRP is an automated theorem prover for a sorted first-order calculus and consists of the selection module and the logic engine. The selection module embodies the control and selection functions that govern the behaviour of the logic engine, a traditional theorem prover based on an extension of Kowalski's Connection Graph Proof Procedure.

We shall now depart from this conceptual point of view and present the system (May 1983) using the flow of control as a guiding line.

A typical session, after starting and initializing the MKRP system, begins by typing in some axiom and theorem formulas. If all formulas are accepted by the system, the user types in OK and the search for a proof commences.

Further progress is controlled by the control module and is divided into the main phases, each of them with several subphases:

Phase 1: Preprocessing

- Conversion of the formulas into clausal form and construction of the connection graph.
- Application of the simplification rules to the formulas.

Phase 2: The deduction loop

- Selection and execution of a resolution, paramodulation or factorization step, and
- Reduction of the graph until certain break off conditions come true.

If it is possible to split the formulas into several smaller pieces and the option GEN:SPLITTING is not NIL, some steps of phase 1 and the deduction loop are performed for every single split part. The proof of the original theorem is assumed to be correct only after the system has found a refutation for every split part.

These major phases, summarized in fig.1, are now sufficiently elaborated in order to understand why the logic machine makes a particular step in a particular situation and to enable the user to set and apply the control options. Although the default values of these control options are preset in an "optimal" way, they are of course not optimal for every theorem and some knowledge about the mechanism they are controlling is a prerequisite for a manual setting, which is advantageous in exploiting the whole power of the system.

A complete description (at the implementational level) is outside of the scope of this report, at the end of this section, however the major modules are mentioned and summarized in fig.6.

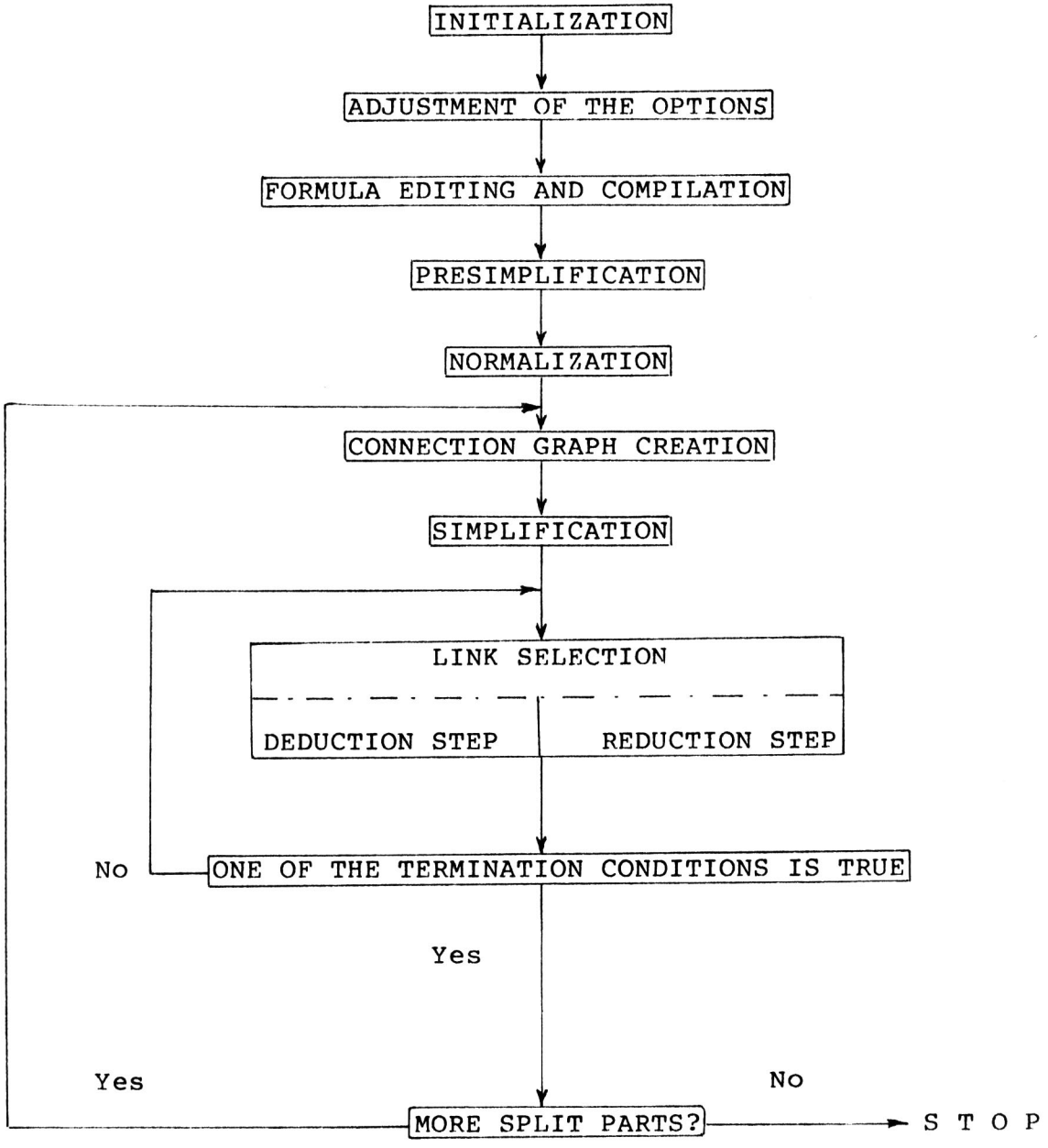


Figure 1: Flow of Control

6.4.1 Preprocessing

The initial formulas given to a theorem prover are usually highly redundant - in particular when they are automatically generated, for example by a verification condition generator.

Secondly the use of the connection graph prover with its extensive and expensive selection procedures is sometimes like using a steamhammer to crack a nut: the theorem may be just too simple to activate the whole machinery.

For these reasons elaborated simplification and fast special purpose theorem proving techniques are called upon, before the logic machine is activated.

The actual programs of the preprocessing phase are located within the modules PRESIMPLIFICATION, NORMALIZATION, SIMPLIFICATION and CONSTRUCT, all of which are coordinated by the CONTROL module.

6.4.1.1 Preprover

This module which is still under developement, is planned as a special expert system, containing simplification techniques for arithmetic, logical expressions, equalities and inequalities, set theory and others. Fast special purpose decision methods, as discussed in 6.3.10 are employed to either proof or at least reduce the initial formulas.

If the theorem has not been proved the reduced and simplified set of first-order formulas is passed on.

6.4.1.2 Normalization and Splitting

The formulae remainig after simplification, which are already in prefix form, are now converted in skolemized conjunctive normal form subject to various optimization techniques. As a result

there are two sets of variable disjoint clauses: the axiom clauses and the theorem clauses.

If the option GEN:SPLITTING was set to TRUE or to a natural number the variable disjoint parts of the input formula are split into several sets of clauses , which can be proved independently [EI83].

Example:

A theorem NOT [(A OR B) AND (C OR D)]
(after negation)

is split into the two parts

NOT (A OR B), NOT (C OR D)

And a theorem NOT [A IMPL (B AND C)] (after negation)

is split into NOT (A IMPL B), NOT (A IMPL C)

The algorithm is optimized for getting a maximum number of split parts with a minimum number of clauses.

6.4.1.3 Clause Simplifications

The resulting set of clauses is further simplified according to the following rules:

Deletion of Tautological Clauses

A clause containing two literals which have
- different signs
- the same predicate symbols and
- corresponding term lists, which are equal under a given theory

is recognized as a tautology.

Example: ‹NOT P(a), P(a), Q(x) › is a tautology
‹NOT P(f(a,f(b,c))), P(f(f(a,b),c)), Q(x) › is a tautology, if f has the attribute "associative"
If R is a symmetric predicate:
‹NOT R(a,b), R(b,a), Q(x) › is a tautology too.

A clause containing a literal of the form $R(t_1, t_2)$ with t_1 equal to t_2 under a given theory is recognized as a tautology too, if R is a symmetric predicate.

Example: ‹g(a) = g(a), Q(x) › is a tautology.
‹R(f(a,f(b,c)),f(f(a,b),c)), Q(x) › is a tautology if R is reflexive.

Deletion of Multiple Literals

Two literals are recognized as identical and one of them is erased if they have

- the same sign
- the same predicate symbol
- corresponding term lists, which are equal under a given theory

Example: If f is associate and P is symmetric in
‹P(a,f(x,f(y,z))), P(f(f(x,y),y),a) ›
then the second literal is erased.

Deletion of Reflexive or Irreflexive Literals

Suppose the predicate R denotes a reflexive [irreflexive] binary relation. Now if the literal $R(t_1,t_2)$ occurs in a clause and t_1 is equal to t_2 under a given theory T, then this literal is false and hence may be deleted. If the clause was a unit, all further processing is stopped with the theorem proved.
The literal $t_1 = t_2$ is but a special case of this rule and may hence be deleted.

Replacement Factoring

A clause may be subsumed by its own factor.

Example: <P(x), P(a), P(b) > has a factor:
<P(a), P(b) >, which subsumes its parent.

A literal L in a clause C is recognized as a factorization literal and immediately removed if there is a second literal M in C that can be factored, i.e.:

- L and M have the same sign
- L and M have the same predicate symbol
- there is a matcher μ such that μL=M
- for any other literal K in C, which is affected by μ:
  μK is either FALSE (contradicts the reflexivity or irreflexivity) or else is equal to another literal, which is not affected.

6.4.1.4 Construction of the Connection Graph

Once the clause simplification is finished the following steps are performed:

- The generation of the axiom graph, i.e. the search for and the construction of the links between the axiom clauses.
- The initial reduction of the axiom graph.

If there are more than one split parts of the theorem, the axiom graph is saved automatically and then for every split part of the theorem:

- The theorem clauses are connected to the axiom graph, i.e. the links between the theorem clauses and axiom clauses are drawn.
- A graph simplification and an
- Initial reduction of the entire graph are performed.

Initial Reduction of the Axiom Graph

The reductions described below as well as the deletion of tautologies may be switched on and off by setting the following options to TRUE or NIL:

RED: PUR.INITIAL	RED:PUR.CLAUSES.INITIAL
RED: TAU.INITIAL	RED:TAU.CLAUSES.INITIAL
RED: SUB.INITIAL	RED:SUB.CLAUSES.INITIAL

Depending on the adjustment of these options the following reductions take place:

(i) Clause subsumption (RED:SUB.CLAUSES.INITIAL)
Elimination of clauses which are subsumed by other clauses, see section 6.3.3

(ii) Link tautology (RED:TAU.INITIAL)
Elimination of R- and P-links which would generate a tautological resolvent of paramodulant, see section 6.3.4

(iii) Link subsumption (RED:SUB.INITIAL)
Elimination of an R-link the resolvent of which would be subsumed by an already existing clause, see section 6.3.3

(iv) Link purity (RED:PUR.INITIAL)
Elimination of R- and F-links the resolvent resp. factor of which would be pure, see section 6.3.4

Graph Simplification

After the theorem clauses are connected to the axiom graph, the complete graph (with axiom and theorem clauses) is further simplified by:

(v) Purity

Every clause containing a predicate symbol which does not occur with opposite sign in any of the other clauses is removed from the graph.

Note: A literal may be connected by a P-link -{i.e. it is not pure in the sense defined below}- although there is no literal with opposite sign and the same predicate symbol.

(vi) Deletion of Isolated Partial Graphs

This simplification is performed if

- there is at least one theorem clause and
- there is no equality literal in the graph.

In this case clauses are deleted, if they are completely separated from the theorems, i.e. no resolution with a theorem clause is possible. This deletion rule preserves completeness only if the axioms are satisfiable (which at this stage we always assume to be the case).

Example:	AXM1	<NOT P(b) NOT S(a)>
	AXM2	< P(b) S(x)>
	AXM3	<NOT R(a)>
	AXM4	< R(x) Q(a)>
	THM1	< NOT Q(x)>

AXM1 and AXM2 are removed.

Initial Reduction of the Entire Graph

Now the four initial reductions for the axiom graph mentioned aboved are applied again to the entire graph. Additionally, a fifth reduction is performed, which could not be applied to the axiom graph alone, because of the missing links to the theorem clauses:

(vii) Clause Purity (RED:PUR.CLAUSES.INITIAL)
Elimination of clauses with at least one literal which is not connected by at least one R- or P-link.

Once all the above operations are performed the initial graph construction is finished and the search for a proof commences.

This search for a proof is done in the following deduction loop and the first essential step to be carried out in this loop is a call to the terminator (see 6.3.5) to see if an easy proof can be found within a predefined complexity bound. If a proof can not be found so easily, the actual search is started.

6.4.2 The Deduction Loop

The selection of the most appropriate resolution, paramodulation or factorization step as well as the decision when certain reduction rules are to be applied are a central issue of the MKRP system.

The selection module is organized as a production system and consists of a number of rules (which are called operation blocks) of the form:

CONDITION → ACTION

These rules (operation blocks) are ordered and the first rule in this order, whose condition becomes TRUE, takes over control and performs a chain of correlated deductions.

More specifically, each operation block consists of the following three key functions:

(i) The update function
This function permanently watches the changing connection graph and updates the internal information necessary for its operation block; in other words each

production has a local memory, which can be read and updated.

(ii) The activate function
The activate functions of the blocks are called in a fixed order at the beginning of the deduction loop and immediately after the currently activated operation block has released control. The first activate function which returns a positive value, determines that its operation block shall take over control for the next steps.

(iii) The execute function
This function actually selects the links to be operated upon and the particular reductions to be performed.

The following paragraphs describe the currently implemented operation blocks in the same order as their activation functions are called initially, the overall flow of the control is however data driven by the actual state of the graph and may of course be entirely different.

6.4.2.1 Factoring

a) Activation condition:
There is at least one F-link in the graph.

b) Deductions:
All F-links in the current graph are worked off.

c) Reductions:
The executed F-links are removed after each step. The clause reduction rules are applied to the factors only after the last F-links has been removed.

d) User control:
Factorization of the initial graph is switched on by setting the option FAC:INITIAL to TRUE.
Factorization during the subsequent steps is switched on by setting FAC:EACH.STEP to TRUE.

Remark: As long as any of the other operation blocks is active, no factorization is performed.

6.4.2.2 Graph Reduction

Deductions which generally reduce the size of the connection graph or else are for other reasons important enough to be carried out immediately are given a favoured status with high priority.

a) Activation conditions:
There is at least one R-link which satisfies at least one of the following conditions:

- Both parent clauses are pure after resolution upon this link (i.e. they can be deleted).
- The resolvent subsumes one of the parent clauses (replacement resolution, i.e. the parent is deleted).
- One parent clause is a unit clause and the other one becomes pure after resolution upon this link, (i.e. the resolvent replaces the parent clause and has fewer literals).
- Resolution upon this link is a merge resolution, such that the resolvent has fewer literals than either of the parents.

Remark: These conditions are tested only once for every link, so the purity conditions which may change during the progress of the proof are not always recognized.

b) Deductions:
All favoured R-links are worked off.

c) Reductions:
At each step the clause reduction rules are applied.

d) User control:
None.

6.4.2.3 Term Elimination

a) Activation conditions
There is at least one unit equality clause in the graph, satisfying one of the following conditions:

- The form of the clause is

  constant = term

  and the constant does not occur in the term.

- The form of the clause is

  $f(x_1, \ldots, x_n) = \text{term}$

  and the function symbol f does not occur in term and no theory is defined for f.

  With such an equation every occurrence of the constant or function symbol can be eliminated in the entire graph, subject to the proper sort relationships.

b) Deductions:
The paramodulations necessary to eliminate constant and function symbols satisfying the conditions given in a) are carried out.

c) Reductions:
Each time such an equation has been worked off, the equation and every parent (of the paramodulant thus generated) is removed from the graph.
The remaining paramodulants are reduced.

d) User Control:
None.

Example:

AXM1 ‹P(c1) Q(c1) Q(c2)›
AXM2 ‹c1 = c2›

AXM1 + AXM2 = PAR1 ‹P(c2) Q(c1) Q(c2)›
PAR1 + AXM2 = PAR2 ‹P(c2) Q(c2)›

AXM1, AXM2 and PAR1 are removed.

(Notice: Each clause is paramodulated only once).

Remarks:

- Due to missing P-links in the current implementation, it is possible that not all occurrences of the constants or functions are eliminated.

- The treatment of functions for which theories like associativity etc. are declared is not yet implemented:
  In order to preserve completeness one has to insert a clause which expresses the theory and to paramodulate this clause too.

6.4.2.4 Terminator

The terminator is a separate module, which is called by the selection module and finds a refutation for a unit refutable clause set (see section 6.3.5). The terminator operation block

within the selection module determines when the terminator has to be activated and it processes the output of the terminator module.

Three options influence the flow of control within the terminator:

(i) STR:TERM.DEPTH
If this value is set to a natural number, no unit clauses with a term list deeper nested than this number are internally generated.

(ii) TERM.ITERATIONS
This number determines how often a non-unit clause is examined for a terminator situation.
Remember that a first examination of a clause might fail, but a second or third one succeed because in the meantime new unitclauses may have been generated during the examination of the other clauses.
If TERM:ITERATIONS is equal to zero, no new unitclauses are generated and only the one level terminator situations can be found.

(iii) TERM:UNITS

The effect of this option is explained below.

The terminator operation block within the selection module consists of:

a) Activation condition:
Case 1: TERM:ITERATIONS = 0
Because this is a very fast operation mode for the terminator, it is called after every resolution step and takes over control immediately once a terminator situation has been found.

Case 2: TERM:ITERATIONS > 0
This is a more time consuming operation mode for the terminator used only for difficult examples. The module is called at the beginning of the proof and is called again whenever at least three clauses are generated by other operation blocks (and are not subsequently removed by reduction operations).

b) Deductions:
If a terminator situation has been found, the corresponding resolution steps are performed.
If a terminator situation has not been found, but some unitclauses can be calculated, the action to be taken depends on the option TERM:UNITS:

TERM:UNITS = NIL
The result of the terminator is ignored and control is released.

TERM:UNITS = TRUE
The resolution steps which generate the unitclauses as proposed by the terminator are performed.

c) Reductions
Intermediate results, i.e. those non-unit clauses, which are necessary to deduce unitclauses are deleted. In the TERM:UNITS = TRUE mode, the reduction rules are then applied to the remaining new clauses.

d) User control:
Adjustment of STR:TERM.DEPTH, TERM:ITERATIONS and TERM:UNITS.

6.2.4.5 Term Rewriting Rules

A unit equality clause t1 := t2 resp. t1 =: t2 (formulated with the special equality symbol := resp. =:) is treated as a rewrite rule (see 6.3.9). Each P-link connected to the side of the equation marked by the colon is worked off as soon as possible.

a) Activation condition:
There is at least one P-link, not marked "inhibited" and connected to the marked side of unit equality clause (with a one-way unifier).

b) Deductions:
Every paramodulation step, which uses this equation as a rewrite rule (or demodulator) is performed.
Depending on the options

STR:P.CONFLUENT = TRUE
STR:P.CONFLUENT = NIL

a normal form is computed or not.

c) Reductions

Case 1: Option STR:P.DEMODULATION = NIL
all paramodulants are reduced.

Case 2: Option STR:P.DEMODULATION = TRUE
the parent clauses and the intermediate results are removed from the graph and the remainig new clauses are reduced.

d) User control:
Usage of the special equality signs := or =: in the input language and adjustment of the option STR:P.DEMODULATION.

Example: AXM1: <R(f(y))>

AXM2: <P(f(f(a))) Q(f(f(b)))>

AXM3: <f(f(x)) := x>

AXM2 + AXM3 = PAR1: <P(a) Q(f(f(b)))>

PAR1 + AXM3 = PAR2: <P(a) Q(b)>

If STR:P.DEMODULATION = TRUE AXM2 and PAR1 are removed. (Notice: No clause is paramodulated more than once).

Side effects:
P-links connected to the other not preferred side of the equality literal are marked "inhibited". All P-links connecting the preferred side of the equality, but marked with a unifier (i.e. not a one-way unifier) are also inhibited. For example the P-link between AXM3 in the above example is inhibited for rewriting purposes. These inhibited links may only be selected for term elimination purposes.

6.4.2.6 Literal Rewriting Rules

A similar type of rewriting, not for terms, but for whole literals is handled in this operation block. For example: P(f(x)) : IMPL P(x) is a candidate for these reductions, where the left hand literal is replaced by the right hand one.

a) Activation condition:
A clause is marked as literal rewrite rule, if

- the clause consists of exactly two literals
- the predicate of both literals are equal
- the literals have opposite signs
- one of the literals is marked preferred by a colon.

If there is such a clause and an R-link connecting the preferred literal (with a matcher), the operation block is activated.

b) Deductions:
All resolution steps necessary to rewrite this literal are performed.

c) Reductions:

Case 1: STR:R.DEMODULATION = NIL
all resolvents are reduced.

Case 2: STR:P.DEMODULATION = TRUE
the parent clause and intermediate results are removed, the remaining new clauses are reduced.

d) User control:
- Marking literals as "preferred" is indicated in the input language by writing :IMPL :AND OR: etc.
- Adjustment of the option STR:R.DEMODULATION.

Example:

```
Input language          P(f(a)) AND P(f(b))
                        ALL x   P(f(x)) :IMPL P(x)

Clausal form:           AXM1:   <P(f(a))  P(f(b))>
                        AXM2:   <NOT P(f(x))  P(x)>

   AXM1 + AXM2 = RES1           <P(a)  P(f(b))>
   RES1 + AXM2 = RES2           <P(a)  P(b)>
```

If STR:P.DEMODULATION = TRUE, AXM1 and RES1 are removed.

Side effect:
All R-links connecting such a literal rewrite rule which are not preferred are marked "inhibited".

6.4.2.7 Conditional Term Rewriting Rules

This is a combination of term rewriting rules and deduction rules as explained below.

A clause is marked as "conditional term rewriting rule", if

- the clause consists of at least two literals
- one literal is a term rewriting rule as explained above.
- all other literals are marked preferred.

Example:

```
All x,y P(x) AND Q(f(y)) :IMPL g(x y) := c
```

a) Activation condition:
   - There is a P-link connecting the preferred side of the equation.
   - Every preferred literal is connected via an R-link to a unitclause.
   - The unifiers of these R-links are compatible with each other and with the matcher of the P-link.

b) Deductions:
   The literals representing the conditions areresolved away and afterwards the term rewriting is performed.

c) Reductions:
   The reductions of the resolvents depend on the option STR:R.DEMODULATION and the reductions of the paramodulants depend on STR:P.DEMODULATION as described below for the deduction rules and term rewriting rules above respectively.

d) User control:
   Adjustment of STR:R.DEMODULATION and STR:P.DEMODULATION and by the usage of the colons in the input language.

Example:

Input language: All x,y P(x) AND Q(f(y)) :IMPL g(x y) := c
P(a) AND Q(f(b))
R(g(a b) c) OR R(c g(a b))

Clausal form: AXM1: <NOT P(x) NOT Q(f(y)) g(x y) := c>
AXM2: <P(a)>
AXM3: <Q(f(b))>
AXM4: <R(g(a b) c) R(c g(a b))>

AXM1 + AXM2 = RES1: <NOT Q(f(y)) g(a y) := c>
RES1 + AXM3 = RES2: <g(a b) := c>
RES2 + AXM4 = PAR1: <R(c c) R(c g(a b))>
RES2 + PAR1 = PAR2: <R(c c)>

If STR:R.DEMODULATION = TRUE then RES1 is deleted.
If STR:P.DEMODULATION = TRUE then PAR1 is deleted.

Side effects:
All other R- and P-links connecting the conditional term rewriting rule which do not satisfy the conditions given in a) are marked "inhibited".

6.2.4.8 Deduction Rules

A clause is marked as a deduction rule, if

- the clause consists of at least two literals
- at least one but not all literals are marked preferred.

a) Activation condition:
Each preferred literal in a deduction rule is connected via an R-link to a unit clause and this set of R-links is compatible.

b) Deductions:
The set of compatible R-links to unitclauses resp. their descendants is worked off.

c) Reductions:
STR:R.DEMODULATION = NIL
Once the set of R-links has been worked off, the new resolvents are reduced and the intermediate clauses are kept.
STR:R.DEMODULATION = TRUE
Intermediate clauses are removed and the remaining clauses are reduced.

d) User control:
adjustment of STR:R.DEMODULATION and the usage of the junctor symbols with colons to mark a formula as a deduction rule.

Example:

Input language: P(a) AND Q(a) :IMPL R(b)
All x P(x) AND Q(a)

Clausal form: AXM1: <NOT P(a) NOT Q(a) R(b)<
AXM2: <P(x)>
AXM3: <Q(a)>

AXM1 + AXM2 = RES1: <NOT Q(a) R(b)>
RES1 + AXM3 = RES2: <R(b)>

If STR:R.DEMODULATION = TRUE, RES1 is removed.

6.4.2.9 Refinements

This is the last operation block, which is activated only if none of the other blocks is in control.
Classical refinement strategies like set-of-support, linear etc. are simulated in this block by classifying the R- and P-links as active or passive depending on the actually chosen strategy. Only one of the classified R- or P-links is selected and control is then released again.

a) Activation condition:
The activation condition is always TRUE.

b) Deductions:
Only one resolution or paramodulation.

c) Reductions:
The resolvent resp. paramodulant is reduced.

d) User control:
- Selection of the resolution strategy STR:RESOLUTION and paramodulation strategySTR:PARAMODULATION.

Resolution Refinements

The following is a listing of the refinements that are implemented so far, further extensions are planned.

BASIC

All links are marked active such that the selection mechanism described above cause a breadth first search.

User option: STR:RESOLUTION = BASIC

SET-OF-SUPPORT

Case 1: There is no theorem clause:
The strategy is switched to BASIC.

Case 2: There is at least one theorem clause:
All theorem clauses of the initial graph are assumed to be in the set of support. Hence all R-links between axioms are marked passive and those between axioms and theorem clauses are marked active. Later on all new clauses derived from at least one supported parent are put into the set of support and the R-links connecting to them are marked active. All other links are marked passive.

LINEAR

All R-links connecting the top clause are marked active. During the following deduction steps the active links are passivated and the links connecting the new clauses are activated resulting in a linear deduction chain. If no active link is available, for instance because the newly deduced clause is a tautology and removed from the graph, a backtracking mechanism reactivates one of the formerly created clauses.

The LINEAR resolution strategy interferes with the paramodulation strategies and activates only the P-links for the top clause.
The top clause of the LINEAR strategy is either user defined by a concatenation of "LINEAR", with the printname of the top clause or else is asked for by the system.

Example: STR:RESOLUTION = LINEAR.THM2" (or L.THM2)
defines THM2 as top clause.
If the user does not know the printname before starting the proof, he may use STR:RESOLUTION = LINEAR and the

system prints the clauses and then asks for the top clause.

UNIT

All R-links connecting unit clauses are activated, the other links are passivated.

User option: STR:RESOLUTION = UNIT

Combined Resolution Refinements

UNIT RESOLUTION prunes the search space considerably, but unfortunately it is not complete for all clause sets. For this reason, the user can define strategies:

STR:RESOLUTION = U-SOS
STR:RESOLUTION = U-B
STR:RESOLUTION = U-L

saying "if the clause sets is Horn renamable then UNIT RESOLUTION, else SET-OF-SUPPORT" (resp. BASIC or LINEAR). The clause set is then automatically tested weather it is Horn renamable, i.e. unit refutable, using a fast technique proposed in [LE78], and the actual strategy is adjusted according to the result of the test.

Paramodulation Refinements

Equality reasoning is a top-level module of its own and is currently being developed as described in 6.4.1. For that reason we only present the few rudimentary facilities that remained permanently within the logic engine.

NONE

All P-links are passivated. Hence, as long as any active R-link is available, no paramodulation is performed.

User option: STR:PARAMODULATION = NONE

REWRITE

If this option is taken a unit equality clause is automatically defined as a rewrite equation if the term size of one side of the equation is larger than that of the other side. P-links connecting a unitclause to the larger side of the equation (with a matcher) are activated, the other ones are passivated.

User option: STR:PARAMODULATION = REWRITE

UNIT ANCESTRY

P-links between a unit equation E and a unit clause, of which none of the ancestors is E itself, are activated, the other ones are passivated.
This strategy allowes to deduce copies of each unitclause under a given equation, but paramodulation into the descendant of such a copy (by the same equation) is suppressed.

Example:

AXM1	<NOT Q(b)>
AXM2	< Q(x) OR P(a)>
AXM3	< a = f(a)>
AXM4	<f(x) = g(c)>

AXM1, AXM2 = RES1	<P(a)>
RES1, AXM3 = PAR1	<P(f(a))>
PAR1, AXM4 = PAR2	<P(g(c))>

The last two paramodulation steps are performed by this strategy,

but further paramodulation steps with PAR1 producing <P(f(f(a)))>, <P(f(f(f(a))))> etc. are suppressed.

6.4.2.10 Deletion Steps

The following deletion operations, which have all been explained above, can be (dis)activated by setting the option keys to NIL or TRUE:

Clause purity	RED:PUR.CLAUSES.EACH.STEP
Clause subsumption	RED:SUB.CLAUSES.EACH.STEP
Clause tautology	RED:TAU.CLAUSES.EACH.STEP
Link tautology	RED:TAU.EACH.STEP
Link subsumption	RED:SUB.EACH.STEP
Link purity	RED:PUR.EACH.STEP

If the options are set TRUE the reductions are processed as described in 6.4.1.1 and 6.4.1.4.

All the clause simplifications of 6.4.1.3 are also performed as soon as a new clause is generated.

6.4.2.11 Resolution of Conflicts

A clause may fulfill several of the above requirements resulting in a conflict of which operation is to be performed. For this reason the clauses in the initial graph as well as every deduced clause are classified according to the following priority hierarchy:

1. eliminating equation
2. conditional term rewrite rule
3. term rewrite rule

4. literal rewrite rule
5. deduction rule

Hence, for instance, a clause satisfying the conditions for a literal rewrite rule as well as for a deduction rule is classified as a literal rewrite rule.

The refinement operation block classifies the R- and P-links as "active" and "passive". It depends now on the actual resolution strategy which of these links is selected for the next deduction step. To this end the list of active R-links is sorted according to unit preference strategy, i.e. the link with smallest link depth as first criterion and fewest number of literals in the resolvent as second criterion is the one with the highest priority. If the resolution strategy is LINEAR, the passive links are sorted such that backtracking is simulated.
The link to be operated upon is selected in the following order:

Case 1: The resolution strategy is not LINEAR

1. The first one in the list of active P-links.
2. The first one in the list of active R-links.
3. The first one in the list of passive P-links.

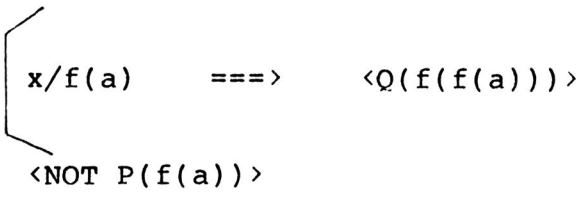
Case 2: The resolution strategy is LINEAR

1. The first one in the list of active R-links.
2. The first one in the list of active P-links.
3. The first one of the passive R-links.
4. The first one of the passive P-links.

There is an additional test to be performed at every step, depending on the options STR:TERM.DEPTH and STR:LINK.DEPTH, both of which can be set to NIL or to a natural number.
If STR:TERM.DEPTH is set to a natural number N, all R- and P-links generating clauses with terms lists deeper nested than N are marked "inhibited" in order to avoid the creation of very

deeply nested term lists.

Example: STR:TERM.DEPTH = 1



The term depth of f(f(a)) is 2 (maximum number of nested functions), so this resolution step would be suppressed.

If STR:LINK.DEPTH is set to a natural number, all R- and P-links with a link depth greater than this number are marked "inhibited" too.

The link depth d is defined as follows:

Links between clauses in the initial graph have the link depth d(L) = 0. For all other links L:

$$d(L) = 1 + \max(d(L1), d(L2))$$

where L1 and L2 are the links which generated the two clauses connected by L.

6.4.2.12 Termination of the Loop

The processing is terminated with positive result, if a refutation is found, i.e. the empty clause has been deduced.

It is terminated with negative result, if one of the following conditions come true:

(i) The number of deduction steps exceeds the boundary value GEN:MAXIMUM.STEPS.

(ii) The graph is collapsed, i.e. all clauses are removed.

(iii) The strategy operation block is activated, but there is no active link (resp. passive P-link) available. (LINEAR strategy still works with passive links).

Two additional functions are permanently watching the graph and may intervene regardeless of the currently activated operation block:

(iv) As soon as an R-link between two unitclauses is generated, a refutation is found and the first of these functions will notice that fact and terminate with positive result.

(v) An unsatisfiable clause set has at least one positive clause (a clause with only positive literals) and at least one negative clause.
If this condition is not fulfilled, either in the initial graph, or later due to the deletion of some clauses, the second function will stop all further processing and terminate with negative result.

Finals Remarks:

The classification of the links as active and passive is only respected by the refinement operation block. All other blocks ignore this classification.
For example, if the resolution refinement is SET-OF-SUPPORT, the reduction blocks may nevertheless force a resolution step with two axiom clauses. In this case however, the resolvent is not inserted into the set of support. In other words, the refinements do not provide the most important information, but are only taken into account if nothing else is against it, i.e. even with the strategies switched on, the system is not a "classical" theorem prover.

Resolution upon passive R-links is generally forbidden in the

refinement operation block (except for LINEAR strategy), because we assume that the completeness results for the refinements (which hold for ordinary resolution) are also valid for the connection graph procedure. Our paramodulation strategies however are not complete, therefore we allow paramodulation upon passive P-links too. Hence, "passive" P-links are not really passive, but they are links with lower priority than "active" ones.

Those links which are marked inhibited by some blocks are (with very rare exceptions) completely forbidden for all of the operation blocks. An exception is made, if it is necessary to paramodulate with an inhibited P-link in order to eliminate a constant or function symbol from the graph. Also it is too difficult (and therefore not checked) to observe the link depth boundary values for resolutions proposed by the terminator. The link depth boundary value is ignored too, when resolving away the conditions in conditional term rewrite rules and deduction rules.

6.4.3. The Module Configuration

At the bottom of the system there is a storage management module, which was necessary to save space and to overcome some gross blunders of the SIEMENS INTERLISP system .

On the top of this module the data structures for the usual logical objects (as e.g. variables, literals, clauses, links etc.) are defined as abstract data types.

The next layer provides the operations for various unification tasks and for the manipulation of the connection graph: each layer is carefully designed to operate only on those items defined at the layer below (see also [LMO82] for such an architecture).

The last layer is used by the deduction and reduction modules,

which perform the logical operations (like resolution, subsumption, factoring etc.). The order of these steps is determined in the selection module, which states for any given state of the graph which step is to be taken next.
The interplay between these modules is organized by the control module, which also initiates the various preprocessors and provides the necessary information for the protocol module.

6.5 Software tools

In the course of the development and implementation of the MKRP system many deficiencies of the programming environment, in this case SIEMENS-INTERLISP, were detected. To overcome these problems we developed several software tools to ease the program development and to increase the productivity. Of these tools three are of a more general kind: a service library, a module-concept [EI82] and a measurement-system [MI83] for INTERLISP. These tools are surveyed in these section.

The Service Library

The service-library is a collection of functions for basic operations and datatypes extending the INTERLISP standard. It represents the collective programming experience of our project and is constantly modified according to the joint decisions of our group. It is a very useful tool increasing productivity by avoiding the ´reinvent-the-wheel´ effect. Especially the extensions of LISP by functions for manipulating sets for accessing the operating-system, for symbolic computation and for an extended file-handling are important for our programming purposes and are heavily used. All our programs run in this much more comfortable environment without affecting their portability, because the service-library itself is implemented in standard INTERLISP.

A Module Concept

The introduction of a module-concept is the result of our effort to adopt features of typed languages, e.g. PASCAL, for INTERLISP. Essentially it consists of some identifier-conventions, e.g. the use of module-prefixes in variable or function names, and a large set of supporting functions. The new variable types LOCAL, COMMON, GLOBAL, EXTERNAL and the distinction between interface and internal functions of a module are introduced this way. These variable types and the module-concept are supported by an extension of the standard MAKEFILE command. It computes a detailed analysis of the call-hierarchy and the used variables of each function as well as a crossreference list of each variable of a module. Additionally open macros and substitution macros are automatically derived from the function definitions. This eases considerably the updating and the use of functions and supports a modular programming style, while avoiding runtime inefficiencies. Together with commenting and programming conventions this module-concept allows the static detection of some trivial errors like missing quotes or misspelled identifiers and greatly reduces the amount of run-time errors and the costs of testing.

The Measurement System

In order to optimize algorithms, to measure their performance and to get a detailed insight into the program execution, we developed a measurement system. Versatility, exactness, reproducability and minimality are the basic demands such a system should fulfill. In order to achieve these requirements, a special measurement language was developed. In the dialogue-part of the system the user specifies, using this language, which resources and which program or module he wants to be measured. Using this information a task-specific measurement program is completed, which, when loaded, automatically performs the measurement and saves the results. Thus for each task only a fraction of the system is in core, minimizing the inevitable

additional load of the CPU. To guarantee exact and reproducable results the system takes its own fraction of the consumed measured resources exactly into considerations: For example a detailed measurement of the MKRP system resulted in a more than tenfold increase in execution time (with a CPU-time error of +- 20%).

After the execution of the measured programs the results can be evaluated using a separate module of the system. This allows the preparation of statistical evaluations of sets of measured programs as well. The measurement system can be augmented by simply inserting new task-specific measuring functions.

In the present implementation the consumed CPU-time, the used LISP-storage, the number of calls and the share of subfunctions and submodules can be measured for each function or module.

Software Engineering Aspects

A large software developement of this kind could not be implemented without strict obedience to generally accepted software standards: apart from the selfimposed (i.e. not supported by INTERLISP) discipline with respect to modularisation and typing as mentioned above, the system has been designed in several layers.

These layers are as follows: at the bottom there is an INTERLISP system augmented by a special memory management system, that tries to cope with the extraordinary space requirements.

On top of this layer there is an intermediate level of auxiliary functions that are used to program the next conceptual level, which consists of abstract data types like clause, link etc. and provides functions like GETCLAUSE, MAKECLAUSE, GETUNIFIER and so on. This is the lowest conceptual level from the MKRP system point of view and provides the programming primitives for each

layer above. In particular this way a change in the interal representation does not affect the whole system, but only requires a reformulation of the intermediate level.

On top of these data structure functions are the three layers described in the previous paragraph on the module configuration and it is most interesting to see that these structures are very similar to the implementational structures of the Argonne National Lab Prover [LMO82]. The fact that their system was completely independently implemented but nevertheless the same pattern of implementational structures emerged is a clear indication of their "objective" nature and may be seen as the early hall mark of an emerging engineering discpline of theorem proving.

Just as a compiler or operating system consists of generally agreed upon parts (the symbol table, the parser, scheduling etc.), all of which are implemented according to well understood techniques and principles, a theorem prover is a wellstructured software product based on the historical software engineering experiences of earlier implementational attempts. A presentation and evaluation of these techniques would be one of the most desirable publications in order to prevent the "reinvention of the wheel" for every new theorem proving system.

6.6. How To Use The System

After the LOGON command the user types

Do ATP, X

which loads the INTERLISP system and the MKRP modules and sets up an initial system. Depending on the option X, which can be

- C10, C50, C100
- I10, I50

the system is either compiled (C) or interpreted (I) and consists

of a small configuration (10) for testing purposes or a medium (50) or large (100) configuration, depending on the anticipated work space.

The user is now in the operating system of the MKRP-PROCEDURE, he may use the input language PLL to formulate his problem, edit and check his input and finally set the output options, all of which are briefly surveyed below.

6.6.1 The MKRP Operating System

The operating system provides the following user commands:

HELP	CONTINUE
EXIT	EDIT
LOGOFF	OPTIONS
PROVE	LISP
INDUCTION	HC
	V

Their effect is briefly summarized as follows:

HELP CO explains a command CO to the user.

EXIT and LOGOFF are the termination commands.

Prove is the main command at this level, which provides the entry to the actual theorem proving system: after a short dialogue with the user,in the course of which source and destination of the axioms, theorem and proof(s) respectively are settled, the search for a proof is initiated.

INDUCTION enters the induction theorem proving facilities and CONTINUE is a useful command for the resumption of interrupted proof runs.

The OPTIONS command is the key into the module, which sets the various parameters (options) that govern the overall search behaviour. These options are classified into four areas:

RED, STR, GEN, PR

which can be set and updated with the following commands:

PRINT ‹area› displays the actual setting of the options in ‹area› on the screen.

WRITE ‹area› ‹file› writes the actual settin of the options in ‹area› onto a file ‹file›, for example for later use.

READ ‹file› reads the setting of the options from ‹file› into the predefined area.

V TRUE manual teletype control on

V NIL manual teletype control off

OK terminates the setting of the options

The four areas consist of the following options, which can be called by typing the name of the area.

RED

This area collects the options that govern the reduction steps. The user can switch the reductions on and off by typing TRUE and NIL:

RED	Defaults
RED:SUB.CLAUSES.INITIAL	T
RED:SUB.CLAUSES.EACH.STEP	T
RED:SUB.INITIAL	T
RED:SUB.EACH.STEP	T
RED:PUR.CLAUSES.INITIAL	T
RED:PUR.CLAUSES.EACH.STEP	T
RED:PUR.INITIAL	T
RED:PUR.EACH.STEP	T
RED:TAU.CLAUSES.INITIAL	T
RED:TAU.CLAUSES.EACH.STEP	T
RED:TAU.INITIAL	T
RED:TAU.EACH.STEP	T

STR

This area collects the options that are used to switch the refinements on or off:

STR	Defaults
FAC:INITIAL	NIL
FAC:EACH.STEP	NIL
STR:RESOLUTION	SET-OF-SUPPORT
STR:PARAMODULATION	UNIT-ANCESTRY
STR:LINK.DEPTH	NIL
STR:TERM.DEPTH	NIL
STR:R.DEMODULATION	T
STR:P.DEMODULATION	T
TERM:UNITS	T
TERM:ITERATIONS	0

GEN

The options of this area are used to influence some general parameters:

1 GEN:BATCH.ANSWER (Default SC)
IMPORTANT ONLY IF PROVER RUNS IN BATCH.
IS NEEDED IF THERE IS NO MORE SPACE FOR MEMORY-MODULE.
ABORT (A)
CONTINUE (C)
SAVE-ABORT (SA)
SAVE-CONTINUE (SC)

2 GEN:SAVE.FILE (Default SAVE.DEFAULTS)
<FILENAME> SYSTEM SAVES GRAPHS ON THIS FILE (IF GEN:GRAPH.SAVING NOT NIL !)

3 GEN:GRAPH.SAVING (Default NIL)
NIL : NO EFFECT

POSITIVE INTEGER: NUMBER OF DEDUCTION-STEPS BETWEEN TWO SAVINGS OF THE GRAPH (SEE GEN:SAVE.FILE)

4 GEN:MAXIMUM.STEPS (Default NIL)
POSITIVE INTEGER (NIL MEANS 'INFINITE')
THIS IS MAXIMUM NUMBER OF DEDUCTION-STEPS AT PROOF)

5 GEN:MANUAL.CONTROL (Default NIL)
INFLUENCE OF USER ON PROOF:

T, Y, YES	SWITCHED ON
NIL, N, NO	SWITCHED OFF

6 GEN:SPLITTING (Default 0)
AND-SPLIT OF THEOREM:

NIL, N, NO	SWITCHED OFF
NAT NUMBER	SWITCHED ON. MAXIMAL NESTING DEPTH UP TO WHICH MULTIPLICATION INTO DNF TAKES PLACE IN ORDER TO ENABLE SPLITTING
T, Y, YES	SWITCHED ON. MULTIPLICATION IN ALL NESTING DEPTHS.

PR

This area collects the options that account for the proof protocol mode:

PR	Default
PR:PREPROCESSING	NIL
PR:STEP.MODE	LR
PR:DUMP	NIL
PR:CLAUSE.MODE	I
PR:LINK.MODE	I
PR:TRACE.FILE	NIL
PR:TERMINAL	NIL
PR:PROOF.FILE	PR.DEFAULT

All of these options are set to a predefined default value that turned to be optimal for most runs. However to get the most out of the system they should be individually adjusted for every problem.

6.6.2. The Input Language

The PREDICATE LOGIC LANGUAGE (PLL), a formal language in which sorted first-order predicate logic formulas can be formulated, is described. Axioms and theorems, which are given to the MARKGRAF KARL REFUTATION PROCEDURE, are represented in PLL and the language constructs of PLL which reflect the special facilities of this system are exhibited, i.e.

- an inference mechanism based on a many-sorted calculus,
- the incorporation of special axioms into the inference mechanism, and
- the control of the inference mechanism using special derivation strategies.

Basic Concepts

In PLL all usual junctors, denoted OR, AND, IMPL, EQV and NOT, the universal quantifier ALL and the existential quantifier EX are present. Junctors and quantifiers have the following priorities when used in a formula without parantheses:

(1) NOT
(2) AND
(3) OR
(4) IMPL
(5) EQV
(6) ALL, EX

In a formula without parantheses, the rightmost junctor has precedence over all junctors with the same priority left of it.

Example

NOT A OR B AND C is equivalent to (NOT A) OR (B AND C) and
A IMPL B IMPL C is equivalent to A IMPL (B IMPL C).

In PLL the sign = denotes the equality symbol, i.e. we use a first-order predicate calculus with equality. As an example for using PLL, we axiomatize a group:

Example

```
* AXIOMIZATION OF A GROUP WITH EQUALITY,
* F IS A GROUP OPERATOR AND 1 IS THE IDENTY ELEMENT

ALL X,Y EX Z    F(X Y) = Z
ALL X,Y,Z       F(X F(Y Z)) = F(F(X Y) Z)
ALL X           F(1 X) = X AND F(X 1) = X
ALL X EX Y      F(X Y) = 1
```

A theorem given to the MKRP system could be for instance:

```
* IDEMPOTENCY IMPLIES COMMUTATIVITY

ALL X F(X X) = 1 IMPL (ALL X,Y,F(X Y) = F(Y X))
```

The lines starting with a '*' are PLL-comments. We give another axiomatization of a group:

Example

```
* AXIOMATIZATION OF A GROUP WITHOUT EQUALITY
* P(X Y Z) MEANS F(X Y) = Z WHERE F IS THE
* GROUP OPERATOR. E IS THE LEFTIDENTTY.
```

```
ALL X,Y EX Z       P(X Y Z)
ALL X,Y,Z,U,V,W    P(X Y U) AND P(Y Z V) IMPL
                   (P(X V W) EQV P(U Z W))
ALL X              P(E X X)
ALL X EX Y         P(X Y E)
```

Now a theorem could be for instance:

```
* LEFTIDENTITY IS RIGHTIDENTITY

ALL X P(X E X)
```

The Many-Sorted Calculus

Assume we have a set of symbols ordered by the subsort order, a partial order relation which is reflexive, antisymmetric and transitive. Variable, constant and function symbols are associated with a certain sort symbol. The sort of a variable or constant symbol is its rangesort and the sort of a term which is different from a variable or constant symbol is determined by the rangesort of its outermost function symbol.

All argument positions of a function or predicate symbol are associated with certain sort symbols, called the domainsorts. In the construction of the well formed formulas of the many-sorted calculus, only those terms may fill an argument position of a function or predicate symbol, whose sorts are subsorts of the domainsorts given for the argument position of the respective function or predicate symbol.

Beside the increase of readability of axiomatizations, the usage of the information given by the range- and domainsorts and by the subsort order prevents the inference mechanism of the theorem prover to do useless derivations. The theoretical foundation of the many-sorted calculus implemented in the MARK GRAF KARL

REFUTATION PROCEDURE can be found in [WAL82].

As an example for an application of a many-sorted calculus we axiomatize sets of letters and digits and some basic operations for these sets:

Example

```
* DEFINITION OF THE SORTS LETTER AND DIGIT, I.E.
* A,B, ... ,Z ARE CONSTANTS OF SORT LETTER AND
* 0,1, ... ,9 ARE CONSTANTS OF SORT DIGIT

TYPE A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P,Q,R,S,T,U,V,W,X,Y,Z : LETTER
TYPE 0,9,8,7,6,5,4,3,2,1 : DIGIT

* LETTER AND DIGIT ARE SUBSORTS OF SORT SIGN

SORT LETTER, DIGIT:SIGN

* DEFINITION OF THE EMPTY SET AND SET-MEMBERSHIP,
* I.E. EMPTY IS A CONSTANT OF SORT SET AND MEMBER
* IS A BINARY PREDICATE DEFINED ON (SIGN SET)

TYPE EMPTY:SET
TYPE MEMBER(SIGN SET)
ALL X:SIGN NOT MEMBER(X EMPTY)
ALL U,V:SET U = V EQV (ALL X:SIGN MEMBER (X U) EQV MEMBER (X V) )

* DEFINITION OF SINGLETONS, I.E.
* SINGLETON IS A FUNCTION MAPPING SIGN TO SET

TYPE SINGLETON(SIGN):SET
ALL X:SIGN ALL U,V:SET (MEMBER(X U) OR MEMBER(X V) )
                        EQV MEMBER (X UNION (U V))
```

Theorems to be proved by the ATP system could be for instance:

```
* UNION IS IDEMPOTENT AND EMPTY IS AN IDENTITY ELEMENT

ALL X:SET UNION (X X) = X AND UNION(EMPTY X) = X

* SINGLETON IS INJECTIVE

ALL X,Y:SIGN SINGLETON(X) = SINGLETON(Y) IMPL X = Y

* EACH LETTER IS A SIGN

ALL Y:SET (EX U:LETTER MEMBER(U Y))
                    IMPL (EX X:SIGN MEMBER(X Y))
```

Attributes of Functions and Predicates

Attributes are abbreviations for their defining axioms, i.e. first-order axioms which axiomatize certain properties of functions or predicates.

The effect in stating a certain attribute of a function or predicate using an attribute declaration is formally the same as giving the defining axiom to the ATP. At the moment the following properties can be declared.

Attribute Declaration	Defining Axiom
REFLEXIVE(P)	ALL X P(X X)
IRREFLEXIVE(P)	ALL X NOT P(X X)
SYMMETRIC(P)	ALL X,Y P(X Y) IMPL P(Y X)
ASSOCIATIVE(F)	ALL X,Y,Z F(X F(Y Z)) = F(F(X Y) Z)

The defining axioms of attributes are incorporated into the inference mechanism of the system as described above.

Example

In example 2.1.2 for instance the associativity of the group operator F could be stated by: ASSOCIATIVE (F). In example 2.2.1 we could write as an axiom: ASSOCIATIVE(UNION).

Special Junctors and Equality Symbols

For the junctors introduced above, PLL offers various alternative notations:

Junctor	Alternative Notations
AND	AND: , :AND or :AND:
OR	OR: , :OR or :OR:
IMPL	IMPL: , :IMPL or :IMPL:
EQV	EQV: , :EQV or :EQV:

The eqality symbol = can be alternatively denoted by :=, =: or :=:.

The colon-notation of junctors and equality symbols is used to influence the sequence of deductions as discussed in section 6.4. Formally there is no difference between junctors and equality symbols written with or without colons.

The Syntax of PLL

The formal syntax of PLL defined by a context free grammar is contained in [WA83], which presents additional details and examples.

Semantic Constraints for PLL

In the sequel we state the semantic constraints (i.e. the context dependent language features) for PLL. The strings in angle

brackets, e.g. ‹term›, refer to the production rules of the PLL-grammar as defined in [WA83].

Sort Symbols

Sort symbols are introduced with their first usage in

- a ‹sort declaration›, e.g. SORT LETTER,DIGIT:SIGN,ALPHABET
- a ‹type declaration›, e.g. TYPE A,B:BOOL,
  TYPE MEMBER(SIGN,SET) or TYPE SINGLETON(SIGN):SET,
- a ‹variable declaration›, e.g. ALL Z:INT EX N:NAT ABS(Z) = N.

The direct subsort relation imposed on the set of sort symbols is a partial, irreflexive and non-transitive relation such that the predefined sort symbol ANY is no direct subsort of each sort symbol and each sort symbol different from ANY is a direct subsort of at least another sort symbol.

The subsort order imposed on the set of sort symbols is the reflexive and transitive closure of the direct subsort relation.

The subsort symbols left of the colon in a ‹sort declaration› are direct subsorts of each sort symbol to the right of the colon in the ‹sort declaration›.

The sort symbols right of the colon in a ‹sort declaration› are direct subsorts of ANY, provided these sort symbols are introduced by this ‹sort declaration›.

The sort symbols which are introduced by a ‹type declaration› or by a ‹variable declaration› are direct subsorts of ANY.

Example

For the ‹sort declaration› given above LETTER and DIGIT are subsorts of SIGN and of ALPHABET, and SIGN and ALPHABET are

direct subsorts of ANY. Hence LETTER, DIGIT and SIGN are subsorts of SIGN and ANY, SIGN, ALPHABET, LETTER and DIGIT are subsorts of ANY.

Variable Symbols

Variable symbols are introduced by a ‹variable declaration› in a ‹quantification›.

Example

ALL X,Y EX Z:S P(X Y Z)

The scope of a ‹variable symbol› is the ‹quantification› following the ‹variable declaration› in a ‹quantification›.

In its scope each ‹variable symbol› has as rangesort the sort symbol given by the ‹sort symbol› following the colon in its ‹variable sort› of the ‹variable declaration›. If no ‹variable sort› is present, the rangesort of the ‹variable symbol› is the predefined sort symbol ANY.

Example

The expression given in the above example have the following sorts:
rangesort(X) = rangesort(Y) = ANY and
rangesort(Z) = S.

In each ‹quantification› variable symbols are consistently renamed from left to right to resolve conflicts on multiple introductions of variable symbols.

Example

ALL Y,X P(Y) is the same as ALL X,Y P(Y) and
ALL X (EX X P(X)) IMPL Q(X) is the same as
ALL X (EX Y P(Y)) IMPL Q(X)

Constant Symbols

Constant symbols are introduced with their first usage
- in a ‹type declaration›, e.g. TYPE -1,+1:INT
- as ‹terms›, e.g. ALL X P(X A) OR F(C) = D

Each constant symbol has a rangesort the ‹sort symbol› following the colon in the ‹type declaration› which introduces the ‹constant symbol›.

Example

For the expressions given above we find
rangesort(-1) = rangesort(+1) = INT.

The rangesort for a constant symbol which is introduced with its first usage as a ‹term› is ANY.

Note that in PLL variable symbols are always preceded by a quantifier and thereby can always be distinguished from constant symbols. As a consequence there is no concept of free variables in PLL.

Function Symbols

Function symbols are introduced with their first usage in

- a ‹type declaration›, e.g. TYPE ABS(INT):NAT
- a ‹attribute declaration›, e.g. ASSOCIATIVE(PLUS)
- a ‹term›, e.g. ALL X P(F(X)) OR G(X) = A

Each function symbol is associated with a sort symbol for each argument position i, called its ith domainsort, with a natural number, called its arity, and with a sort symbol, called its rangesort.

Function symbols which are introduced by ‹type declaration› have as ith domainsort the ‹sort symbol› given on the ith position in the list of ‹sort symbols› following the ‹function symbol› in the ‹type declaration›.

Example

For the expression TYPE PRODUCT(SCALAR VECTOR):VECTOR we get domainsort (PRODUCT 1)=SCALAR and domainsort (PRODUCT 2)=VECTOR.

A ‹function symbol› which is introduced by a ‹property declaration› or by its first usage in a ‹term› has ANY as ith domainsort for each argument position i.

The arity of a function symbol is defined as

- the number of sort symbols in the list of ‹sort symbols› following the ‹function symbol› in the ‹type declaration› which introduces the ‹function symbol›
- two, for a ‹function symbol› introduced by a ‹attribute declaration›
- or else the number of arguments on its first usage in a ‹term›.

Example

For the expressions given above we get arity(ABS) = 1, arity(PLUS) = 2 and arity(F) = arity(G) = 1.
The rangesort of a ‹function symbol› is defined by the ‹sort symbol› following the colon in a ‹type declaration›. Its rangesort is ANY if the ‹function symbol› is introduced by a ‹property declaration› or by its first usage in a ‹term›.

Example

For the examples given in the expressions above we get
rangesort(ABS) = NAT
rangesort(PRODUCT) = VECTOR, and
rangesort(PLUS) = rangesort(F) = rangesort(G) = ANY.

Predicate Symbols

A predicate symbol is introduced with its first usage in
- a ‹type declaration›, e.g.TYPE MEMBER(SIGN SET)
- a ‹atom› in a quantification, e.g. EX X,Y P(X Y) AND Q

Each predicate symbol is associated with a natural number, called its arity, and with a sort symbol for each argument position i, called its ith domainsort.

The arity and domainsort of predicate symbols are determined in the same way arity and domainsorts are determined for function symbols.

The ‹equality symbols› are predefined predicate symbols whose arity is 2 and whose 1st and 2nd domainsort is ANY. They are the only predicate symbols which are written in infix notation.

´TRUE´ and ´FALSE´ are predefined predicate symbols with arity 0, which have the obvious meaning.

In the following the numbers in angle brackets, e.t. ‹23›, denote error code numbers returned by the PLL-compiler of the MARKGRAF KARL REFUTATION PROCEDURE (summarized below) when given a semantically incorrect ‹expression› as input. The phrase unknown symbol denotes a string of the terminal alphabet of the PLL-grammar, which was not used before.

Semantically correct Sort Declarations

A ‹sort declaration› SORT S1, ... ,Sm:T1, ... ,Tn is semantically correct, if

- all Si and all Tj (i=1...m, j=1...n) are sort symbols or else are unknown symbols (otherwise error message) ‹61,62,63,64› and Si is a direct subsort of Tj ‹1› or else at least one of the symbols Si and Tj is an unknown symbol ‹2›.

Semantically correct Type Declarations

A ‹type declaration› T is semantically correct if

- T is TYPE C1, ... ,Cn:S and S is a sort symbol or else is an unknown symbol ‹61,62,63,64› and for all i=1...n Ci is a constant symbol with rangesort(Ci = S ‹14 or Ci is an unknown symbol ‹11,12,16,17›
- or T is TYPE P(S1,...Sn) and for all i=1...n Si is a sort symbol or else is an unknown symbol ‹61,62,63,64› and P is a predicate symbol with arity(P)=n ‹34› and domainsort(P=i)=Si‹36› or else is an unknown symbol ‹31,32,36,37›
- or T is TYPE F(S1...Sn):S and for all i=1...n S and Si are sort symbols or else are unknown symbols ‹61,62,63,64› and F is a function symbol with arity(F)=n ‹23›, rangesort(F)=S ‹27› and domainsort (F i)=Si ‹26› or an unknown symbol ‹21,22,24,28›.

Semantically correct Attribute Declarations

A ‹attribute declaration› ASSOCIATIVE(F) is semantically correct if

- F is a function symbol with arity(F)=2 ‹23›, rangesort (F) = domainsort(F 1) = domainsort(F 2) ‹26› or else is an unknown symbol ‹21,22,24,28›.

The ‹attribute declarations›s REFLEXIVE(P), IRREFLEXIVE(P) and

SYMMETRIC(P) are semantically correct if

- P is a predicate symbol with arity(P)=2 <34> and domainsort (P 1) = domainsort (P 2) <36> or else is an unknown symbol <31,32,33,37>.

Semantically correct Terms, Atoms and Quantifications

The sort of a term t, denoted sort(t), is the rangesort of t, if t is a variable or constant symbol, and else is the rangesort of the outermost function symbol of t.
A <term> T is semantically correct if

- T is a constant symbol, a variable symbol or an unknown symbol <11,12,16,17>

- or T is F(T1...Tn) and for all i=1..n, Ti is a semantically correct term, F is a function symbol with arity(F)=n <23> and sort (Ti) is a subsort of domainsort(F i) <81> or else F is an unknown symbol <21,22,24,28>.

An <atom> A is semantically correct if

- A is a predicate symbol with arity(A)=0 <34> or A is an unknown symbol <31,32,33,37>

- or A is P(T1..Tn) and for all i=1...n, Ti is a semantically correct term, P is a predicate symbol with arity(P)=n <34> and sort(Ti) is a subsort of domainsort(P i) <81> or else P is an unknown symbol <31,32,33,37>

- or A is T1 = T2, T1 and T2 are semantically correct terms and = is an <equality symbol>.

A <quantification>Q is semantically correct if

- Q is ALL X... or EX X... and X is a variable symbol or an unknown symbol <51,52,53,55> and each atom in Q is semantically correct.

Errors detected by the Compiler

The PLL compiler of the ATP system checks each input for syntactical and semantical correctness. An input containing signs which are not member of the terminal alphabet is responded by a message.

***** SYMBOL ERROR <<< xxx IS NO ADMISSIBLE SYMBOL

where xxx is a sign which is not member of the terminal alphabet.

For a syntactically correct input, the compiler responds

SYNTAX ERROR >>> xxx NOT ACCEPTED
UNEXAMINED REMAINDER OF THE INPUT >>> zzz

where xxx is the sign which causes the syntactical incorrectness and zzz is the unanalysed remainder of the given input.

For a syntactically correct but semantically incorrect input, the compiler responds

***** SEMANTIC ERROR nnn >>> message
UNEXAMINED REMAINDER OF THE INPUT >>> zzz

where ´nnn´ is the semantic error code, ´message´ is an error message explaining the kind of the semantic error and ´zzz´ is the (not analysed) remainder of the given input.

Particularities of the Input Routines

Since the whole ATP system is an INTERLISP program, the special features of the INTERLISP input routines have to be taken into account, i.e.

- () is read as NIL

- ´X is read as (QUOTE X)

- ‹ is read as (

- › is read as a non empty sequence of)´s

- › closes all left-brackets up to the first left-superbracket ‹

- each left-bracket has to be matched by a right-bracket or by a right-superbracket›

- each input has to contain an even number of " (i.e. the string indicator)

- a sequence of blanks is read as one blank (except in a string)

- + is read as a blank if it is followed by a sequence of digits, e.g. +4711 is read as 4711

- a sequence of zeroes is read as a zero, unless the sequence is preceeded by non-zero sign, e.g. 007 is read as 7.

Separator Characters

In INTERLISP each of the following characters separates S-expressions:

- a blank

- a bracket, i.e.), (, > or <

- the quote sign, i.e. ´

- the string indicator, i.e. "

Signs acting as separators in PLL are

- all INTERLISP separators

- the colon, i.e. X:Y is the same as X : Y and

- the comma, i.e. X,Y is the same as X , Y .

6.6.3. The EDITOR

The EDITOR is a screen oriented, syntax directed editor for sorted logical formulae, which evolved as the MKRP-Procedure developed: It is based on the practical experience of almost five years of use. Although the MKRP system is at best a pilot implementation for purely research purposes it turned out that the use of the system became intolerably complicated even for the experienced user without strong software support.

The EDITOR provides about fourty commands the major ones are now summarized to get a feeling for what can be done. The formulae in the area to be edited are divided into two kinds: the active area and the passive area. Roughly the passive area consists of the most recently inserted formulae and the symbol table, the semantic checks and others have not been performed or updated for these formulae. The problem is, that the system maintains a record of everything that is going on, in particular a listing of the intended meaning of the symbols occurring in the formulae, their arity and so on and this record is built up only for the active area.

INSERT <formula> inserts a syntactically and semantically correct formula as the last one into the active area

DELETE - deletes all formulae

DELETE 2 deletes formula No 2

DELETE 3 - deletes all formulae from the third one onwards.

+SHIFT shifts the first formula of the passive area into the active area

++SHIFT shifts all formulae of the passive area into the active area

-SHIFT shifts the last formula of the active area into the passive area

--SHIFT shifts all

EDIT 3 calls the LISP-editor for the formula No 3

READ <file> reads a <file>, which must have been created by the WRITE command

WRITE <file> the content of the edited area will be saved on <file> so that it can be restored later with the READ command

SWITCH 3 4 formula 3 and formula 4 in the passive area are exchanged

UNDO "Undoes" the last destructive command

REPLACE <OLD><NEW> replaces formula <OLD by formula <NEW> in the active and passive area

PRINT <file> writes the symbol table and all formulae in readable format onto <file>

SHOW <Symbol 1> ... <Symbol m>
<Kind 1>..... <Kind n> displays all
<symbol 1> to <symbol m> or every symbol of the kind <Kind 1> to <Kind n>
There are five kinds of symbols:
SORT
RELATION
CONSTANT
FUNCTION
PREDICATE

PREFIX	displays the compiled prefix form of the formulae
INFIX	displays the (uncompiled) input formulae in infix
STATE	displays the distribution of formulae in the active and passive areas
HELP <COMMAND>	explains the use of <COMMAND>
H HELP	even more explanation
OK	terminates the user session with the EDITOR

6.6.4. The Output Facilities

When the user sets the options for a proof, he can specify a file (proof file), upon which the course of the search for the proof is recorded. At run-time as little time as possible should be lost. Therefore all the information which is important for the final proof documentation is written onto the proof file in a very fast way and compact format. Thus a dense and complicated sequence of 'words' is generated in the file containing all the essential information (the events; for a similar approach see [LMO82])

This information includes:

- the user's input: axioms, theorems, options

- preliminary transformation: axioms and theorems in the internal language or the symbol table of all symbols used

- proof steps: parent clause(s), type of operation, resulting clause, unifier

- statistical data: this can be printed also in order to analyze the logical performance (number of clauses, of links, G- and R-penetrance etc.)

Only after the proof is finished the data on the proof file is further processed such that the course of the proof can be exactly reconstructed.

There are four levels of abstraction upon which the search for a proof can be recorded.

DUMP: the complete dump of the proof file

IMPLEMENTATIONAL: a very detailed (but readable) record of everything that happened (and why it happened) during the search for a proof

LOGICAL: a pretty printed record of the resolution steps that were performed

PROOF: Similar to LOGICAL but only those steps are recorded that were actually used in the proof

Current work transforms the PROOF in a first transformation into a natural deduction style proof and a second transformation is then to translate this natural deduction proof into "natural language", i.e. a format close to a human mathematician's way of expressing his reasoning.

6.6.5. Traces

For debugging purposes some of the modules have additional protocol functions. These functions can only be activated fron the LISP level and not from the ATP operating system.
In order to activate a trace function, a programmer calls the LISP interpreter and writes:
(mod-TRACE ON) or (mod-TRACE T), where mod is an abbreviation of

the module name.
(mod-TRACE ON) directs the trace to the general protocol file and (mod-TRACE T) to the terminal.

With (mod-TRACE OFF) the trace is switched off.

The following trace functions are available:

1. Normalization: NORM-TRACE

   Detailed information about the transformations of the prefix formula to clausal form are printed.

2. Selection: SEL-TRACE

   Information about the activation of the operation blocks and deduction and reduction steps are printed.

3. Terminator: TERM-TRACE

   The name of the currently examined clause together with statistical values about the compatibility test of unifiers are printed. In addition all internally generated unitclauses are listed.

4. Connection graph: CG-TRACE

   All objects (clauses, links, unifiers) which are removed from the graph together with a short explanation why they are removed, are listed.

6.6.6. A Test Run

The following protocol lists a typical session: the first set of instructions is used to set up the database etc. The second set is the final output protocol. The theorem to be proved is P. Andrew's (small) challenge problem:

$$(\forall x, Qx \equiv \exists y.\ Qy) \equiv (\exists x \forall y\ Qx = Qy)$$

The session as it appears on the Screen:

```
DO ATP,C50
%  P500 LISPV4/    /81-09-29 LOADED
ATP SYSTEM: MARKGRAF KARL REFUTATION PROCEDURE, UNI KARLSRUHE
            VERSION: 21-DEC-83

DEUTSCH, ENGLISH? (D/E)

e
THE DIALOGUE LANGUAGE IS ENGLISH FOR THIS ATP-SESSION.
 H[ELP]         PRINTS A LIST OF ALL AVAILABLE COMMANDS.
 H[ELP] <COM>   EXPLAINS THE COMMAND <COM>.

a

o
THIS IS THE OPTIONS-MODULE. FOR ASSISTANCE TYPE H[ELP].
&

&

pr:proof.file pr.andrews.small
PR:PROOF.FILE = PR.ANDREWS.SMALL
gen:splitting 1
GEN:SPLITTING = 1
&

ok

a

p
CALLING THE FORMULA EDITOR. PLEASE EDIT THE AXIOM FORMULAS:
 H      PRINTS A LIST OF ALL AVAILABLE EDITOR-COMMANDS.
 OK     TERMINATES THIS EDITOR-SESSION.

>
```

```
ok
EDITOR TERMINATED.
WARNING: THERE ARE NO AXIOM FORMULAS!
CALLING THE FORMULA EDITOR. PLEASE EDIT THE THEOREM FORMULAS:

>

r $kinf47.f.th.andrews.small
DATE OF FILE GENERATION: 15-JUL-83 12:49:01
SYMBOL-TABLE AND ONE FORMULA LOADED FROM FILE $KINF47.F.TH.ANDREWS.SMALL.
INSERTED IN THE ACTIVE AREA AT POSITION 1.

>

l-
* 1 *   ((ALL X Q (X)) EQV (EX Y Q (Y))) EQV (EX X ALL Y Q (X) EQV Q (Y))

>

ok
EDITOR TERMINATED.


---------------------   NOW THE PROOF BEGINS:   ----------------------

***** REFUTATION OF SPLIT PART 1 INITIATED. ////////////////////////////////

THEOREM       53 MSEC LINKS:2    +0   -0   CLAUSES:4   +0  -0  STORE:265 49
REDUCED       25 MSEC LINKS:2    +0   -0   CLAUSES:4   +0  -0  STORE:265 49
THEOREM      168 MSEC LINKS:10   +0   -0   CLAUSES:4   +0  -0  STORE:264 49
REDUCED       69 MSEC LINKS:10   +0   -0   CLAUSES:4   +0  -0  STORE:264 49
THEOREM       91 MSEC LINKS:12   +0   -0   CLAUSES:4   +0  -0  STORE:263 49
REDUCED      440 MSEC LINKS:12   +1   -1   CLAUSES:4   +0  -0  STORE:261 49
MARKED       135 MSEC LINKS:12   +0   -0   CLAUSES:4   +0  -0  STORE:260 49
STEP 1       109 MSEC LINKS:16   +4   -0   CLAUSES:5   +1  -0  STORE:260 49
             210 MSEC LINKS:5    +1   -12  CLAUSES:3   +0  -2  STORE:259 49


***** REFUTATION DETECTED BY TERMINATOR! *****

STEP 2       102 MSEC LINKS:7    +2   -0   CLAUSES:4   +1  -0  STORE:257 49
              11 MSEC LINKS:7    +0   -0   CLAUSES:4   +0  -0  STORE:257 49
STEP 3        48 MSEC LINKS:7    +0   -0   CLAUSES:5   +1  -0  STORE:256 49

***** REFUTATION OF SPLIT PART 1 SUCCEEDED. ////////////////////////////////


***** REFUTATION OF SPLIT PART 2 INITIATED. ////////////////////////////////

THEOREM      345 MSEC LINKS:21   +0   -0   CLAUSES:4   +0  -0  STORE:252 49
REDUCED      168 MSEC LINKS:13   +0   -8   CLAUSES:3   +0  -1  STORE:252 49
THEOREM      105 MSEC LINKS:16   +0   -0   CLAUSES:3   +0  -0  STORE:251 49
REDUCED      838 MSEC LINKS:16   +4   -4   CLAUSES:3   +0  -0  STORE:248 49
MARKED       103 MSEC LINKS:16   +0   -0   CLAUSES:3   +0  -0  STORE:247 49
STEP 1       358 MSEC LINKS:22   +14  -8   CLAUSES:4   +1  -0  STORE:246 49
             156 MSEC LINKS:11   +1   -12  CLAUSES:3   +0  -1  STORE:245 49
STEP 2       131 MSEC LINKS:16   +5   -0   CLAUSES:4   +1  -0  STORE:243 49
             220 MSEC LINKS:3    +1   -14  CLAUSES:2   +0  -2  STORE:242 49
```

```
***** REFUTATION DETECTED BY TERMINATOR! *****

STEP 3        104 MSEC LINKS:6    +3   -0   CLAUSES:3   +1  -0  STORE:241 49
               12 MSEC LINKS:6    +0   -0   CLAUSES:3   +0  -0  STORE:241 49
STEP 4         49 MSEC LINKS:6    +0   -0   CLAUSES:4   +1  -0  STORE:240 49

***** REFUTATION OF SPLIT PART 2 SUCCEEDED. ////////////////////////////////

******************   PROOF SUCCESSFULLY TERMINATED.   ******************

@

pr pr.andrews.small

TCOMPL FORMAT IWOC.ATP.PROT.EXECUTE CREATED 10-DEC-83 08:50:32  FILENAME: IWOC
.ATP.PROT.EXECUTE.COM.00
ATP.PROT.EXECUTECOMS


TCOMPL FORMAT IWOC.ATP.PROT.PREPARE CREATED 10-DEC-83 08:51:26  FILENAME: IWOC
.ATP.PROT.PREPARE.COM.00



 ATP.PROT.PREPARECOMS


 TCOMPL FORMAT IWOC.ATP.PROT.DATASTRUCTURE CREATED 10-DEC-83 08:56:42  FILENAME
 : IWOC.ATP.PROT.DATASTRUCTURE.COM.00
 ATP.PROT.DATASTRUCTURECOMS


 TCOMPL FORMAT IWOC.ATP.PROT.PRINT CREATED 10-DEC-83 09:01:24  FILENAME: IWOC.A
 TP.PROT.PRINT.COM.00
 ATP.PROT.PRINTCOMS
 PROTOKOLL AUF FOLGENDE DATEI ERZEUGT: PR.ANDREWS.SMALL.LIST.00

 @

 ex
```

The Final Protocol

```
*********************************************
*                                           *
*   ATP-SYSTEM :  M K R P , UNI KARLSRUHE   *
*                                           *
*      VERSION : 21-DEC-83                  *
*         DATE : 18-JAN-84  20:28:58        *
*                                           *
*********************************************
```

FORMULAE GIVEN TO THE EDITOR
============================

```
THEOREMS :    ((ALL X Q (X)) EQV (EX Y Q (Y)))
              EQV
              (EX X ALL Y Q (X) EQV Q (Y))
```

REFUTATION OF SPLITPART 1

```
INITIAL CLAUSES :        * T1    : ALL X:ANY   -Q(C_1) -Q(X)
                         * T2    : ALL X:ANY   +Q(X) +Q(C_2)
                         * T3    : ALL X:ANY   +Q(C_3) -Q(X)
                         * T4    : ALL X:ANY   -Q(C_3) +Q(X)

T1 1_REPL_2          -->  * T1.1 : -Q(C_1)
T2 2_REPL_1          -->  * T2.1 : +Q(C_2)
T4,2  +  T1.1,1      -->  * R1   : -Q(C_3)
T3,1  +  R1,1        -->  * R2   : -Q(C_2)
T2.1,1  +  R2,1      -->  * R3   :
```

REFUTATION OF SPLITPART 2

```
INITIAL CLAUSES :        * T1    : ALL X,Y:ANY  +Q(X) -Q(Y)
                           T2    : -Q(C_4) +Q(C_5)
                         * T3    : ALL X:ANY  -Q(X) -Q(F_1(X))
                         * T4    : ALL X:ANY  +Q(X) +Q(F_1(X))

T4,1  +  T1,2      -->   * R1    : ALL X,Y:ANY  +Q(F_1(X)) +Q(Y)
R1 1_REPL_2        -->   * R1.1  : ALL X:ANY  +Q(F_1(X))
R1.1,1  +  T1,2    -->   * R2    : ALL X:ANY  +Q(X)
R2,1  +  T3,1      -->   * R3    : ALL X:ANY  -Q(F_1(X))
R2,1  +  R3,1      -->   * R4    :
```

Q. E. D.

Literatur:

[AH72] Andersen B., Hayes P.
An Arraignment of Theorem Proving or the Logicians Folly. Comp. Logic Memo 54, Univ. of Edinburgh

[AN68] Andrews P.
Resolution with Merging. JACM, vol 15, no 3, 1968

[AN70] Anderson R.
Completeness Results for E-Resolution. Proc Spring Joint Conf., 1970, 653-656

[AN81] Andrews P.B.
Theorem Proving via General Matings. J.ACM 28:2, 193-214, 1981

[B75] Bruynooghe M.
The Inheritance of Links in a Connection Graph. Report CW 2, Katholieke Universiteit Leuven, 1975

[BA73] Baxter L.D.
An Efficient Unification Algorithm. Univ. of Waterloo, Techn. Report CS-73-23, 1973

[BB75] Ballayntyne M., Bledsoe W.
Autom. Proofs and Theorems in Analysis using nonstandard Techniques. ATP-23, 1975, Univ. of Texas

[BBH72] Bledsoe W., Boyer B., Hemmeman
Computer Proofs of Limit Theorems. J.Art. Int., vol 3, 1972

[BES81] Bläsius K., Eisinger N., Siekmann J., Smolka G., Herold A.,Walther C.
The Markgraph Karl Refutation Procedure. Proc. IJCAI-81, Vancouver, 1981

[BI81] Bibel W.
On Matrices with Connections. JACM 28:4, 633-645, 1981

[BL71] Bledsoe W.
Splitting and Reduction Heuristics in ATP. J. Art. Int., vol 2, 1971

[BL77] Bledsoe W.
A maximal method for set variables in ATP. ATP-33, Univ. of Texas, 1977

[BM78] Boyer R.S., Moore J.S.
A Computational Logic. Academic Press, 1978

[BM79] Boyer R.S., Moore J.S.
A Computational Logic. Academic Press, 1979

[BR75] Brand D.
Proving Theorems with the Modification Method. SIAM Journal of Comp., vol 4, no 4, 1975

[BT75] Bledsoe W., Tyson M.
The UT Interactive Prover. Univ. of Texas, ATP-7, 1975

[CHL73] Chang C.L., Lee R.C.
Symbolic Logic and Mechanical Theorem Proving. Academic Press, 1973

[CS79] Chang C.L., Slagle J.R.
Using Rewriting Rules for Connection Graphs to Prove Theorems. Artificial Intelligence 12 (1979) 159-180

[CHA78] Champeaux D. de
A Theorem Prover Dating a Semantic Network. Proc. of AISB/GI Conference, Hamburg, 1978

[DE71] Deussen P.
Halbgruppen und Automaten. Springer 1971

[DI79] Digricoli V.J.
Resolution by Unification and Equality. Proc 4th Workshop on Automated Deduction, 1979, Texas

[EI81] Eisinger N.
Subsumption and Connection Graphs. IJCAI-81, Vancouver, 1981

[EI82] Eisinger N.
A Pragmatic Module Concept for INTERLISP. Univ. Karlsruhe, Bericht 23, 1982

[EW83] Eisinger N., Weigele M.
A Technical Note on Splitting and Clausal Form Algorithms. Proc. GWAI-83, Springer Fachberichte, 1983

[GB69] Guard, Oglesby, Bennet, Settle
Semi-Automated Mathematics. JACM 16, no 1, 1969

[GEN34] Gentzen G.
Untersuchungen über das logische Schließen. Mathematische Zeitschrift 39, 1934

[GIL58] Gilmore P.C.
An Addition to "Logic of Many-Sorted Theories". Compositio Mathematica 13, 1958

[HAI57] Hailperin T.
A Theory of Restricted Quantification I. The Journal of Symbolic Logic 22, 1957

[HAY71] Hayes P.
A Logic of Actions. Machine Intelligence 6, Metamathematics Unit, Univ. of Edinburgh, 1971

[HE72] Hewitt C.
Description and Theoretical Analysis of PLANNER. AI-TR-258, MIT

[HEN72] Henschen L.J.
N-Sorted Logic for Automatic Theorem Proving in Higher-Order Logic. Proc. ACM Conference, Boston, 1972

[HER30] Herbrand, J.
Recherches sur la théorie de la démonstration (Thése Paris). Warsaw, 1930, chapter 3.
Also in "Logical Writings" (W.D. Goldfarb ed.), D. Reidel Publishing Company, 1971

[HO80] Huet, Oppen
Equations and Rewrite Rules: A Survey, SRI Technical Report, CSL-11, 1980

[HR78] Harrison M.C., Rubin N.
Annother Generalization of Resolution. JACM, vol 25, no 3, 341-351, July 1978

[IDE64] Idelson, A.V.
Calculi of Constructive Logic with Subordinate Variables. American Mathematical Society Translations (2) 99, 1972 - Translation of Trudy Mat. Inst. Steklov. 72, 1964

[KI69] King J.
A Program Verifier. Ph,D. Carnegie Mellon, 1969

[KO75] Kowalski G.
A Proof Procedure using Connection Graphs. JACM, vol 22, no 4, 1975

[LE78] Lewis H.R.
Renaming a Set of Clauses as Horn Set. JACM 25, 1, 1978

[LMO82] Lusk L., McCune W., Overbeck R.
Logic Machine Architecture: Kernel Functions and Inference Rules. Proc. CADE-82, Springer Lect. Notes, vol 138, 1982

[LOV78] Loveland D.W.
Automated Theorem Proving: A Logical Basis. North-Holland Publishing Company 1978

[MH56] Montague R., Henkin L.
On the Definition of Formal Deduction. The Journal of Symbolic Logic 21, 1956

[MI83] Mischke G.
An Interlisp Function Measurement System. Univ. Karlsruhe, 1983

[MM79] Martelli H., Montaneri U.
An Efficient Unification Algorithm. Univ. of Pisa, Techn. Report, 1979

[MO69] Morris J.B.
E-Resolution: An Extension of Resolution to include the Equality Relation. Proc IJCAI, 1969, 287-294

[NO77] Nelson, Oppen
Fast Decision Algorithms based on Congruence Closure. Stanford Univ., STAN-CS-77-646

[OBE62] Oberschelp A.
Untersuchungen zur mehrsortigen Quantorenlogik. Mathematische Annalen 145 ,1962

[OH82] Ohlbach H.J.
The Markgraph Karl Refutation Procedure: The Logic Engine. Interner Bericht 24/82, Univ. Karlsruhe, 1982

[PS81] Peterson G., Stickel M.
Complete Sets of Reductions for Equational Theories with Complete Unification Algorithms. JACM 28:2, 1981

[PW78] Paterson M., Wegman M.
Linear Unification. Journal of Comp. and Syst., 16, 1978

[RO71] Robinson J.A.
Computational Logic: The Unification Computation. Machine Intelligence, vol.6, 1971

[RW69] Robinson G., Wos L.
Paramodulation and TP in first-order Theories with Equality. Machine Intelligence 4, 135-150

[SCH38] Schmidt A.
Über deduktive Theorien mit mehreren Sorten von Grunddingen. Mathematische Annalen 115, 1938

[SCH51] Schmidt A.
Die Zulässigkeit der Behandlung mehrsortiger Theorien mittels der üblichen einsortigen Prädikatenlogik. Mathematische Annalen 123, 1951

[SH78] Shostak R.E.
An Algorithm for Reasoning About Equality. JACM, July 1978, vol 21, no 7

[SI69] Sibert E.E.
A machine-oriented Logic incorporating the Equality Axiom. Machine Intelligence, vol 4, 1969, 103-133

[SI76] Sickel S.
A Search Technique for Clause Interconnectivity Graphs. IEEE Trans. on Computers C-25(8), 823-835, 1976

[SIG76] Sigart
ACM Special Interest Group on AI, April 1981, no76

[SIG80] dito
no80

[SM82] Smolka G.
Completenes and Confluence Properties of Kowalski´s Clause Graph Calculus, Univ. Karlsruhe, Techn. Report 31/82, 1982

[SS81] Siekmann J., Szabó P.
Universal Unification and Regular ACF-Theories. Proc. of 7th IJCAI, Vancouver 1981

[SS82] Siekmann J., Szabó P.
Universal Unification and a Classification of Equational Theories. 6th Conference on Automated Deduction, Springer LNCS 138, 1982

[SW79] Siekmann J., Wrightson G.
Paramodulated Connectiongraphs. Acta Informatica, 1979

[SZ69] Szabo
"The Logic Writings of G. Genken"

[VA75] van Vaalen J.
An Extension of Unification to Substitutions with an Application to ATP. Proc of 4th IJCAI, 1975

[WA82] Walther Ch.
PLL - A First Order Language for an Automated Theorem Prover. Bericht 35/82, Univ. Karlsruhe, 1982,

[WA83] Walther Ch.
A Many Sorted Calculus based on Resolution and Paramodulation. Bericht 34/82, Univ. Karlsruhe, 1983

[WAN52] Wang H.
Logic of Many-Sorted Theories. The Journal of Symbolic Logic 17, 1952

[WEY77] Weyhrauch R.W.
FOL: A Proof Checker for First-Order Logic. MEMO AIM-235.1, Stanford Artificial Intelligence Laboratory, Stanford University, 1977

[WM76] Wilson G., Minker J.
Resolution, Refinements and Search Strategies: A Comparative Study, IEEE Trans. on Comp., vol-C-25, no 8, 1976

[WR64] Wos L., Carson D., Robinson G.
The Unit Preference Strategy in Theorem Proving. AFIPS Joint Comp. Conf., vol 26, 1964

[WR67] Wos L., Carson D., Robinson G., Shallar L.
The Concept of Demodulation in Automated Theorem Proving, JACM, vol 14, no 4, 1967

[WR73] Wos L., Robinson G.
Maximal Models and Refutation Completeness: Semidecision Procedures in Automated Theorem Proving. In "Wordproblems" (W.W. Boone, F.B. Cannonito, R.C. Lyndon, eds.), North-Holland Publishing Company, 1973