The Markgraf Karl
Refutation Procedure

H.J. Ohlbach & J. Siekmann
SEKI Report SR-89-19

The Markgraf Karl
Refutation Procedure

H.J. Ohlbach & J. Siekmann
Fachbereich Informatik, Universität Kaiserslautern
Postfach 3049, D-6750 Kaiserslautern, W.-Germany

THE MARKGRAF KARL REFUTATION PROCEDURE

Hans Jürgen Ohlbach, Jörg H. Siekmann

FB Informatik, University of Kaiserslautern

Postf. 3049

D-6750 Kaiserslautern

West Germany

net: ohlbach(siekmann)@uklirb.informatik.uni-kl.de

Abstract The goal of the *MKRP project* is the development of a theorem prover which can be used as an inference engine in various applications, in particular it should be capable of proving significant mathematical theorems. Our first implementation, the *Markgraf Karl Refutation Procedure*[1] *(MKRP)* realizes some of the ideas we have developed to this end. It is a general purpose resolution based deduction system that exploits the representation of formulae as a graph (clause graph). The main features are its well tailored selection components, heuristics and control mechanisms for guiding the search for a proof.

This paper gives an overview of the system. It summarizes and evaluates our experience with the system in particular, and the logics we use as well as the clause graph approach: as 1990 marks the fifteenth birthday of the system, the time may have come to ask: "Was it worth the effort?"

Key words: automated deduction, resolution, clause graphs, sorted logics.

---

[1]Markgraf Karl was the founder of Karlsruhe where the project started.

1. INTRODUCTION

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 deduction systems attained a certain level of performance, which will not be significantly improved by:

- developing more and more intricate refinements (like set of support, unit preference, linear resolution, ...), whose purpose is to syntactically filter the search space, nor by

- using different calculi (like natural deduction, sequence calculi, matrix methods etc.),

although this was the main focus in automated deduction research in the past and of course it still has its merits today. The relative weakness of current deduction systems as compared to human performance is due to a large extent to their lack of the rich mathematical and extra mathematical knowledge that human mathematicians have; in particular, knowledge about the subject and knowledge of how to find proofs in that subject.

Secondly, deduction systems based on a single (or a set of) general purpose inference rule(s) perform even the most elementary calculations, like numerical evaluation or algebraic simplification, with these inference rules, wasting time and space on tasks, which can be done more efficiently by special purpose algorithms. Hence a typical resolution style proof will contain a few important deduction steps, that represent the essential idea of the proof and that a human mathematician might care to communicate to his colleagues, and ten times as many trivial steps. Apart from this mixed and generally wrong level of abstraction in communicating proofs, there is the more important issue, that the amount of search is the same for "garbage and gold", as the system has no way of distinguishing between them.

And finally, to a lesser, but still important extent, the relative weakness of deduction systems can often be attributed to the insufficient emphasis, which in the past has been put onto the software engineering aspects 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.

The overall goal of the MKRP project was stated in 1975 by the following three claims: it is possible to build a deduction system and augment it by appropriate heuristics and domain-specific knowledge such that

- 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 deduction systems of the past. Consequently

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

- Such a deduction system will establish an unprecedented leap in performance over previous systems expressed in terms of the difficulty of the theorems it can prove.

With this goal in mind we started different lines of research: the development of methods for incorporating algorithmically tractable concepts into the deduction process, the development of a heuristically controlled proof procedure based on Kowalski´s connection graph[1] procedure [Kowalski 75] and the investigation of knowledge representation and reasoning in more complex mathematical domains.

The first line of research lead to the development of sorted logics for representing fragments of set theory and taxonomical hierarchies [Walther 83, 87, Schmidt-Schauss 88], equality reasoning methods [Bläsius 86, Bläsius & Siekmann 88, 90], unification theories and algorithms [Siekmann 89], resolution calculi for "state transition logics" such as modal logics, temporal logics, epistemic logics etc. [Ohlbach

---

[1] We prefer the term "clause graph".

88, 89] and the automation of induction proofs [Biundo et al 86, Walther 88] in the Boyer and Moore style [Boyer &Moore 79].

For the actual proof search we chose the clause graph approach as the basic paradigm. The advantages of the clause graphs are the explicit representation of the potential resolution steps as links of a graph. By selecting an appropriate link, heuristics can determine the most promising step according to many strategic, logical and topological criteria.

Due to limited resources and other reasons we could not spend as much time as we wanted for the investigation of global mathematical knowledge and strategies. Although a significant part of an automata theory textbook [Deussen 71] was encoded and proved automatically and a general representation formalism for mathematical knowledge has been developed in the meantime, we are still at the beginning and do not expect significant results in the near future.

Our aim is to integrate all methods into a single albeit large deduction system, which is strong enough to solve nontrivial mathematical problems. The result of this effort is the MKRP system. As it is always the case with long lasting software developments, which are continuously modified, earlier design decisions turn out to be wrong, new programming paradigms arise, the system grows larger and larger and the maintenance becomes more and more problematic. Eventually is is better to freeze the state of the system and to start a new development from scratch. This point has come for the MKRP system with the availability of object oriented programming on the software technology side and the shift from a "generative calculus" where resolvants are explicitly generated to a pure graph search.

In this report which was solicited to mark the 65th birthday of A. Robinson in a Festschrift, we take the chance to summarize our experience with clause graph theorem proving in general and the MKRP system in particular. Due to the lack of space we have to omit our work on induction theorem proving[1] and theorem proving in nonclassical logics. Theory unification and many sorted logics can only briefly be sketched.

We shall present evidence that the first two of our original claims have indeed been achieved and the system performs (in these terms) substantially better than *any other of the currently available deduction systems*. The final, albeit essential claim has not been achieved with the MKRP implementation: after a decade of racing against the strongest systems, most notably the deduction system of Argonne National Laboratories, where the pendulum swung sometimes to this side and then to the other side of the Atlantic, it is presumably fair to say that these systems all perform more or less in the same bracket, irrespective of what the latest statistics are. As heated (and important) as the discussions about the latest findings may be, this view "through the microscope" clouds the general fact, that the field as such has advanced enormously in the last decade - but there was no breakthrough by one single system.

In the last chapter we try to evaluate the system and to propose ways to further develop the field of automated theorem proving. After having introduced the basic logical concepts used in the MKRP system in chapter two, we present in chapter three an extended clause graph procedure which goes beyond Kowalski´s original version. In chapter four some new ways to handle equality reasoning are discussed. The architecture of the MKRP system is described in more detail in chapter five.

---

[1] After the "emigration" of Jörg Siekmann from Karlsruhe the rest team pursued the automation of induction theorem proving under the leadership of Peter Deussen and Christoph Walther. They developed the INKA system [Biundo et al 86] which has as its first-order kernel a modified version of the MKRP system.

2. FOUNDATIONS

In the following we shall fix our notions and notation and summarize some basic techniques of automated deduction. These are consistent with most standard textbooks [Smullyan 68, Chang & Lee 73, Loveland 78, Wos et al 84, Bläsius & Bürckert 89], but take some more recent developments of the field into account. In section 2.4. we shall introduce several *many sorted* variants of predicate logic, which we have developed as a very first approximation to incorporate elementary set theory and taxonomical hierarchies. They have been used as the basic object languages of the MKRP-System.

2.1 Basic Notions and Notation

The language used in this report is that of first-order predicate logic (PL1) which we assume the reader to be familiar with. 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* symbols.

Individual constants and variables are terms as well as n-ary 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 $\neg$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 clause with just one literal is called a unit clause or just a *unit*.

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 finitely representable and denoted as a set of pairs $\delta = \{(v_1 \mapsto t_1) \ldots (v_n \mapsto t_n)\}$ 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, τ for substitutions. A substitution σ is called a *unifier* for two terms s and t iff $\sigma s = \sigma t$ (two atoms L and K iff $\sigma(L) = \sigma(K)$); σ is called 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 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 $\sigma L = K$.

A substitution ρ is a *variable renaming* iff $\rho(x)$ is a variable for all x and ρ is injective. A *weak unifier* for two terms s and t is a tupel (ρ, σ) such that ρ is a variable renaming and $\sigma \rho s = \sigma t$. Example: x and f(x) are not unifiable, but they are weakly unifiable with a weak unifier $(\{x \mapsto y\}, \{y \mapsto f(x)\})$.

A Tarski interpretation of a PL1 formula consists of a *universe* or *domain* of discourse and an assignment of functions to function symbols and relations to predicate symbols. Together with assignments of domain elements to variable symbols an interpretation can be used to evaluate terms to domain elements and formulae to truth values. An interpretation which *satisfies* a formula $\mathcal{F}$, i.e. it evaluates $\mathcal{F}$ to the truth value TRUE is a *model* for $\mathcal{F}$. A formula $\mathcal{G}$ is a (semantical) consequence or a *theorem* of a formula $\mathcal{F}$, written $\mathcal{F} \vDash \mathcal{G}$, iff $\mathcal{G}$ holds in all models of $\mathcal{F}$. This is equivalent to the condition that the negated theorem together with the axioms are unsatisfiable, i.e. have no model.

In PL1, a satisfiable set $\mathcal{A}$ of formulas can be uniquely associated with the class $\mathcal{M}$ of its models, i.e., of the interpretations satisfying all the formulae in $\mathcal{A}$. This class of interpretations in turn uniquely corresponds to a maximal (in general infinite) set $\mathcal{T}$ of formulas that are satisfied by all interpretations in $\mathcal{M}$. The set $\mathcal{T}$ is maximal in the sense that any additional formula would restrict the class $\mathcal{M}$ of models because it would be falsified by at least one model of $\mathcal{A}$. By definition, $\mathcal{T}$ is just the set of consequences of

$\mathcal{A}$. From this perspective, $\mathcal{M}$ and $\mathcal{T}$ contain the same information, and both are often called the *theory* of $\mathcal{A}$. Since different sets of formulae may have the same models, any specific $\mathcal{A}$ is just one alternative in defining the theory. $\mathcal{A}$ is also called a *presentation* or *axiomatization* of the theory.

For a given theory $\mathcal{T}$ and a formula $\mathcal{F}$, the $\mathcal{T}$-*models* of $\mathcal{F}$ are simply all those models of $\mathcal{T}$ that are models of $\mathcal{F}$ as well. The notions $\mathcal{T}$-*consequence*, $\mathcal{T}$-*satisfiable*, $\mathcal{T}$-*unsatisfiable*, etc. are then defined correspondingly.

An important theory is the theory of *equality*. Equality can either be axiomatized with the usual axioms for equality - reflexivity, symmetry, transitivity and the substitution axioms for function and predicate symbols - or it can be built into the logic by restricting the interpretations to those interpretations, usually called E-interpretations, where the equality symbol = is mapped to the identity on the domain.

A clause is a *tautology* if it is true in every interpretation. It is a $\mathcal{T}$-*tautology* in a given theory $\mathcal{T}$ if it is true in every $\mathcal{T}$-interpretation, where a $\mathcal{T}$-*interpretation* is an interpretation that is a model for $\mathcal{T}$. Syntactically, a tautology contains two complementary equal literals, i.e. the atoms are equal and the signs are different. The syntactical structure of $\mathcal{T}$-tautologies is more difficult to recognize. If for instance $\mathcal{T}$ contains the equality theory and additionally an axiom to express the commutativity of the function symbol f: {f(x,y) = f(y,x)}, then the clause {P(f(a,b)), ¬P(f(b,a)) } is not a tautology, but a $\mathcal{T}$-tautology.

A clause C *subsumes* a clause D if every model of C satisfies D. C $\mathcal{T}$-*subsumes* D in a theory $\mathcal{T}$ if every $\mathcal{T}$-model of C satisfies D. For example {P(x)} subsumes {P(a), Q}, {¬P(x), P(f(x))} subsumes {¬P(y), Pf(f(y)))}. {R(f(x a))} $\mathcal{T}$-subsumes {R(f(a b))}, Q} in the theory described above.

Subsumption in this general form, which is in fact an implication, is undecidable [Schmidt-Schauss 86], therefore only restricted notions of subsumption are suitable for an implementation. If for example D is an instance of C, which can be syntactically recognized, and C has not more literals than D then C subsumes D.

2.2 Theory Resolution

The syntactical inference rules John Alan Robinson invented for PL1 are resolution and factoring [Robinson 65]. The resolution rule derives from a clause L ∨ A and a clause L' ∨ B, where the literals L and L' are complementary unifiable with unifier σ, a new clause, the *resolvent*, σ(A ∨ B). The factorization rule instantiates a clause such that at least two literals become identical. Both rules operate on the most general level. That means all possible inferences are always subsumed by a resolvent or factor respectively. Therefore the branching rate in the search space is always finite, compared to the usually infinite branching rate of other calculi.

Although this feature of the resolution rule brought a substantial improvement to automated theorem proving, certain axioms may still lead a resolution system to go astray, a problem that was recognized as early as the first deduction systems were build. Alan Robinson claimed in 1967 [Robinson 67], that substantial progress could be achieved - in fact "a new plateau" - if these troublesome axioms would be taken out of the database and "built into" the rules of inference.

Theory resolution is a general scheme to build axioms into the rules of inference, i.e. to exploit information about the meaning of predicate symbols and function symbols directly within the calculus by using specially tailored inference rules instead of axioms for these symbols. It was proposed by Mark Stickel at SRI [Stickel 85]. It is in fact a generalization of many special cases that were known before by different names such as E-resolution, T-unification etc. Some of them will be discussed in later paragraphs.

As a motivation for the approach, let us recall the justification for the soundness of the resolution rule:

clause 1: $L, K_1, ..., K_n$
clause 2: $\neg L, M_1, ..., M_m$

_____

resolvent: $K_1, ..., K_n, M_1, ..., M_m$

The essential argument for the parent clauses' entailing the resolvent is that an interpretation satisfying the literal L falsifies ¬L. The crucial point is that no interpretation can satisfy both L and ¬L. This is the case for two literals whenever they meet the purely syntactic condition of being complementary, i.e., if they have opposite signs, equal predicate symbols, and equal term lists.

In many cases one can generalize this syntactic notion of complementarity by utilizing the fact that not arbitrary interpretations need to be considered, but only the interpretations of a certain theory. For example, a set of formulas might contain axioms for a predicate symbol <, such that interpretations can be models only if they associate with < a strict ordering on the universe. Due to the properties of strict ordering relations, no such interpretation can satisfy both a < b and b < a. These two literals are not syntactically complementary, but, as it were, semantically complementary in the assumed context, where the following derivation step would also be sound:

clause 1: $a < b, K$
clause 2: $b < a, M$

_____

resolvent: K, M

As a further generalization, we can even abandon the restriction to two parent clauses. No interpretation of the assumed class can satisfy each of the literals a < b and b < c and c < a. Analogous to the justification for the simple resolution rule, only with more cases, the following step can also be shown to be sound:

clause 1: $a < b, K$
clause 2: $b < c, M$
clause 3: $c < a, N$

_____

resolvent: K, M, N

Thus the idea is to proceed from the special case of two syntactically complementary resolution literals to an arbitrary set of *resolution literals* such that no interpretation of a theory can satisfy all of them.

Now the general scheme for *total theory resolution* is as follows: let $\mathcal{T}$ be a theory and let $C_1, ..., C_n$ be clauses, each of them containing a literal L_i such that for a substitution σ the conjunction of all these literals' σ-instances is $\mathcal{T}$-unsatisfiable. The σ-instances of the union of these n clauses minus the resolution literals constitutes a $\mathcal{T}$-resolvent. This clause is a $\mathcal{T}$-consequence of the formula $C_1 \wedge ... \wedge C_n$.

Figure 2-1: Total Theory Resolution
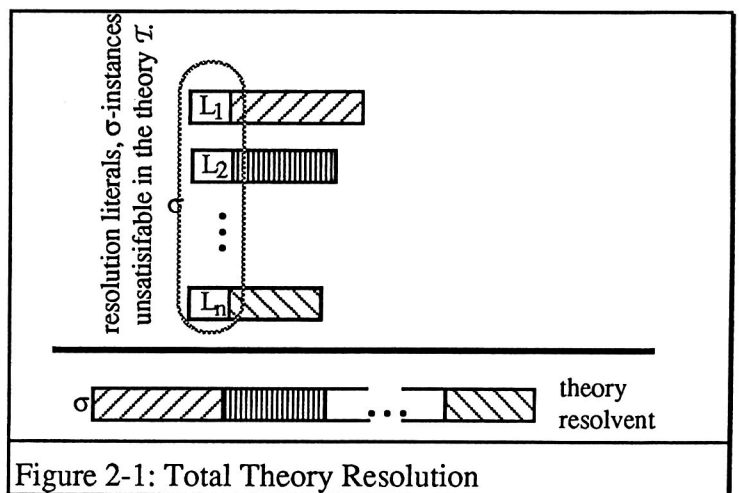
The concept of theory resolution allows a much more natural and efficient treatment of specific interpretations of symbols, than would the usual axiomatization and normal resolution. The knowledge about the particular theory is essentially encoded in the algorithm for finding the resolution literals and the substitution σ, which, however, has to be developed for each theory anew. Since this algorithm is a

generalization of the usual unification algorithm, we shall again call it a unification algorithm, although in general nothing will be unified.

The unification algorithm required for an implementation of theory resolution may, for some theories, be too expensive or not even known. This holds in particular, when the theory actually consists of several subtheories that are not independent of each other.

As an example consider the theory $\mathcal{T}_1$ whose models associate with the predicate symbol $\leq$ a reflexive and transitive relation on the universe and with the predicate symbol $\equiv$ the largest equivalence relation contained in the former relation. Each of these interpretations satisfies an atom $s \equiv t$ for two terms s and t, if and only if it satisfies both $s \leq t$ and $t \leq s$. Another theory $\mathcal{T}_2$ may be such that its models associate with the predicate symbol $\equiv$ the equality relation. In the combination of these two theories, the conjunction of the literals $a \leq b$, $b \leq a$, $P(a)$, $\neg P(b)$ is unsatisfiable, so that these are candidates for resolution literals in a theory resolution step.

However, an appropriate theory unification algorithm would have to be designed for just this combination of theories. As soon as a third theory was added, the algorithm could no longer be used. Therefore it would be more convenient to develop algorithms for the individual theories only and to have available a general mechanism that takes care of the interaction between theories.

Let's look at the combination of the theories $\mathcal{T}_1$ and $\mathcal{T}_2$ above and at the clauses:

clause 1:	$a \leq b$, K	clause 3:	$P(a)$, M
clause 2:	$b \leq a$, L	clause 4:	$\neg P(b)$, N

from which we ought to be able to derive the resolvent $\{K, L, M, N\}$. We can obtain this clause through a generalized $\mathcal{T}_1$-step followed by a $\mathcal{T}_2$-step. If an interpretation of the theory $\mathcal{T}_1$ satisfies $a \leq b$ as well as $b \leq a$, then by construction it satisfies the literal $a \equiv b$ as well. It is easy to verify that the clause $C = \{a \equiv b, K, L\}$ is a $\mathcal{T}_1$-consequence of clause 1 and clause 2. The literals $a \equiv b$, $P(a)$, $\neg P(b)$ can now be recognized by the algorithm for $\mathcal{T}_2$ as resolution literals for an "equality theory resolution step" involving the intermediate clause C and clause 3 and clause 4, which results in the desired clause $\{K, L, M, N\}$.

The first step, producing the intermediate clause C, goes beyond theory resolution as presented so far, because the conjunction of the resolution literals is not $\mathcal{T}_1$-unsatisfiable and moreover a new literal was added to the resolvent. This so-called *residue* is characterized by the property that in the theory under consideration it follows from the resolution literals. If a residue is included, one speaks of *partial theory resolution*, otherwise of *total theory resolution*.

As a residue we may also admit a disjunction of several literals. The empty residue then stands for FALSE, and hence follows from the resolution literals only if their conjunction is unsatisfiable in the current theory. This special case corresponds to total theory resolution.

For the most general case, (partial) theory resolution is described by the following schema:
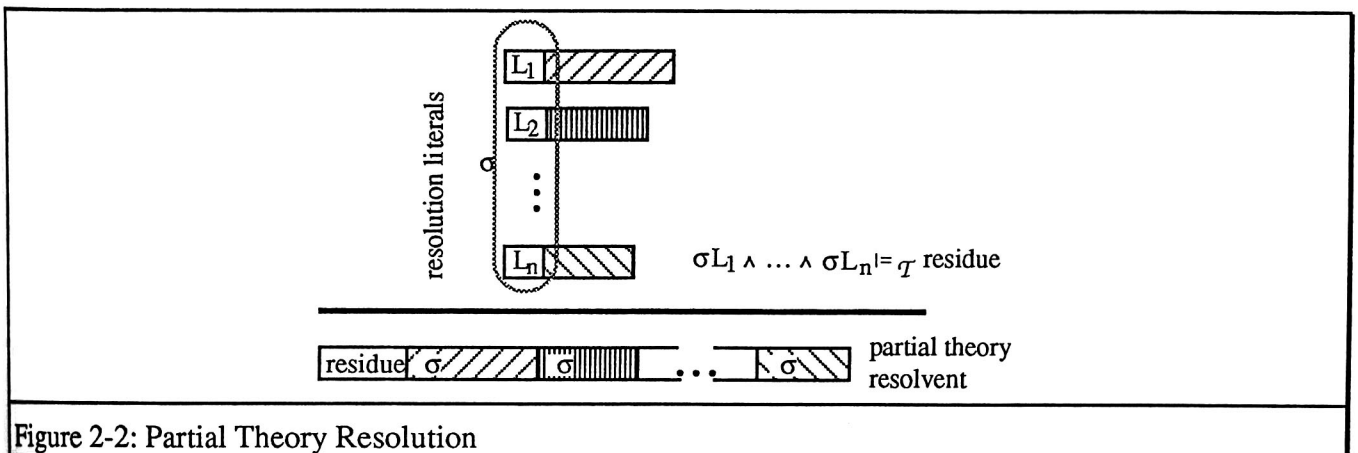

Figure 2-2: Partial Theory Resolution

Furthermore Stickel distinguished *narrow theory resolution* where only one resolution literal per clause participates and *wide theory resolution* with more than one resolution literal per clause participating. As in standard resolution, wide theory resolution can be avoided provided there is a factorization rule.

One of the most important theory resolution rules is paramodulation [Robinson & Wos 69]. It incorporates the meaning of the equality symbol into the calculus by replacing equals by equals.
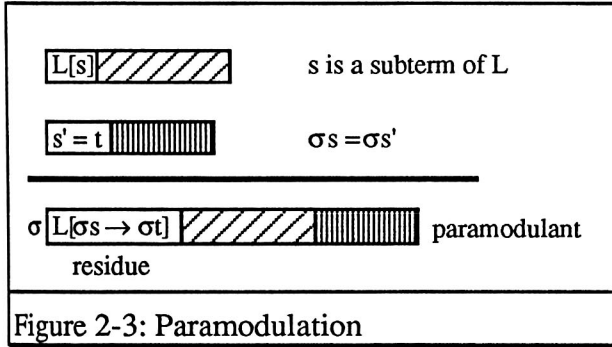
Whenever a subterm s of a literal is unifiable with one side of an equation, a *paramodulant* may be derived where s is replaced by the corresponding instance of the other side of the equation.

L[s]///////	s is a subterm of L
s' = t ‖‖‖‖‖‖	$\sigma s = \sigma s'$

σ L[σs → σt] ///////‖‖‖‖‖‖‖ paramodulant
residue

Figure 2-3: Paramodulation

Special versions of paramodulation are rewriting and demodulation. *Rewriting* is a restricted version of paramodulation where the paramodulating clause is a unit clause (consisting of one equality literal only) and the unifier is a matcher that instantiates the equation only. *Demodulation* is a "destructive rewrite" i.e. the original, *demodulated*, clause is thrown away [Wos et al 67]. It is often used to simplify terms, for example replace $P(+(3, 4))$ by $P(7)$, as it seems to be of no advantage to keep both versions in parallel.

A simple example that illustrates theorem proving by resolution and paramodulation and which was proven by the MKRP-System without a single unnecessary step is the following:

Theorem : The identity element of a subgroup equals the identity element of the group itself.

Axioms necessary for proving this theorem:
$\forall$ x,y subset (x y) $\Rightarrow$ ($\forall$ z member (z, x) $\Rightarrow$ member (z, y))
$\forall$ x,y group (x) $\wedge$ member (y x) $\Rightarrow f(y^{-1}, y) = id(x)$
$\forall$ x group (x) $\Rightarrow$ member (id(x), x)

Theorem:
$\forall$ x,y group (x) $\wedge$ group(y) $\wedge$ subset (x y) $\Rightarrow id(x) \neq id(y)$

Clause form of the axioms and the negated theorem:
A1: ¬subset (x y) ¬member (z, x) member (z, y)
A2: ¬group (x) ¬member (y, x) $f(y^{-1}, y) = id(x)$
A3: ¬group (x) member (id(x), x)

T1: group (a)	T3: subset (a b)	(a and b are Skolem constants)
T2: group (b)	T4: $id(a) \neq id(b)$	

Proof :

A1,T3	⊢ R1:	¬member (z, a) member(z, b)	(resolution: x↦a, y↦b)
A3,T1	⊢ R2:	member (id(a), a)	(resolution: x↦id(a))
R1,R2	⊢ R3:	member (id(a), b)	(resolution: z↦id(a))
A2,T1	⊢ R4:	¬member (y, a) $f(y^{-1}, y) = id(a)$	(resolution: x↦a)
R2,R4	⊢ R5:	$f(id(a)^{-1}, id(a)) = id(a)$	(resolution: y↦id(a))
A2,T2	⊢ R6:	¬member (y, b) $f(y^{-1}, y) = id(b)$	(resolution: x↦b)
R3,R6	⊢ R7:	$f(id(a)^{-1}, id(a)) = id(b)$	(resolution: y↦id(a))
R5,R7	⊢ P1:	$id(a) = id(b)$	(paramodulation: ε)
P1,T4	⊢ R8:	□	(resolution: ε)

2.3 Theory Unification

The paramodulation rule itself cannot produce a contradiction. Its purpose is to manipulate literals by subterm replacement until they become resolvable. (Even in pure equational problems the negated theorem is an inequation and the last step must be a resolution with the reflexivity clause.) With respect to the goal to produce resolvable literals, paramodulation alone is completely blind. In the clause set $\{\{P(a)\}, \{\neg P(b)\}, \{a = b\}, \{a = c\}\}$, for example, nothing prevents the paramodulation rule from producing the irrelevant clause $\{P(c)\}$ instead of $\{P(b)\}$ which is necessary for the refutation.

There is no general solution to this problem, because the general equality problem, that means the problem to prove two terms equal, is undecidable and therefore as hard as theorem proving itself (see chapter 4). Restricted to special equations, however, there may be efficient algorithms for unifying two terms, i.e. for finding the substitutions that make them equal under the given equations.

A *theory unifier* for two terms s and t and an equationally defined theory $\mathcal{T}$ is a substitution σ making the two terms equal under the theory $\mathcal{T}$: $\sigma s =_{\mathcal{T}} \sigma t$, i.e. $\sigma s = \sigma t$ holds in the theory $\mathcal{T}$.

Examples: Let $\mathcal{T}$ be the simple theory of commutativity for f: $f(x,y) = f(y,x)$. The substitution $\{x \mapsto a\}$ is a $\mathcal{T}$-unifier for the terms $f(b,x)$ and $f(a,b)$, which are not directly unifiable. For the terms $f(x,y)$ and $f(a,b)$ there are two T-unifiers, namely $\{x \mapsto a, y \mapsto b\}$ and $\{y \mapsto a, x \mapsto b\}$. Both of them are most general.

There are theories with at most one (called *unitary*), finitely many (called *finitary*), infinitely many (called *infinitary*) and non existing (called *nullary*) most general unifiers. An example for a theory with infinitely many most general unifiers is the theory of associativity: $g(g(x,y)z) = g(x,g(y,z))$). The terms $g(a,x)$ and $g(x,a)$ for example have the following sequence of "associative" unifiers: $\{x \mapsto a\}$, $\{x \mapsto g(a,a)\}$, $\{x \mapsto g(g(a,a),a)\}$, ..., $\{x \mapsto a^n\}$, .. None of them is an instance of the other.

Suppose now that we have a $\mathcal{T}$-unification algorithm for a given theory $\mathcal{T}$ to be used by the resolution rules in place of Robinson's original unification algorithm, then the axioms in $\mathcal{T}$ may be removed from the database 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).

Under these conditions a theorem prover based on theory resolution realized by theory unification algorithms is complete [Plotkin 72]. Based on this result many unification algorithms have been developed (see [Siekmann 89] for a survey). But two main problems remain, when building these algorithms into an automated theorem prover.

The first one is that these algorithms are usually designed for a pure theory only, i.e. they accept as input only terms built from function symbols occurring in the theory axioms, variables and sometimes free constant symbols. Terms like $f(g(a), h(e,y))$ for example are not admissible as input to a pure commutative unification algorithm for the function symbol f. To solve this problem a combination algorithm is necessary which prepares "mixed terms", i.e. terms containing subterms from different theories for submitting them to the corresponding pure unification algorithm and processing their results. The basic idea is to replace "alien" subterms by new variables, submitting the "purified" terms to the special unification algorithm and merging their results with the bindings of the new variables.

As an example, assume the terms $s = g(f(a,y),g(x,y))$ and $t = g(g(c,a),y)$ are to be unified where f is idempotent and g commutative. The first step is to replace the alien subterms of s and t by new variables. We get the bindings $\theta = \{v_1 \mapsto f(a,y), v_2 \mapsto g(x,y), v_3 \mapsto g(c,a)\}$. Unification of the purified terms $g(v_1,v_2)$ and $g(v_3,y)$ with a commutative unification algorithm yields the two unifiers $\sigma_1 = \{v_1 \mapsto v_3, v_2 \mapsto y\}$ and $\sigma_2 = \{v_2 \mapsto v_3, v_1 \mapsto y\}$. Merging of θ with σ_1 is not possible because $f(a,y)$ and $g(c,a)$ are not unifiable. Merging of θ with σ_2 yields the unification problem $y = f(a,y)$ that is solved by the algorithm for

idempotency. The unifier, $\{y \mapsto a\}$ is applied to the binding of v_2 such that together with the the solution $\{x \mapsto c\}$ of the remaining problem $g(x,a) = g(c,a)$ we obtain the final answer $\{y \mapsto a, x \mapsto c\}$.

Different combination algorithms have been developed so far, [Stickel 81,87, Herold 87, Yellick 87, Kirchner 85, 87] there are more efficient ones for restricted classes of theories and less efficient ones for more general classes. A general, although still quite inefficient algorithm for the combination of arbitrary theories has been developed by Manfred Schmidt-Schauss [Schmidt-Schauss 88]. Recently Michael Tepp has improved its efficiency considerably by introducing a more dynamic control structure and by invoking more efficient special combination algorithms for special classes of theories, i.e. his algorithm is a combination algorithm for the combination algorithms [Tepp 89].

The second main problem for the incorporation of theory unification algorithms into a theorem prover is the combination with general purpose equality reasoning. If there is a unification algorithm for, say, associativity of a function f, this is usually not the only equation for f occurring in the axioms. For the equations which are not covered by the unification algorithms, rules like paramodulation have to be used. The following example demonstrates the problem to combine theory unification algorithms with paramodulation. Suppose we have the unsatisfiable axioms:

$$P(f(a, f(b,c)))$$
$$f(a,b) = f(a,d)$$
$$\neg P(f(a, f(d,c)))$$

where f is associative and the associativity axiom is replaced by an $\mathcal{A}$-unification algorithm. This algorithm cannot unify $P(f(a, f(b,c)))$ and $P(f(a, f(d,c)))$ because b and d clash. Since both literals contain neither $f(a,b)$ nor $f(a,d)$ as subterms, paramodulation is also not applicable. The problem is that there is no rule for scanning the equivalence class of $f(a,f(b,c))$ or $f(a, f(d,c)))$ for terms with appropriate subterms. (see [Peterson &Stickel 81] for this problem in the context of term rewriting systems.)

2.4 Many Sorted Logics

In the two previous paragraphs we discussed ways to incorporate the equality predicate and special equations into the calculus. In this section we are going to present our means to build fragments of set theory and taxonomical hierarchies into the inference machinery. The fragments we can handle are characterized by the fact that there are at most finitely many or at least enumerably many sets involved. All sets and their subset relationships have to be known from the beginning. Typical examples are integer, real, mammals, animals etc. In this context these sets are usually called *sorts* in the literature and sorted logics are logics which represent sorts by a special mechanism.

The usual axiomatization of sorts in PL1 utilizes unary predicates. For example, the set of all integers is represented by a predicate 'integer' such that integer(t) expresses that the term t denotes an integer. The explicit representation of sets as unary predicates has many disadvantages. The subset relationship has to be represented by axioms like $\forall x$ integer(x) $\Rightarrow$ real(x) or $\forall x$ ape(x) $\Rightarrow$ mammal(x) and $\forall x$ mammal(x) $\Rightarrow$ animal(x) and in realistic applications these taxonomical hierarchies tend to be large. Reasoning about the sort of an object requires the execution of chains of resolution steps with the sort axioms. For instance given the clause {ape(cheeta)}, if one needs {animal(cheeta)} one has to perform two resolution steps with the above axioms. Furthermore resolution steps are possible which do not make sense in the intended domain of discourse. Suppose for example there are the two clauses {¬ape(x), eats(x, icecream)} and {¬dog(y),¬eats(y, icecream)}. A resolvent is {¬ape(x), ¬dog(x)}. In order to use this clause for a refutation proof an object 'a' which satisfies ape(a) and dog(a) has to be found. If the axioms in fact reflect our intuition about apes and dogs the resolvent {¬ape(x), ¬dog(x)} expressing that all apes are no dogs, is a useless tautology.

The standard method for preventing such things is to attach the sort information directly to the variables, constants and terms, and to allow instantiation only with terms whose sorts fit the sort of the variable. Instead of $\forall x$ ape(x) $\Rightarrow$ eats(x, icecream) one would write $\forall x$:ape eats(x, icecream) where the variable x is tagged with the sort 'ape' and instantiation of x is only allowed with terms of sort 'ape' or smaller.

Many sorted logics represent sorts and subsort relationships in a special datastructure, the *sort structure*. In the simplest version, the sort structure is a flat list of sorts representing mutually disjoint sets. More expressive are sort hierarchies in which case we speak of *order sorted logics*. They allow to model subset relationships. Sort hierarchies are the basic datastructures of the sorted logics Christoph Walther [Walther 87] and Manfred Schmidt-Schauss [Schmidt-Schauss 88] have developed for the MKRP system.

Making the sort structure more expressive is one dimension along which sorted logics can be developed. Anthony Cohn has driven this to one extreme by allowing complete sort lattices which model subset relationships, intersection, union and complement [Cohn 87]. Another extreme is parametric polymorphism which provides constructor functions for sorts. 'List-of' for example is a typical sort constructor function that allows to define sorts List-of(integer), List-of(List-of(integer)) etc. [Milner 77], [Smolka 89].

The other dimension along which sorted logics can be developed is the modelling of the membership predicate. The sort of a term is the syntactic notion for the semantic membership relation. The finer the sort of a term can be determined, the better the membership relation is modelled. In Walther's ΣRP-calculus the sort of a term depends only on the leading function symbol. If you declare 'real' to be the rangesort of a function symbol + for example then the term +(3,4)[1] is of sort 'real' although its interpretation is always an integer.

---

[1] 3 and 4 are taken to be constant symbols of sort 'integer'.

In Schmidt-Schauss´ ΣRP* calculus more detailed information about the domain-rangesort relation of function symbols can be exploited. In the simpler version the sorts of a term depend on its leading function symbol and the sorts of the direct subterms. For example the declarations +:real $\times$ real $\rightarrow$ real, integer $\times$ integer $\rightarrow$ integer are sufficient to compute 'integer' as the sort of the term +(3,4). The more complicated version makes the sort of a term depending of the structure of its subterms. Declarations *:real $\times$ real $\rightarrow$ real, *(x, x^{-1}): integer, yield 'integer' as the sort of the term *(a, a^{-1}), regardless whether the sort of 'a' is real or integer.

Both Walther´s and Schmidt-Schauss calculi include resolution and equational reasoning with paramodulation. For both rules only minimal changes of the original calculus were necessary. First of all unification of terms is restricted such that instantiation of a variable of sort S is only possible with a term whose sort is smaller[1] or equal to S. (Notice that the 'smaller or equal' check is the place where the transitivity of the subset relation is built into the calculus.) For sort structures which are not tree like, i.e. where two sorts S_1 and S_2 may have a common subsort S_3, the unification of two variables has to be refined by a *weakening rule*: In order to unify two variables x of sort S_1 and y of sort S_2, they have to be weakened to the common subsort S_3. The unifier is $\{x \mapsto z, y \mapsto z\}$ where z is of sort S_3. If all pairs of sorts have at most one greatest common subsort, i.e. the sort structure is a semilattice, there is at most one weakening possibility and therefore the unification is still unitary whereas in the general case the unification is of type finitary.

In the ΣRP* calculus the weakening rule has to be extended to terms. Since the sort of a term may depend on the sorts of its subterms, weakening of a subterm may weaken the sort of the term itself such that a *well sorted* assignment to a variable may become possible. For example the unifier for x:integer and +(y:integer, z:real) is $\{x \mapsto +(y, z'), z \mapsto z'\}$ where z' is a new variable of sort integer.

As we have seen appropriate changes in the unification algorithm ensure that instantiation does not destroy well sortedness of terms. A corresponding restriction to the paramodulation rule has to ensure that subterm replacement does not produce ill sorted terms. It is for example not allowed to paramodulation into P(f(a)) with a:integer = b:real when f accepts only 'integer' as domainsort.

So far the technical modifications necessary for a resolution theorem prover to incorporate sorts are easy to implement. Unfortunately the fragment of set theory, these simple sorted logics can handle is not very significant. Minimal extensions, however, already have serious consequences for the calculus. As a next step the incorporation of complement information into the sort structure is only possible with partial theory resolution where the residues reintroduce the sorts as unary predicates. To demonstrate this, assume we have the sort structure $\underset{\text{odd} \quad \text{even}}{\overset{\text{integer}}{\diagup \diagdown}}$ and a constant 'a' of sort integer. The terms x:odd and a:integer are not unifiable because the sort of 'a' is larger than the sort of 'x'. If, however, we exploit the fact that the set of odd numbers is the complement of the set of even numbers then every term of sort 'integer' is interpreted either as an odd or as an even number. Thus it cannot be excluded that the "real sort" of 'a' is 'odd' and 'x' and 'a' are in fact unifiable. In this case the unifier $\{x \mapsto a\}$ must contain the condition "provided a is of sort odd" and a corresponding resolvent between P(x) and $\neg$P(a) would not be the empty clause, but the negated condition, $\neg$odd(a). To continue the example, a similar resolution might produce a clause $\neg$even(a) and now disjointness of 'odd' and 'even' produces the contradiction.

Within the framework of theory resolution this extension to sorted logic is quite feasible and we are currently exploring its possibilities. More complicated however is the combination with equality reasoning and theory unification.

Although the fragment of set theory sorted logics encode is still quite small, many examples have already shown its practical advantages in reducing the search space considerably [Walther 87, Ohlbach &Schmidt-Schauss 85] and it is a common place now to assert that without it no realistic applications of a deduction system are possible.

---

[1] smaller in the partial ordering of the sort hierarchy. S_1 smaller than S_2 means semantically $S_1 \subseteq S_2$.

3 CLAUSE GRAPHS

We now turn to the representational format for the resolution calculus as used in the MKRP system. It was originally developed by Robert Kowalski [Kowalski 75] whose connection graph proof procedure was defined for resolution in Robinson's original sense. After a brief introduction of Kowalski's original version (which is essentially taken from [Eisinger 88]) we shall present it in the more general form we have developed for theory resolution [Eisinger et al 89], [Ohlbach & Siekmann 88]. Many of the reduction operations we present in chapter 3.3 have been developed by Ch.Walther [Walther 81], N.Eisinger [Eisinger 81] and A. Präcklein [Präcklein 85].

3.1 Kowalski's Connection Graph Procedure

The original idea of a clause graph is to graphically represent some binary relation on the literals of a clause set by drawing links between the pairs of literal occurrences for which the relation holds. A clause graph can be regarded as an ordinary graph whose nodes are labelled with literals, but in addition the set of nodes is partitioned into contiguous clusters corresponding to the clauses. They are called clause nodes.

The reason for the distinction between literals and literal nodes and between clauses and clause nodes is purely technical. Different nodes may very well be labelled with the same literal, but be linked to entirely different places. If the literals themselves were regarded as the nodes of the graph, one could not even formulate a phenomenon like this. However, in the sequel we will not strictly distinguish between nodes and formulas, as long as there is no confusion possible.
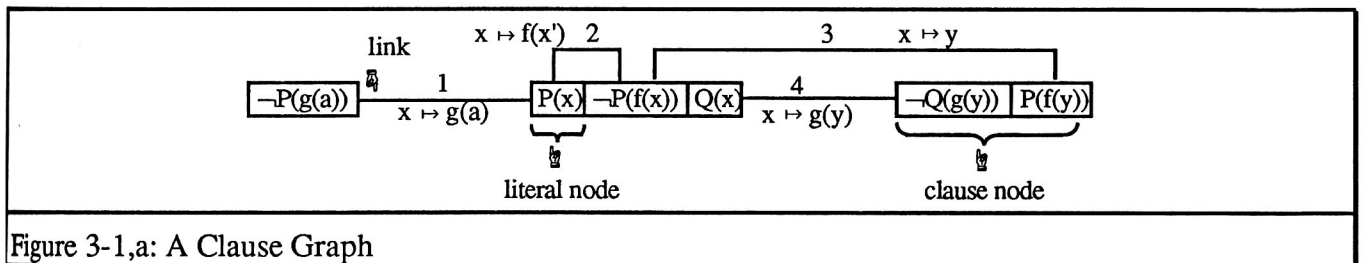

Figure 3-1,a: A Clause Graph

In the example of figure 3-1 the relation represented by the links is resolvability, i.e. any two literals with opposite sign and unifiable atoms are connected by a link. With each link one can associate a most general unifier for the incident literals. In this example the links, except link 2, represent all resolution operations possible among the given clauses. Link 2 represents a self resolution possibility, i.e. a resolution possibility between two renamed copies of the clause {P(x), ¬P(f(x)), Q(x)}. The corresponding resolvent would be {P(x), ¬P(f(f(x))), Q(x), Q(f(x))}.

Turning resolution into a rule operating on clause graphs rather than clause sets is fairly straightforward. As an illustration assume we want to perform the resolution step represented by link 4 in the above example. By applying the most general unifier {x ↦ g(y)} and renaming the remaining variables, we obtain the resolvent {P(g(y')), ¬P(f(g(y'))), P(f(y'))} where the literal P(g(y')) descends from P(x), the literal ¬P(f(g(y'))) descends from ¬P(f(x)) and P(f(y')) descends from P(f(y)).

Along with new nodes corresponding to the resolvent we have to add to the graph new links representing the underlying relations's extension to the enlarged set. This can be achieved by checking each new literal with each of the old ones for resolvability. The resulting graph is as follows:
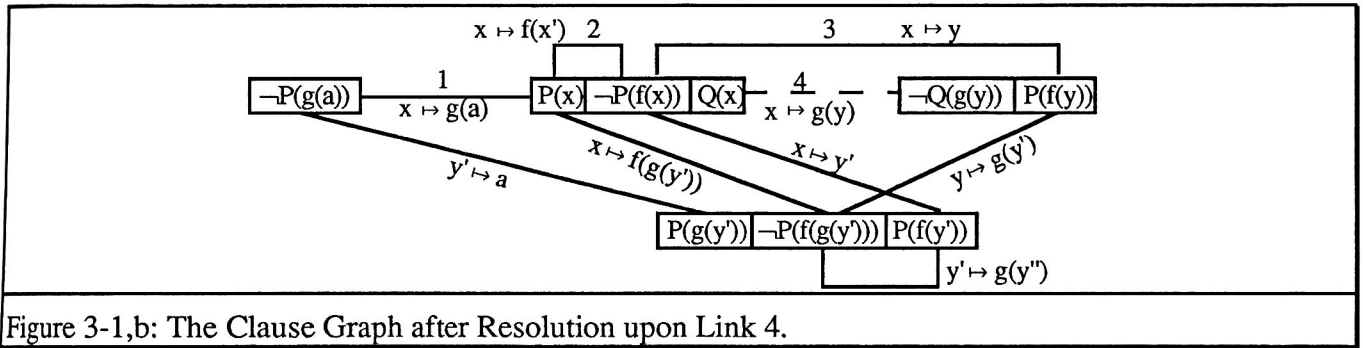
Figure 3-1,b: The Clause Graph after Resolution upon Link 4.

Now, any node connected to a new node is also connected to the node´s ancestor (but not conversely). This is generally the case: Since each descendant literal is a renamed instance of its ancestor literal, a descendant is resolvable with a third literal only if the ancestor is. Thus the links incident with a new node can be obtained without search by examining the links incident with its ancestor. Technically this process is usually conceived as an inheritance of the links from the ancestors to the descendants (in section 3.4.3.1 we shall present the inheritance algorithm.)

Next the inference system can be modified by requiring the removal of each link upon completion of the respective resolution. At first sight this might seem like a trivial book-keeping affair that amounts to simply marking the links after operation in order to avoid repetition of the same inference step. But there are deeper consequences to this link removal, because once a link has been removed, it can no longer be inherited by literals derived in subsequent operations. Actually there are examples where the non-inheritance of removed links avoids the generation of the same clause in exponentially many different ways.

Finally the basic connection graph procedure can be modified once more by incorporating an extension of the purity principle. In [Robinson 65] a literal is defined to be pure, if it does not resolve with any other literal in the clause set. A clause containing a pure literal may be removed without affecting the clause set´s unsatisfiability. This removal can result in further purities if all resolution partners of some other literals belong to the removed clause, thus the purity principle potentially involves a chain reaction of removals. An obvious way to transfer the purity principle to clause graphs is to define a pure literal to be one not incident with any link. Were it not for the link removal after each operation, the two concepts of purity would coincide. With link removal, however, the effect of the purity principle is drastically enhanced.

After removing the resolved link in the example of figure 3-1,b, deleting the two pure parent clauses of the resolution and following the snowball effect of purity deletions we end up with the empty graph which indicates satisfiability of the original clause set. Thus after only one resolution the satisfiability of the clause set has been detected, whereas with clause set resolution infinitely many resolution operations are possible.

3.2 General Clause Graphs as a Datastructure and Indexing Mechanism

The version of clause graphs we have developed for theory resolution requires more complex links connecting arbitrary many literals and more complex clause nodes representing arbitrary many renamed copies of clauses. Considering copies of clauses with renamed variables and constructing links between these copies has only efficiency reasons, but is in principle not necessary for representing ordinary resolution operations. Clause copies, however, can no longer be neglected in the extension of the clause graph approach to theory resolution, because a theory resolution operation may involve arbitrarily many copies of the same clause at once. The second clause, a conditioned equation, in the example

$$P(g(a))$$

$$g(y) = g(f(y)), Q(y)$$

$$\underline{\neg P(g(f(f(f(a))))))}$$

$$Q(a), Q(f(a)), Q(f(f(a)))$$

has to be used in three different instantiations $\{y \mapsto a\}$, $\{y \mapsto f(a)\}$ and $\{y \mapsto f(f(a))\}$ in order to find out that $P(a)$, $\neg P(g(f(f(f(a)))))$ and $g(y) = g(f(y))$ are contradictory. Therefore we introduced the notion of a *clause node bunch* as a conceptual entity for representing an infinite source of variable disjoint copies of a clause. A link may then connect different clause nodes in a clause node bunch.

Figure 3-2: Clause Node Bunches and Literal Node Bunches
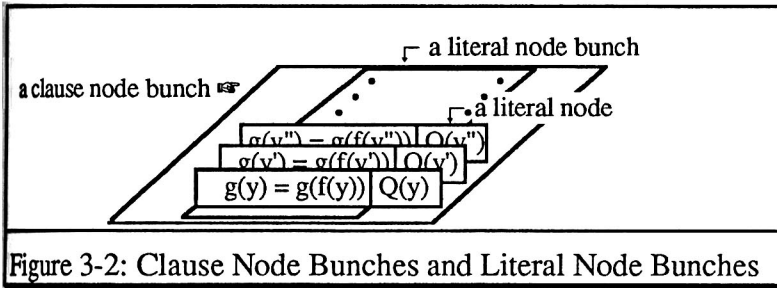
(In an implementation only a finite number of clause nodes in a clause node bunch is usually generated at a time and indicated by the corresponding variable renaming substitution.)

3.3 I-links

Links in the most general version of clause graphs represent partial narrow theory resolution operations. Therefore the so called "I-links" (Implication links) usually connect two groups of literal nodes, the "antecedent" and the "succedent", the first ones serving as resolution literals and the second ones serving as residue:

Figure 3-3: A General I-link
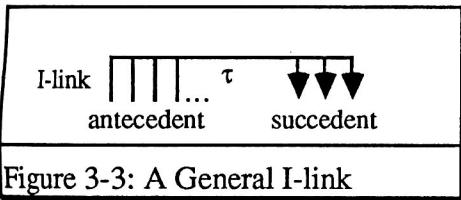
τ is a set of "link substitutions". The notion "unifier" is no lon adequate because in general nothing will be unified.

A general I-link corresponds to a partial theory resolution step in which the instance $\tau K_1 \vee \ldots \vee \tau K_m$ of the succedent is the residue. The graphical notation of an I-link is supposed to reflect the semantics of the antecedent and succedent: The conjunction of the τ-instances of the antecedent literals implies the disjunction of the corresponding instances of the succedent literals. This interpretation of the link type requires the antecedent literals (joined conjunctively) to be parts of different clauses, and the succedent literals (joined disjunctively) to be parts of the same clause. Several antecedent literals within the same clause are taken to mean that they belong to different copies of this clause. If succedent literals are scattered over several clauses, the I-link represents no executable operation. However, resolution steps may cause instances of these succedent literals to become part of the same resolvent, so that inheritance creates an executable I-link.
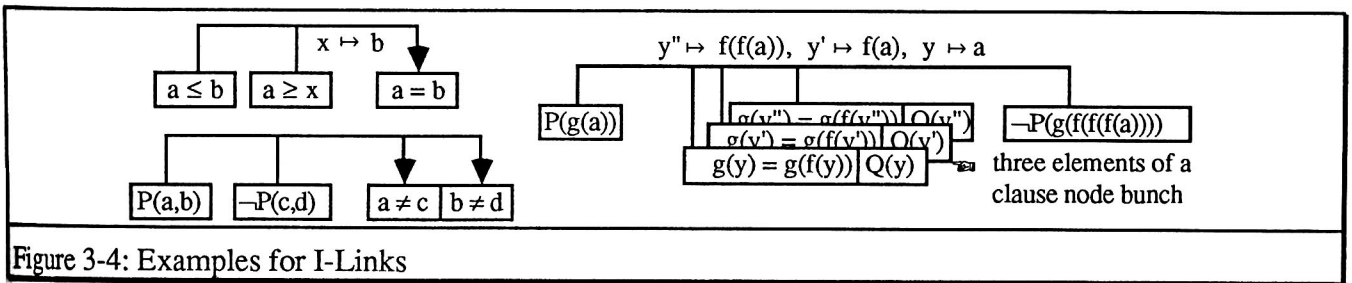

Figure 3-4: Examples for I-Links

3.3.1 R-Links and T-Links

Two special types of I-links, those with an empty succedent and those with an empty antecedent respectively are of particular interest. Links of the first type (Resolution links or R-links) indicate total narrow theory resolution and links of the second type (Tautology links or T-links) indicate actual or potential (after instantiation) tautologies.
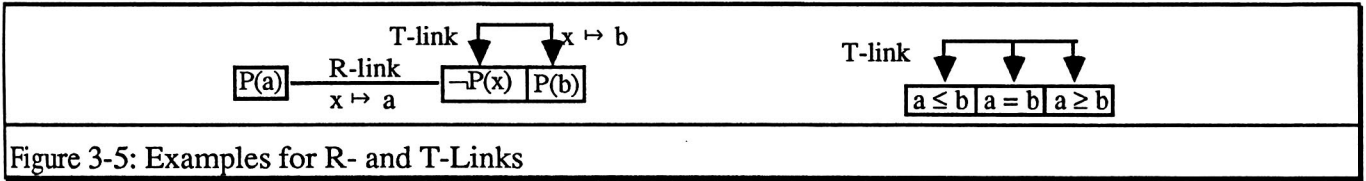
Figure 3-5: Examples for R- and T-Links

3.3.2 F-Links

A third special type of I-links is also of particular interest, namely links connected to one clause node only and with one antecedent literal node and at least one succedent literal node. Links with this structure represent a generalization of the factoring operation: Since the antecedent literal implies the succedent literal or the corresponding instances respectively and they are disjunctively connected, one can create an instance of the clause where the antecedent literal is removed.
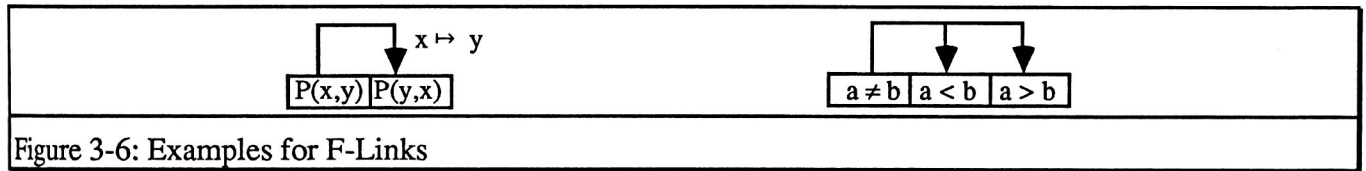
Figure 3-6: Examples for F-Links

3.3.3 P-Links

Since paramodulation can be seen as a special case of partial theory resolution, where the paramodulated literal is the residue, paramodulation operations can also be represented by I-links.

The paramodulated literal is not part of the original graph. It has to be created either when the link is constructed or when the paramodulation operation is executed. Therefore a more compact representation avoids the explicit representation of the paramodulated literal as a residue by not attaching the link to the literal but to the term to be paramodulated and to one side of the equation. With this information the paramodulated literal can be reconstructed when necessary. In the paramodulated connection graph procedure [Siekmann & Wrightson 79] we called these links P-links.
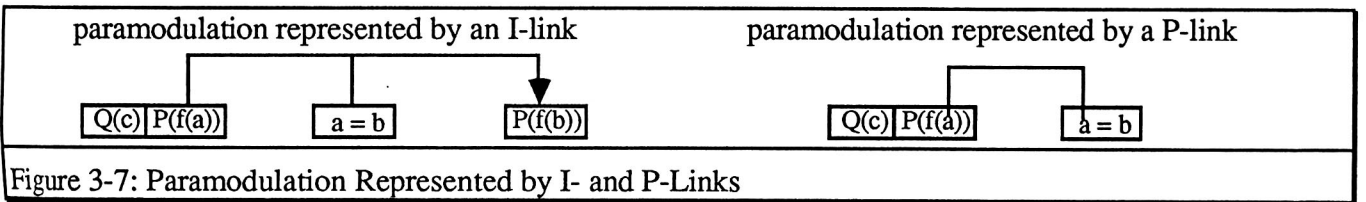
Figure 3-7: Paramodulation Represented by I- and P-Links

We should however keep in mind that P-links are only a compact representation of a special kind of I-links.

3.3.4 Link Bunches

An obvious extension to the link concept that is analog to the concept of clause node bunches is the concept of "link bunches". Just as a clause node bunch is an infinite source for variable disjoint copies of a clause, a link bunch is an infinite source for copies of a link which are connected to the corresponding copies of a clause in a clause node bunch.

Figure 3-8: Examples for Link Bunches

In this chapter no further details of clause node bunches and link bunches need to be explained, because the main principles of the operations on clause graphs can be described without distinguishing between clause (literal) nodes and clause (literal) node bunches. Clause node bunches and link bunches come into play when dealing with non-trivial cases like self resolution, recursion etc., that make life hard and first order logic undecidable.
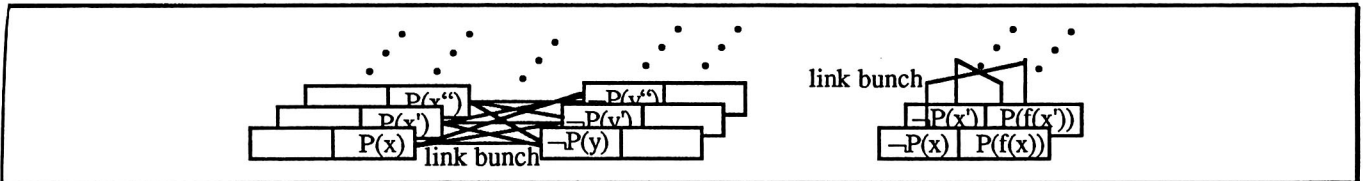
3.3.5 Link Substitutions and Clause Substitutions

To improve conceptual clarity we consider the link substitutions as a - possibly infinite - set of ground substitutions. For example the substitution $\{x \mapsto y\}$ is taken to be a representation of the set of ground substitutions $\{\{x \mapsto a, y \mapsto a\}, \{x \mapsto b, y \mapsto b\}, \{x \mapsto f(a), y \mapsto f(a)\}...\}$. This view has the advantage that operations on substitutions which are necessary in the algorithms described below can be expressed easily with set operations. The merging of substitutions for example turns out to be the intersection of sets of ground substitutions. (As an example consider the merging of $\sigma = \{x \mapsto y\}$ and $\tau = \{x \mapsto a\}$ yielding $\{x \mapsto a, y \mapsto a\}$. In the ground substitution representation we obtain this substitution by intersecting $\{\{x \mapsto a, y \mapsto a\}, \{x \mapsto b, y \mapsto b\}, \{x \mapsto f(a), y \mapsto f(a)\}...\}$ with $\{\{x \mapsto a, y \mapsto a\}, \{x \mapsto a, y \mapsto b\}, \{x \mapsto a, y \mapsto f(a)\}...\}$)

As a further advantage we can use the set-difference operation on the sets of ground substitutions for representing "all σ-instances except those which are also τ-instances". The τ-instances of a link with link substitution σ might for example represent tautologous resolvents and should therefore be removed.

A clause can be seen as a compact representation of all its ground instances. For example the clause $\{P(x,y), \neg P(y,x)\}$ represents all the clauses $\{P(a,b), \neg P(b,a)\}$, $\{P(a,f(a)), \neg P(f(a),a)\},...$. Some of the ground instances, in this case $\{P(a,a), \neg P(a,a)\}$, $\{P(b,b), \neg P(b,b)\},...$ are tautologies and therefore useless for a refutation proof. In order to represent the fact that not all ground instances of a clause are required, we shall assume in the sequel that each clause is labelled with a set of ground substitutions, the *clause substitutions*. In an actual implementation of course an infinite set of ground substitutions cannot be represented, but we can represent the contrary, a finite set of substitutions whose ground instances indicate those instances of the clause which have to be removed. In order to represent that all $\{x \mapsto y\}$-instances of $\{P(x,y), \neg P(y,x)\}$ are tautologies, we have to label the clause node just with $\{x \mapsto y\}$ as *instantiation limit*: $\boxed{P(x,y) \mid \neg P(y,x)}_{\neg\{x \mapsto y\}}$. A consequence is that link substitutions of adjacent links have to be cleaned of those substitutions which are not present in the clause substitutions. On the conceptual level this can be done by removing sets of substitutions from the set of ground link substitutions. In an actual implementation we would label the links with a second set of substitutions, the instantiation limits, which act as constraints in the unification algorithms.

Figure 3-9: Clauses and Links with "Instantiation Limits".
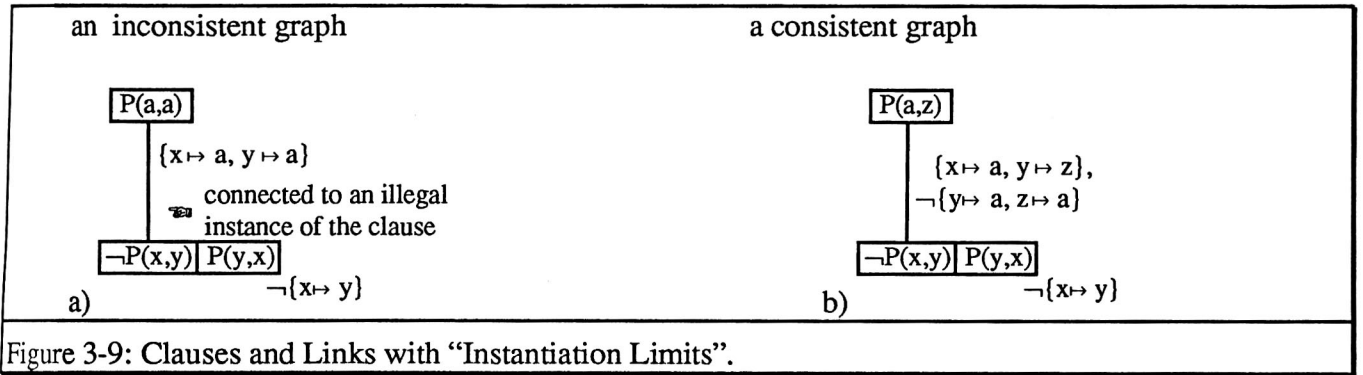
The symmetry clause $\neg P(x,y)\ P(y,x)$ in figure 3-9a) is labelled with the intstantiation limit $x \mapsto y$ stating that the $x \mapsto y$ instances of the clause are tautologies. The literal $P(a,a)$ is connected to these tautologous instances only. This is indicated by the fact that the link substitution $\{x \mapsto a, y \mapsto a\}$ is an instance of the instantiation limit. Therefore the graph is "syntactically inconsistent". The graph in figure 3-9b, however, is consistent because the critical instance $\{y \mapsto a, z \mapsto a\}$ of the link substitution is removed by the link's instantiation limit.

3.4 Clause Graph Inference Rules

The clause graph version of resolution and factoring is in principle sufficient for a complete refutation procedure. Clause graphs, however, support a number of additional operations, which reduce and reorganize the search space such that good and bad steps can be better distinguished. In the sequel only simplified versions of the inference and reduction rules are presented. The exact formulations are technically more complicated, but do not show new principles.

3.4.1   Basic Operations

3.4.1.1   Clause Resolution   (*cres*)

The "traditional" inference rule for clause graphs is clause resolution as proposed by R. Kowalski. It consists of two parts, the generation of the resolvent and a mechanism for "inheriting" the links connecting the new resolvent with the rest of the graph from the links connected to their parent clauses. The inheritance mechanism avoids searching the whole graph for resolvable literals. The formulation of the algorithms for partial narrow theory resolution is technically more complicated, but the ideas are the same. A partial narrow resolution is indicated by an I-link: The antecedent literals are the resolution literals and the succedent literals are the residue. One of the link substitutions is the resolution substitution (unifier).

Clause graph resolution works as follows:

1.   Generation of a partial narrow resolvent from an I-link and one of its link substitutions

-       Form a new clause node by joining the remainders of the antecedent clause nodes without the antecedent literal nodes themselves with the succedent literal nodes and applying the resolution substitution.

-       Make the new clause node variable disjoint with all other clause nodes.
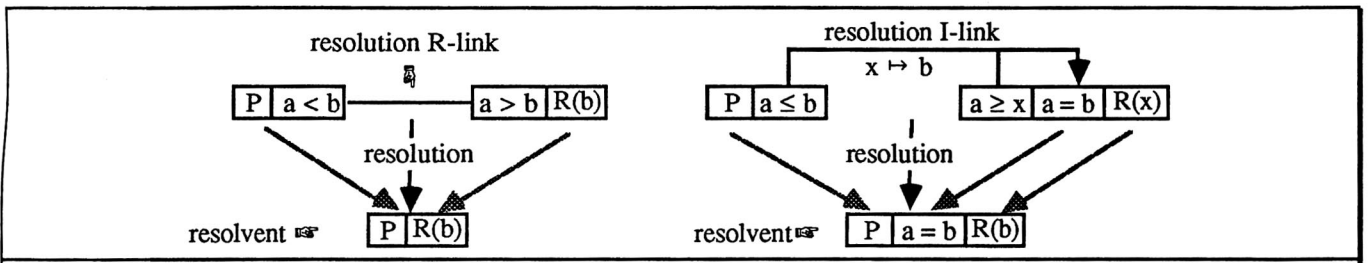
Figure 3-10: Examples for the Generation of Resolvents.

2.   Link Inheritance

The idea for the link inheritance mechanism is very simple: In order to connect a link with a literal node of the resolvent, "grasp" a link connected to its parent literal node and "pull it down" to the resolvent literal node [Smolka 82]. The link substitution for the new link can be computed from the link substitution of the old link and the link substitution of the resolved link by substitution merging [Ohlbach 87].
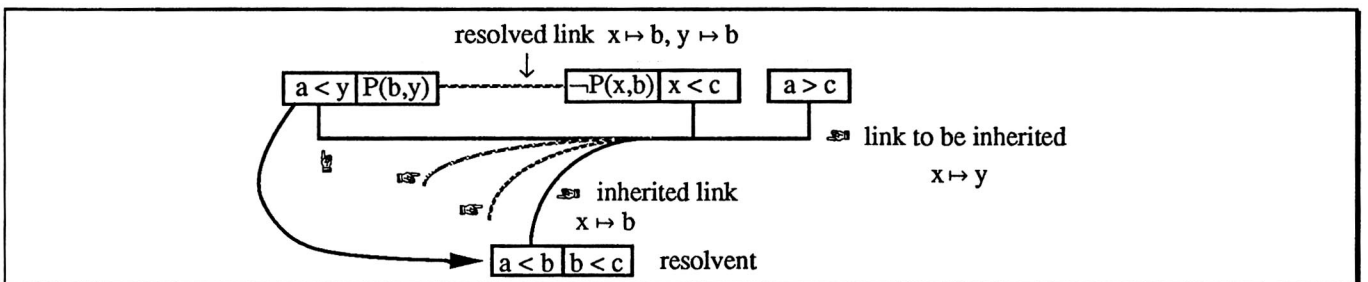
Figure 3-11: Example for Link Inheritance

Of course only a copy of the original link is manipulated this way. The original link itself remains untouched.

The next example demonstrates that this idea is powerful enough to get the internal links of the resolvent, i.e. the links connecting different copies of the same clause, as well as the link connecting the resolvent with its parent clauses. (Remember that the internal links actually represent link bunches, i.e. all corresponding links between copies of the clause.)

Figure 3-12,a:
Example: Link Inheritance after Resolution upon Link 4,
Inheritance to the first Literal of the Resolvent.
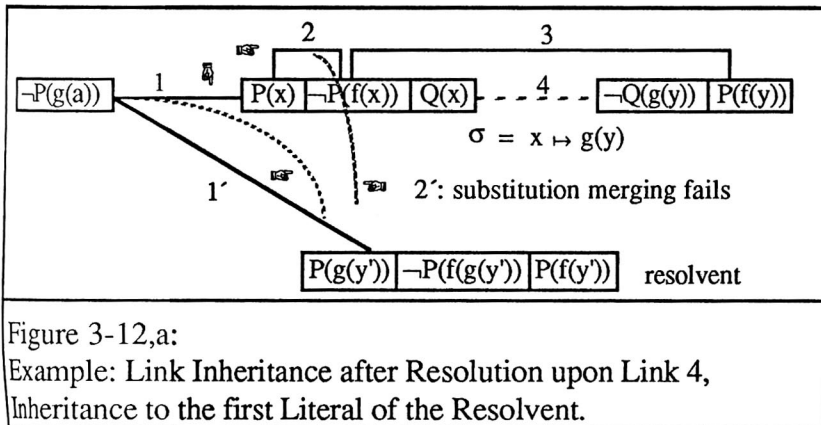
Link 1 is "dragged" from the literal $P(x)$ to its instance $P(g(y'))$. Link 2 cannot be inherited because the resolution substitution $\{x \mapsto g(y)\}$ is incompatible with the link substitution $\{x \mapsto f(x'')\}$.
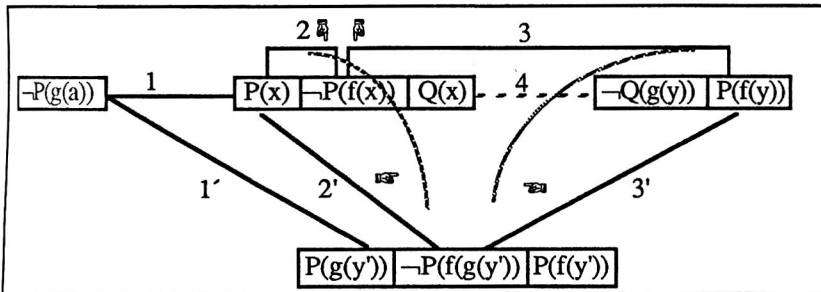
Figure 3-12,b: Inheritance to the Second Literal of the Resolvent.

Link 3 can be inherited normally. For dragging link 2 from the literal ¬P(f(x)) to its instance ¬P(f(g(y'))) we take an element of link 2's link bunch that is connected to P(x") and ¬P(f(x)), i.e. we take the weak unifier {x" ↦ f(x)}. This substitution is compatible with {x ↦ g(y)}
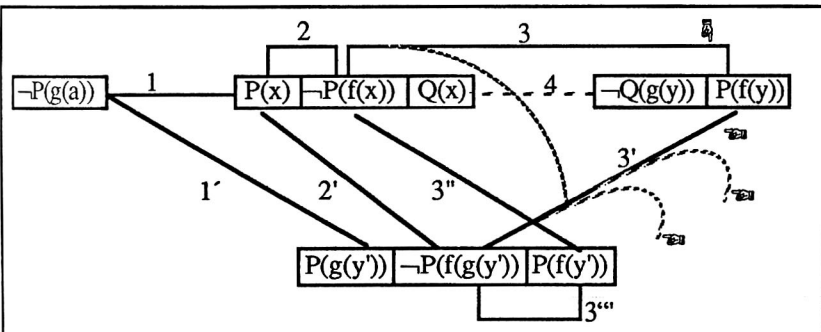
Figure 3-12,c: Inheritance to the third Literal of the Resolvent

Link 3 is inherited normally yielding 3". The new link 3' which is connected to P(f(y)) has also to be inherited yielding the internal link 3"'.

As a final operation the resolved link can be removed in order to inhibit a repetition of the resolution.

The presented link inheritance mechanism works as long as the new literals in the resolvent are instances of already existing literals in the graph. Some of the partial theory resolution rules, in particular paramodulation, which generate new literals as residues need special inheritance mechanisms for connecting the new literals with the rest of the graph. In chapter 4 we shall discuss these problems for paramodulation.

Explicit generation of clause graph resolvents is very expensive, because besides the new clause usually hundreds of new links have to be generated. Since the literals in the resolvent are just instances of literals in the parent clauses and the links to these literals are essentially copies of the original links, a resolvent contains mostly redundant information. However, there may be resolvents which trigger some of the reduction operations explained below and cause the removal of clauses and links.

3.4.1.2 Link Resolution (*lres*)

Link resolution is a new inference rule on our version of clause graphs. Just as clause resolution derives a new clause from a set of clauses, link resolution derives a new link from a set of links and a single clause. The basic idea shall be explained with a few examples. Consider the graph for the symmetry axiom {¬P(x,y), P(y,x)} and the two clauses {¬P(a,b)}, {P(b,a)}.

Example A sequence of link resolutions that proves the unsatisfiability of the clause set { {P,Q}, {¬Q,P}, {¬P,R}, {¬R,¬P} } is shown in the next figure.

Figure 3-16: A Refutation with Link Resolutions
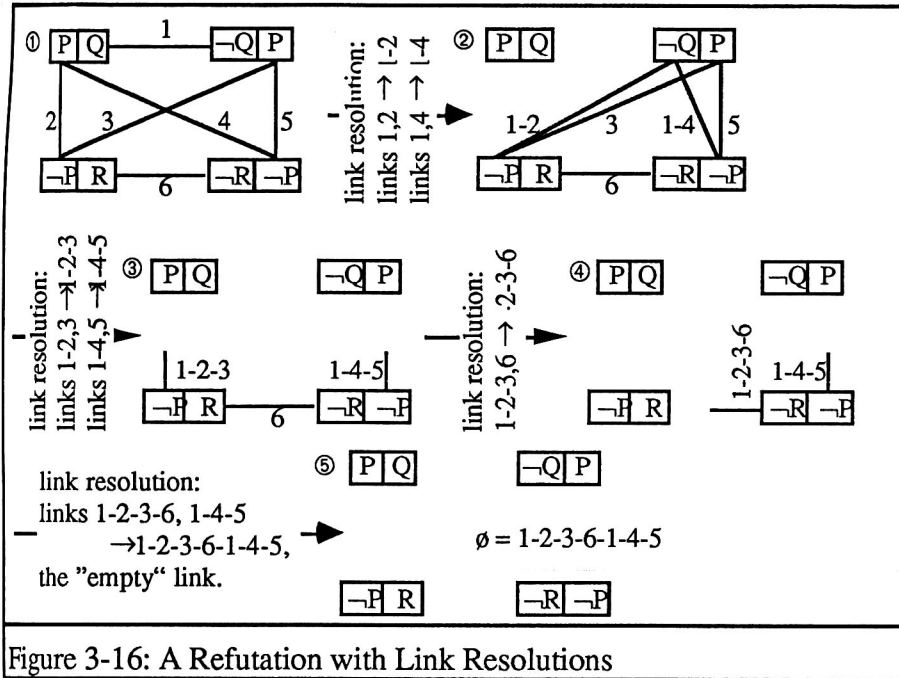
The links 1, 2 and 4 can be removed because all possible combinations are executed in graph ②.

The links 1-2-3 and 1-4-5 contain the information that ¬P is false in the theory generated by the clauses {P, Q} and {¬Q, P}. Note that a resolution of links connected with the same literal, as for example 1-2 with 3, includes an implicit factoring operation.

Finally the "empty link" is generated in graph ⑤. The empty link states that the empty set has no model in the theory generated by the axioms. This is the elementary contradiction which indicates that the axioms are unsatisfiable.

Some more examples show what happens, when different kinds of links are combined.

Figure 3-17: Creation of an F-Link with Link Resolution
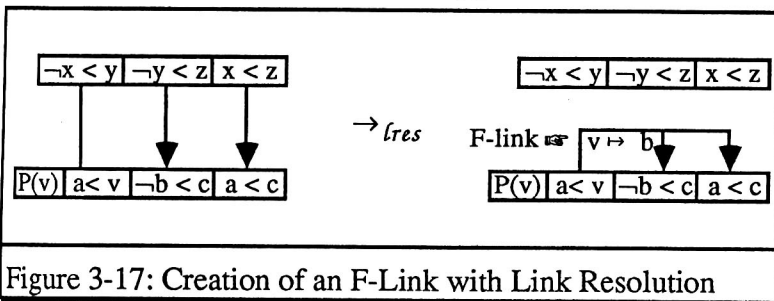
The resolved link is actually an F-link stating that a < b implies ¬b < c or a < c. This information could be used to generate a shorter instance {P(b), ¬b < c, a < c} of the second clause.

Figure 3-18: Creation of a T-Link with Link Resolution
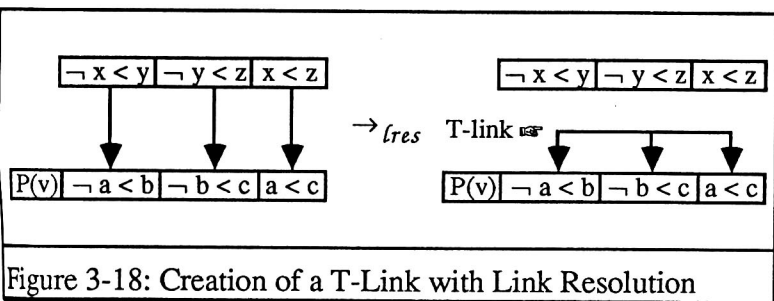
The three links in the left graph indicate that the second clause is subsumed by the transitivity clause. The corresponding combined T-link in the right graph states that the second clause is a tautology in the theory of the first clause. This example shows the very tight correlation between subsumption and tautology.

The two successive link resolutions in the example below generate a link, which represents the information that b<c implies a<c under the theory a<b and the transitivity of <. Using this link and the parallel one, it can be recognized that the clause {b<c, P(c)} subsumes {a<c, P(c)}. A further link resolution would produce a T-link containing the same information.
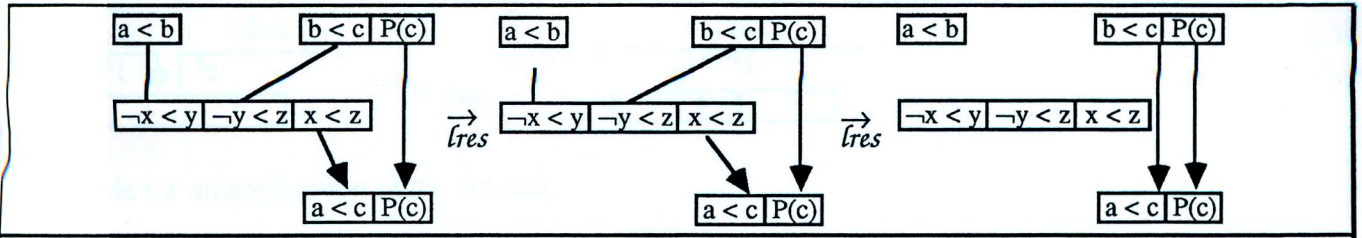
Figure 3-19: Creation of a Subsumption Situation with Link Resolution

(The first operation is a link resolution on the unit clause a<b which simply disconnects the link from the unit clause. The shortened link contains the information that the corresponding instance ¬a<b of ¬x<y is false in the theory of the unit clause.)

3.4.1.3 Link Factoring (*lfac*)

The link factoring rule is the analog to the clause factoring rule. It is applicable to a single link which is connected to two or more renamed copies of the same literal node and generates a new link with fewer adjacent literals, but a usually stronger instantiating substitution. In the example below the links 1-3 and 2-4, must be factorized in order to generate the empty link in a subsequent link resolution.
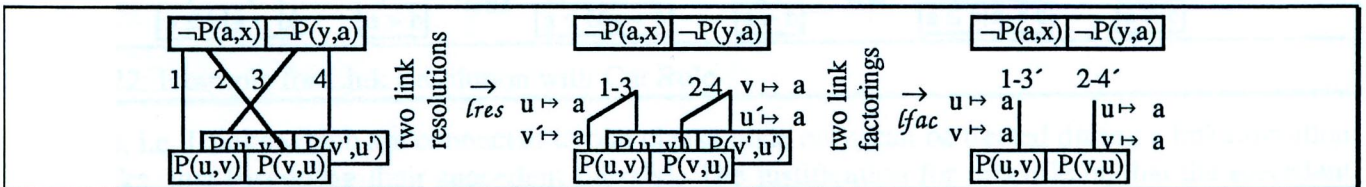
Figure 3-20: Example for Link Factoring

The factorized links 1-3′ and 2-4′ are not equivalent with 1-3 and 2-4 respectively because they instantiate both variables u and v with a, whereas the original links 1-3 and 2-4 leave one variable uninstantiated.

3.4.1.4 R-Link Resolution (*rlres*)

R-link resolution is just a special case of link resolution where only R-links are combined. The interesting thing about R-link resolution is, that this restricted inference rule together with the link factoring rule is refutation complete, i.e. for each unsatisfiable clause set **C** there exists a sequence of R-link resolutions and link factorings on some initial clause graph over **C** that terminates with the empty link. All other operations on clause graphs are in principle not necessary, however, they can considerably increase the efficiency of the deduction system.

3.4.2 Advanced Operations

3.4.2.1 The Link Cut Rule (*lcut*)

The link cut rule is derived from the cut rule of Gentzen's sequence calculus, which states that a new sequence can be derived from two sequences by joining the antecedents and succedents and removing the common parts of the succedent of the first sequence and the antecedent of the second sequence. In the clause graph version this operation works similarly: Two I-links with a common succedent and antecedent literal node may be joined into one new link if the two link substitutions merge. The joined link consists of the union of both antecedents and both succedents respectively, with the common literal node removed:

I-link 1   I-link 2   new I-link
... τ_1 τ_2 ...   ... $\tau_1 \cap \tau_2$ ...

$\rightarrow \ell_{cut}$

An example for an application of the cut rule:

| $a \leq b$ | $a \geq b$ | $a = b$ | $\neg P(a)$ | $P(b)$ |

$\rightarrow \ell_{cut}$

| $a \leq b$ | $a \geq b$ | $a = b$ | $\neg P(a)$ | $P(b)$ |
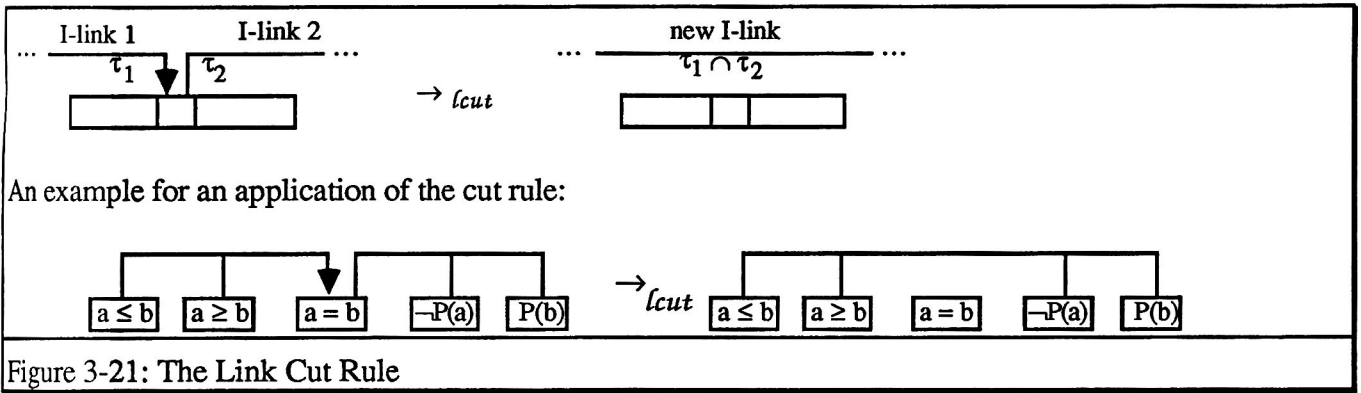
Figure 3-21: The Link Cut Rule

3.4.2.2 Link Resolution with Cut Rule ($\ell rescut$)

The link cut rule can be combined with the link resolution rule giving a more powerful inference rule, which contains an implicit clause factoring operation and is as easy to handle as the link resolution rule itself. As the example below suggests,

1 | 2   1-2 | 2   1-2
$a \leq b$ | $a < b$ $a > b$ $\rightarrow \ell_{cut}$ $a \leq b$ | $a < b$ $a > b$ $\rightarrow \ell_{res}$ $a \leq b$ | $a < b$ $a > b$
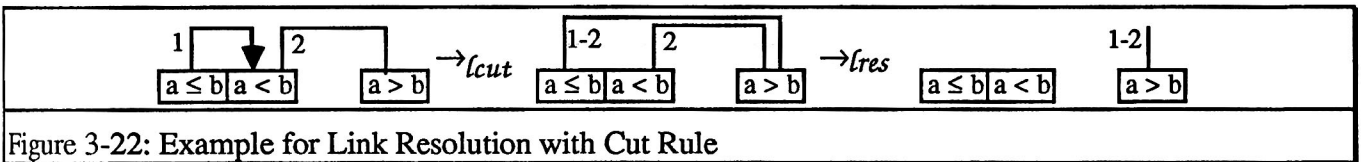
Figure 3-22: Example for Link Resolution with Cut Rule

F-links, i.e. I-links which are connected to one clause node only, can be treated during a link resolution like R-links, just forgetting their succedent literals[1]. The justification for this rule is, that the succedent literal nodes can be removed applying the cut rule to the F-link and the link that is connected to the succedent literal node used in the link resolution. Another viewpoint of this rule is, that the link resolution uses instead of the original clause node the instance of the clause where the corresponding antecedent literal node of the F-link has been removed.

3.4.2.3 Subsumption Factoring ($subfac$)

There are clauses which are subsumed by one of their own factors. $\{P(a,x), P(a,a), Q(x), Q(a)\}$ for instance is subsumed by its factor $\{P(a,a), Q(a)\}$ which can be generated just by removing the superfluous literals $P(a,x)$ and $Q(x)$ from the original clause. The removal of superfluous literal nodes together with all adjacent links may considerably decrease the search space and should be performed as early as possible. Therefore this operation, called subsumption factoring, has the status of a separate inference rule, although it could be achieved with a (more expensive) resolution step followed by a clause subsumption operation. Subsumption factoring is applicable if the clause can be partitioned into two parts L and K such that all literals in L may be removed by resolution and factoring operations without adding new literals and instantiating the literals in K. A subsumption factoring possibility is indicated by a group of F-links and unary R-links with a non empty common link substitution that instantiates only the antecedent literals. A typical situation of this kind looks as follows:

---

[1] This is not totally true. Two or more F-links participating in a $\ell rescut$ operation are not allowed to form a cycle, i.e. the following situation for example has to be avoided:

$$\tau = \tau_1 \cap \tau_2 \cap \tau_3 \cap \tau_4 \cap \tau_5 \neq \emptyset$$

τ instantiates L only

L can be removed
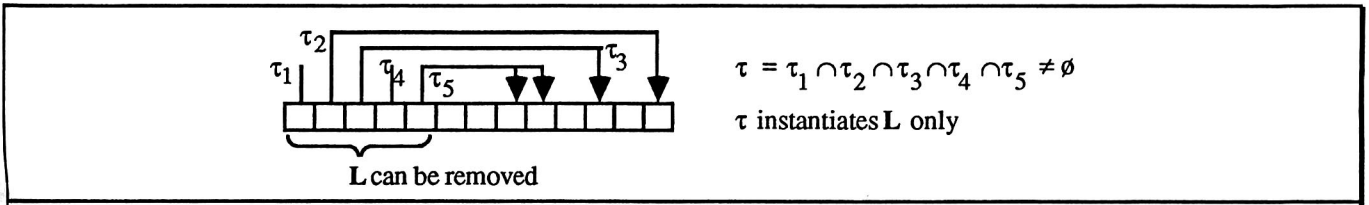
Figure 3-23: Subsumption Factoring

The principle we call subsumption factoring seems to have been invented first by William Joyner [Joyner 73]. He defines a "condensation" of a clause C as an instance that subsumes C; an instantiating substitution resulting in a condensation he calls a "condenser" of C. He presents an algorithm to compute a most specific condenser of a clause and shows that any two most specific condensations of the same clause are variants of each other and variants of a subset of the original clause.

In fact a generalization of subsumption factoring is still possible. As mentioned earlier, all we have to make sure is that the reduced clause follows from the original clause. A subsumption factor follows from the original clause because it is, after all, a factor and thus an instance. But from the clause $\{\neg P(x)\,P(f(x)), \neg P(y)\,P(f(f(y)))\}$ there follows the reduced clause $\{\neg P(y), P(f(f(y)))\}$, which is not an instance. More general, whenever a clause C can be partitioned into two subclauses C_1 and C_0 such that C_1 semantically entails C_0, it is sound to remove all members of C_1 from C, leaving just C_0 as the reduced clause. Subsumption factoring covers the cases where C_1 subsumes C_0. Since subsumption is essentially a syntactic characterization of entailment between two clauses, one can hardly expect significant but still efficiently implementable improvements of the subsumption factoring rule.

3.4.2.4 Subsumption Resolution

The link resolution rule can be combined with subsumption factoring to obtain a more powerful macro operations for removing superfluous literals from clauses. We give some typical examples which can be handled by the algorithms implemented in the MKRP's reduction module [Präcklein 85].
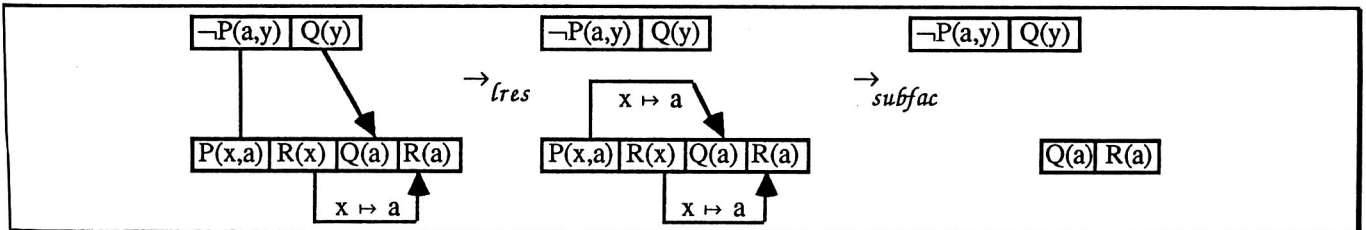
Figure 3-24: Example for Link Resolution enabling Subsumption Factoring
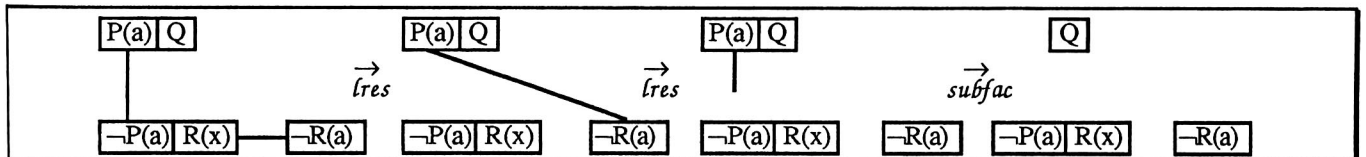
Figure 3-25: Example for Link Resolutions with Unit Clauses enabling Subsumption Factoring

Figure 3-26: Example for exploiting the implicitly existing I-Links between Copies of Literals to enable Subsumption Factoring.
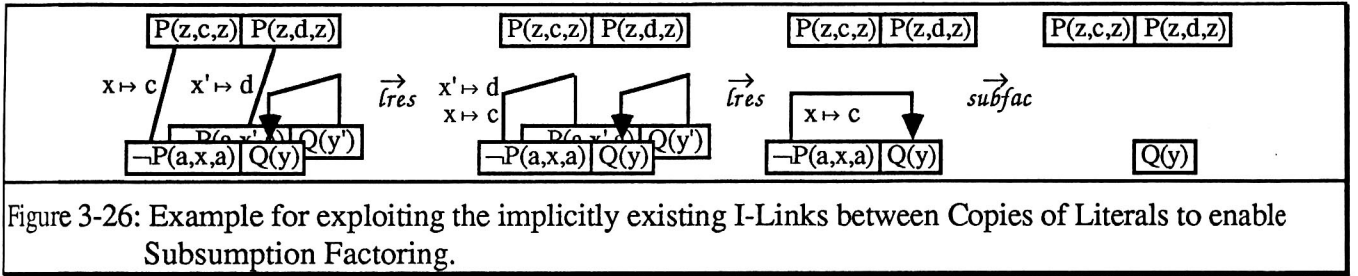
3.4.2.5 Generalizing Subsumption Resolution

The next refinement is called *generalizing subsumption resolution*. It replaces a resolution literal by a more general literal from the resolution partner. In the following example, {P(a), Q} will be changed to {P(x), Q}. Such situations are detected by considering binary I-links between the resolution literal and affected literals.
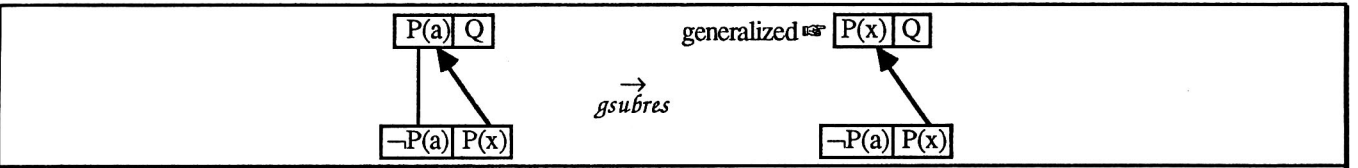
Figure 3-27: Example for Generalizing Subsumption Resolution

Note that this is also a shortcut for resolution followed by subsumption. The advantage of this rule is the better ability of the resulting clause to subsume other clauses. In this way a generalization indirectly reduces the graph.

Strictly speaking, generalizing replacement resolution is not a reduction rule. It does not remove a literal, but overwrites a literal with another one that is properly smaller with respect to the well-founded subsumption ordering. It is a matter of taste whether such rules should be regarded as reduction rules or as a new category of inference rules, beside deduction and reduction rules.

3.5 Clause Graph Reduction Rules

The clause graph inference rules defined above introduce new objects into the graph that represent information explicitly which was contained implicitly in the previous state of the graph. Previous clauses and links may therefore become worthless because their information is fully contained in derived links and clauses. They should be removed as soon as possible. In this section we therefore present a number of rules for removing redundant objects from the graph. An object - a clause, a link or a link substitution - is redundant if its removal turns a graph which is refutable by (clause or link) resolution and (clause or link) factoring into a graph which is still refutable with these two rules.

3.5.1   Clause Deletion Rules

3.5.1.1   Clause Purity (*cpur*)

In the original definition of the clause purity rule, a clause can be removed if one of its literals is not connected to any R-link. The reason is that this literal would remain part of any further resolvent with this clause, and could therefore never be used to generate the empty clause.

The clause purity condition can be slightly weakened: A clause can be removed if it contains a literal that is connected with at most some internal R-links. (Internal R-links connect different literal nodes of different copies of a clause node.)
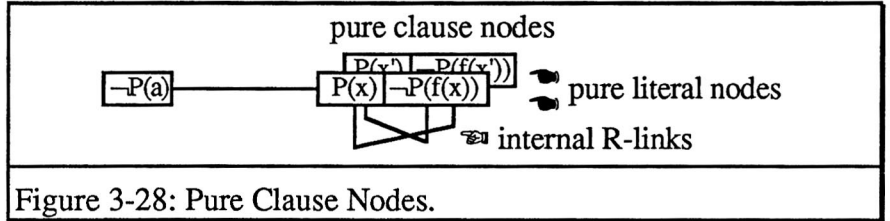
Figure 3-28: Pure Clause Nodes.

Pure clause nodes can never be used in a refutation because it can be shown that the link resolution rule transforms internal links always into internal links, but the last step with a clause node in a refutation must involve external links only.

3.5.1.2   Clause Tautology (*ctau*)

A tautologous clause is true in every interpretation and can therefore be removed from a set of unsatisfiable clauses. A general clause may be regarded as a representation for a certain set of its ground instances, some of those may be tautologies, some others may not. For example the ground instances of $\{P(x), \neg P(y)\}$, which instantiate the variables x and y with equal terms are tautologies and can be removed, all others are not. The whole clause can be removed if only tautologous ground instances remain. The case, where only some ground instances can be removed, makes not much sense in the basic resolution calculus because the clause still remains part of the clause set. In the clause graph environment, however, this rule can be used for further link deletions. Consider the example in figure 3-29:
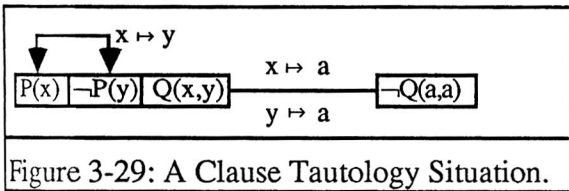
Figure 3-29: A Clause Tautology Situation.

All substitutions which are instances of $x \mapsto y$ generate tautologous instances of the clause C. If these substitutions are marked to be illegal, all the link substitutions of the attached links which are themselves instances of the removed substitutions become illegal too and can be removed as well. In the example above, the whole R-link can be removed because $\{x \mapsto a, y \mapsto a\}$ is an instance of the "tautologous substitution" $\{x \mapsto y\}$.

The actual *ctau*-rule consists of three parts: recognition - completion - deletion.

➤ Recognition

Tautologous instances of a clause are indicated by internal T-links, i.e. T-links which are connected only with the corresponding clause node. All link substitutions of such a T-link denote tautologous instances of the clause which therefore can be removed. The whole clause can be removed if no clause substitution remains.
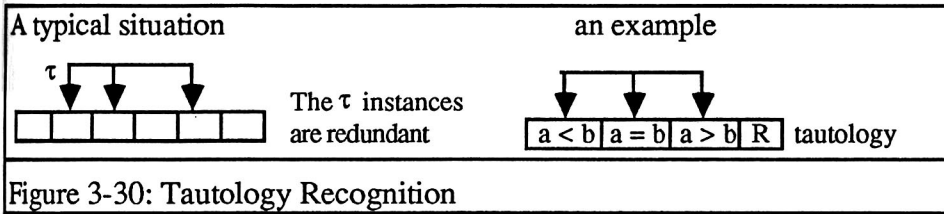
A typical situation	an example	(In an actual implementation the redundant instances of a clause must be stored as a set of "instantiation limits")

Figure 3-30: Tautology Recognition

> Completion

A tautologous clause can be removed from an unsatisfiable clause set and it will remain unsatisfiable. As the following example demonstrates, this does not imply, that a tautologous clause node can be removed without further provisions from a refutable clause graph without losing its refutability [Bibel 81]:
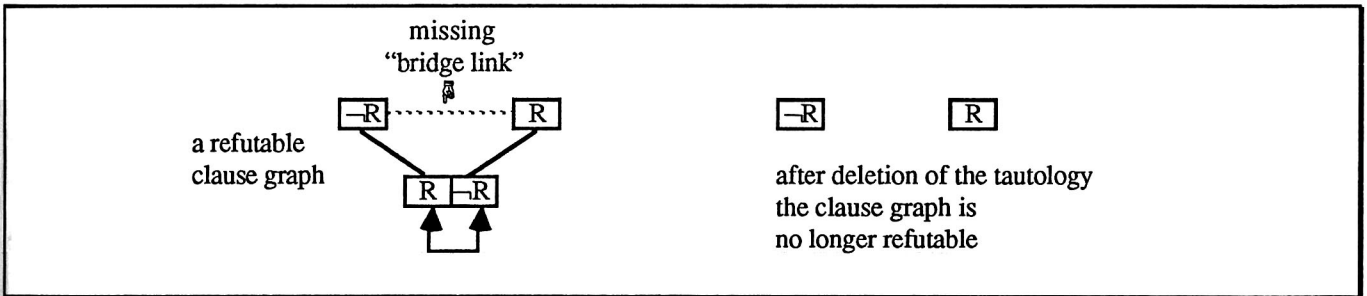
Figure 3-31: Clause Tautology Bridge Link Condition.

The problem is the missing "bridge link" between R and ¬R that would allow for another refutation. (This link might have been removed in previous steps by some link reduction rules.) One could either renounce on removing tautologies if such links, or the corresponding link substitutions respectively, are missing, or insert them before the clause node is removed. Since removing a clause node usually shrinks the search space much more than the insertion of one special link increases it, we decided to formulate the clause tautology rule in the second way: tautology removal with link insertion. It depends on the structure of the clause node´s T-links which links have to be actually inserted before a tautology can be removed,

> Deletion:

All "tautologous" instances of the clause are taken to be illegal. If there are no remaining clause instances, the whole clause node is deleted. Furthermore the link substitutions of all adjacent links which are instances of the illegal clause instances are also removed. Links with an empty link substitution are deleted altogether.
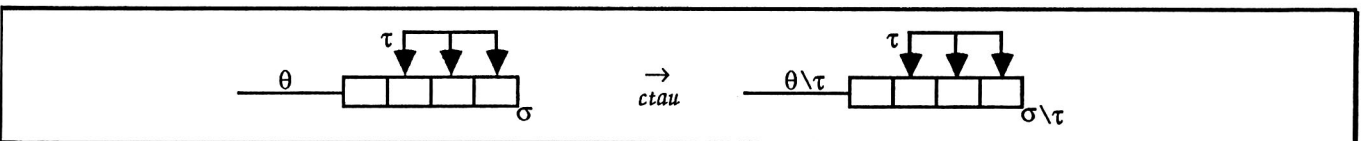
Figure 3-32: General Scheme for Clause Tautology Deletion.

3.5.1.3 Clause Subsumption (csub)

A clause of an unsatisfiable clause set C, which is implied by some other clauses in C, can be removed without losing unsatisfiability. A full application of this deletion rule is not only impossible, because of the undecidability of this implication problem, but it is also undesirable because derived clauses may be useful in finding the refutation. Therefore one is interested only in a restricted version, where the implication can be easily detected (subsumption) and where the removal of the implied clause does not lengthen the refutation. Similar to the clause tautology rule, we do not only consider the case where a subsumed clause can be completely removed, but it is also possible to declare some instances of a clause as illegal, which are

subsumed by another clause. This rule can trigger further link deletions, as is shown in the following example:

Figure **3-33**: Clause Subsumption
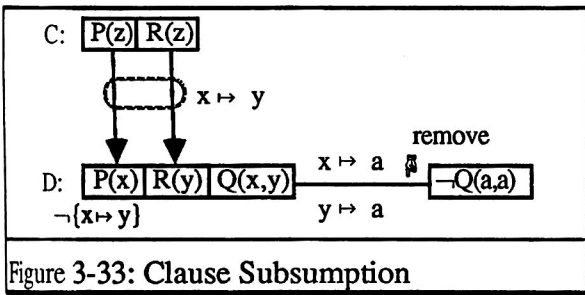
All instances of D where x and y become equal are subsumed by C. The R-link can therefore be removed.

The clause subsumption rule consists also of the three parts: recognition - completion - deletion:

➤ Recognition:

In the environment of clause graphs, a clause subsumption situation can be easily detected using binary I-links: There must be a set of binary I-links with compatible link substitutions which map the literal nodes of the subsumer injectively to the literal nodes of the subsumed clause (however see [Eisinger 81] for a generalization of this basic observation):
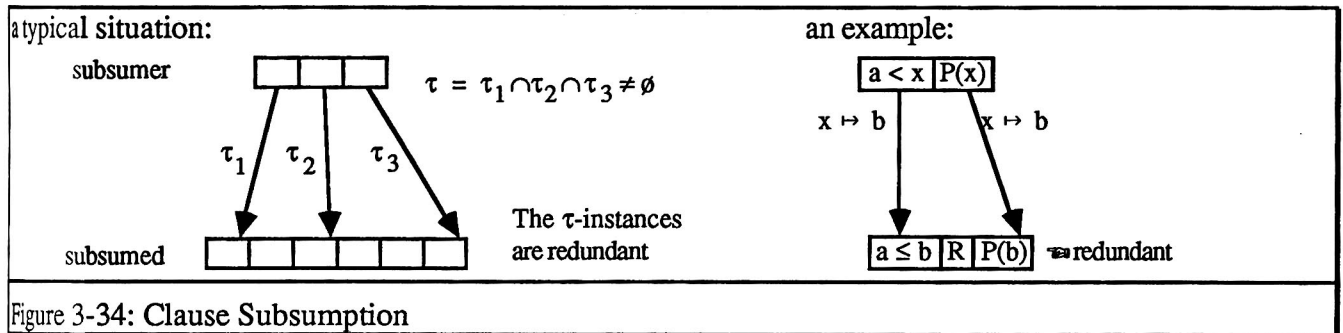
Figure 3-34: Clause Subsumption

➤ Completion:

Again clause nodes which are logically superfluous cannot be removed from a clause graph without further precautions. Due to certain link deletion rules, the subsumer may have lost some links which are necessary for a refutation without the subsumed clause node or the removed instances, respectively. These links, or the corresponding link substitutions respectively, must be reinserted before the subsumed clause node can be removed. The example below shows such a situation. Before the subsumed clause node $P(a)$ can be removed, a link between $\neg P(a)$ and $P(x)$ must be inserted into the graph.
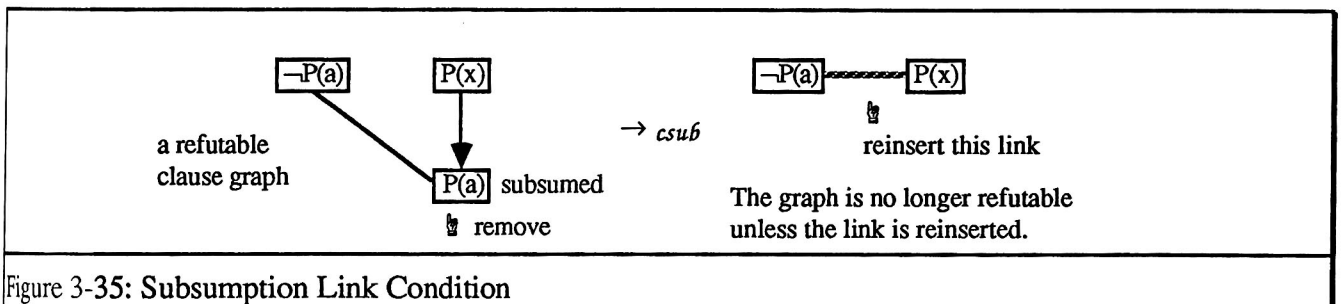
Figure 3-35: Subsumption Link Condition

➤ Deletion:

All "subsumed" clause instances are taken to be illegal. If there are no remaining clause instances, the whole clause node is deleted. Furthermore the link substitutions of all adjacent links, which are instances of the illegal clause instances are also removed. Links with an empty link substitution are completely deleted.

3.5.2 Link Deletion Rules

A link or at least some of its link substitutions may be removed either if it can be shown that the link cannot contribute to a refutation or if there is some other link in the graph, which can take its role in a refutation.

3.5.2.1 Link Incompatibility (*linc*)

The general idea of this deletion rule is to instantiate the link substitutions such that they represent only those ground substitutions which can actually be used in a link resolution. The example below illustrates the idea.

Figure 3-36: Example for Link Incompatibility
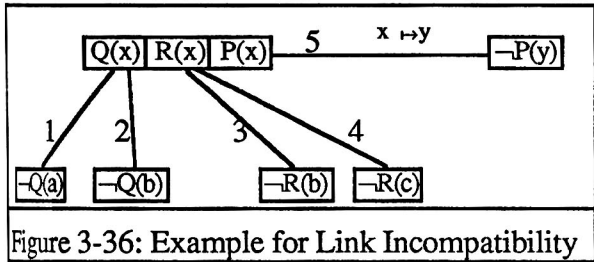
The only possible link resolution is 2,3,5 yielding a merge substitution $\{x \mapsto b, y \mapsto b\}$. The link substitution of link 5 may therefore be reduced to $\{x \mapsto b, y \mapsto b\}$.

The rule works as follows: Given a link, select one of its antecedent clause nodes as base clause node. Compute the merge substitutions of all possible link resolutions with this link and the selected base clause node. Reduce the link substitutions to the computed set of merge substitutions.

Of course this might be very expensive, but it may be used to start a constraint propagation sequence which can cause a snowball effect of further deletions, as the following example demonstrates:
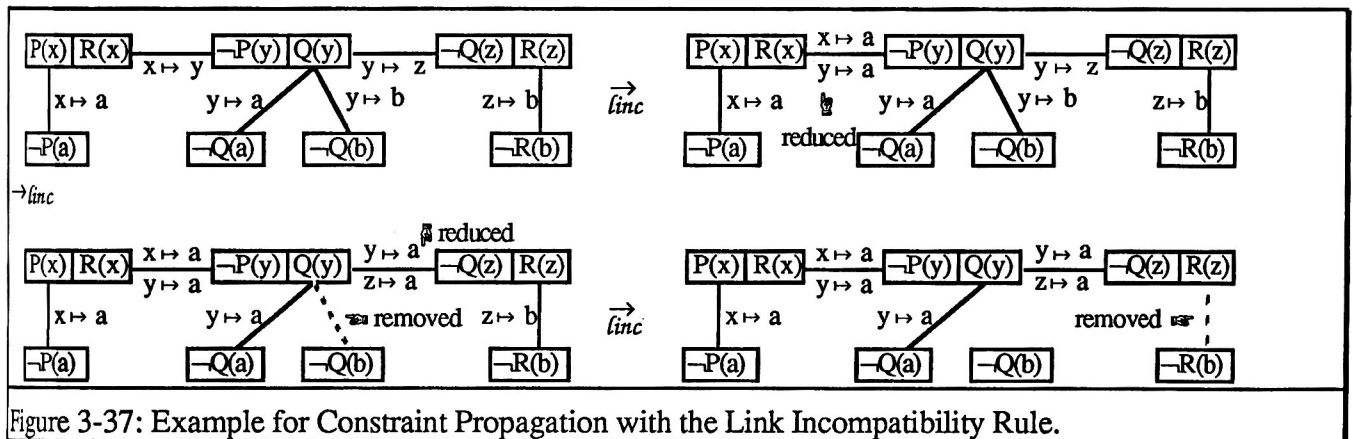
Figure 3-37: Example for Constraint Propagation with the Link Incompatibility Rule.

It is noted that clause tautology and clause subsumption deletion may also trigger a constraint propagation sequence via the link incompatibility rule.

3.5.2.2 Parallel Link - Link - Subsumption (*plsub*)

Consider the situation in the following figure:
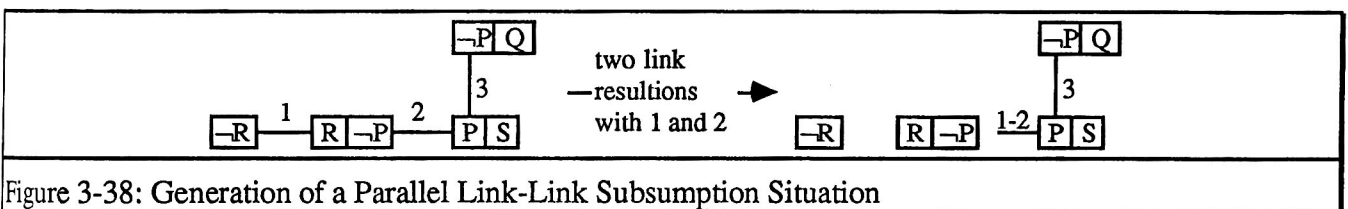
Figure 3-38: Generation of a Parallel Link-Link Subsumption Situation

The combined link 1-2 contains the information that the literal P must be false, whereas the link 3 contains only the information that P and ¬P are contradictory. Link 3 is subsumed by link 1-2 and can therefore be removed without losing information. The *plsub* rule allows the removal of substitutions from

any link which is "larger" than another one, i.e. its antecedent and succedent literal nodes are supersets of the antecedent and succedent literal nodes of a "parallel" link and its link substitution is an instance of the link substitution of the parallel link.
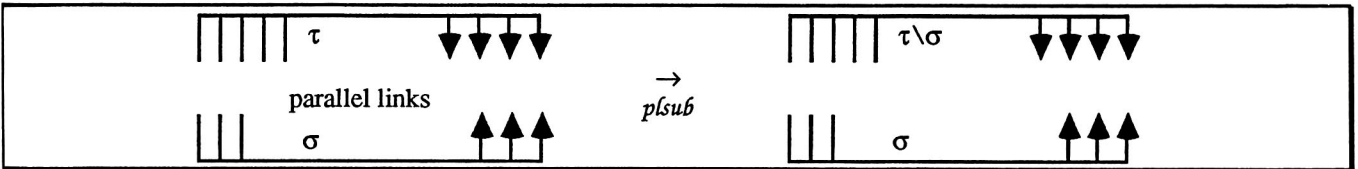
Figure 3-39: General Scheme for parallel Link-Link Subsumption

3.5.2.3 Link Tautology (*ltau*)

The link tautology rule is a consequent extension of the clause tautology rule. Each link with a non empty antecedent represents a resolvent and if this resolvent has tautologous instances which can be removed, these instances should already be removed from the link substitution. Consider for instance the following graph:
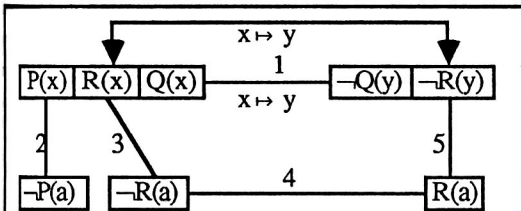
Figure 3-40: Example for Link Tautology

Link 1 represents a tautologous resolvent and is therefore useless. Since resolution need not be the main operation of our procedure, we need another justification for such a removal. And in fact, link resolutions involving the corresponding instance of the link substitution of link 1 and its successors generate a combined link, which is subsumed by the "bridge link" 4:
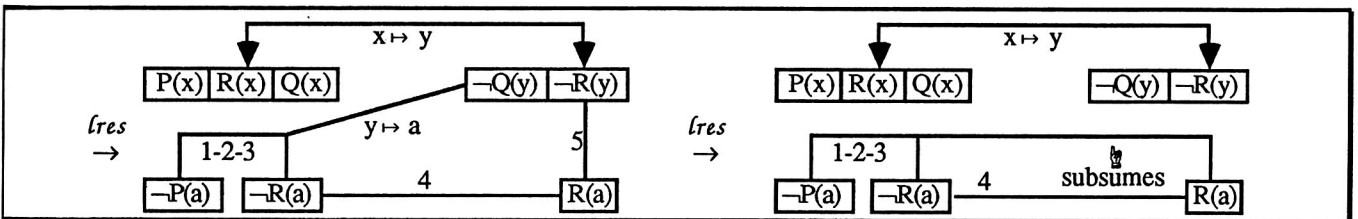
Figure 3-41: Bridge Link subsumes Tautologous Instances of an R-Link.

This effect is no coincidence and "tautologous" instances can be removed from link substitutions - provided the graph contains enough "bridge links".

Link tautologies can easily be recognized using T-links which are parallel to the R-link under consideration. In contrast to the clause tautology rule, reinsertion of missing bridge links is not a good idea because the comparatively small effect of the *ltau* rule does not justify the reinsertion of previously removed links. A typical situation is shown below.
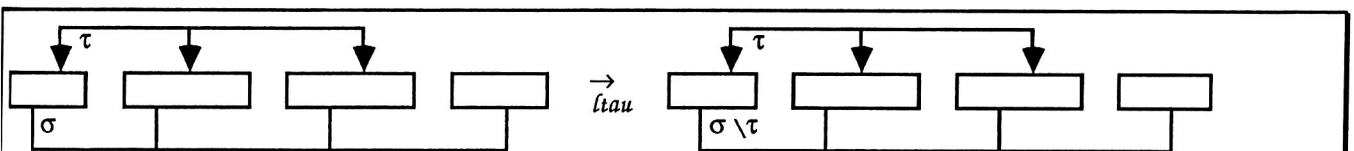
Figure 3-42: General Scheme for Link Tautology Deletion.

3.5.2.4 Clause-Link Subsumption (*clsub*)

Again, since links with non empty antecedent represent resolvents where instances may be subsumed by other clauses, we can remove the subsumed instances already from the link's substitutions. The actual algorithm is very similar to the clause subsumption algorithm [Eisinger 81]. A typical situation is shown in the following figure.
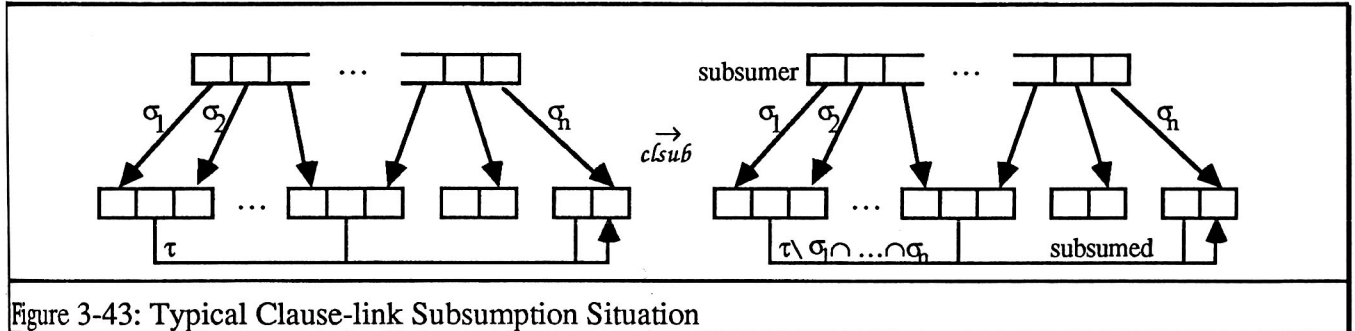
Figure 3-43: Typical Clause-link Subsumption Situation

This terminates our list of operations on clause graphs. Inventing further useful operations is subject to ongoing research. Hopefully, the examples in this chapter convince the reader that the possibilities in clause graphs have as yet by no means been exhausted. More ideas can for example be found in [Socher-Ambrosius 89].

3.6 Properties of Clause Graph Resolution

Norbert Eisinger has investigated the theoretical properties of Kowalski's original clause graph procedure with the essential kernel operations clause resolution, purity, clause tautology and clause subsumption deletion. Since the corresponding deletion operations on links can be seen as extensions of these kernel operations in the sense that they can be imitated by the successive generation and then the deletion of the redundant resolvent and link, his results cover also these extended reduction rules. The generalization to clause graphs for theory resolution seems to be possible without further difficulties. We briefly present Eisinger's framework and results. Details can be found in [Eisinger 86, 88].

3.6.1 Logical State Transition Systems

Several problems that are either vacuous or trivial for clause set resolution, become considerably hard in the context of clause graph resolution. Their description requires an adequate level of abstraction going beyond traditional notions of completeness, and require new notions of the essential properties a "good" deduction system is required to possess.

A *state transition system* consists of a set S of *states* and a binary relation $\rightarrow$ on S, the *transition relation*. Frequently $\rightarrow$ is the union of some simpler relations conceived as a set of elementary transition rules. There are two distinguished subsets of S, the *initial states* and the *final states*. A sequence of states successively related by $\rightarrow$, beginning with S and ending with S', represents a *derivation* of S' form S. As usual, $\xrightarrow{+}$ and $\xrightarrow{*}$ denote the transitive and reflexive-transitive closure of $\rightarrow$. A state S' is *reachable* if S $\xrightarrow{*}$ S' holds for some initial state S, and *unreachable* otherwise. With the appropriate restriction of the transition relation the reachable states define the *reachable subsystem* of a state transition system.

The selection from among the possible inference steps and the administration of the sequence of steps already performed and states thereby produced are subject to a separate constituent named the *control strategy*. Control strategies come under two major classes: when applying an inference rule, *tentative* control strategies make provisions for the later reconsideration of alternatives, whereby *irrevocable* control strategies do not. Backtracking and hill-climbing, respectively, are prominent examples of the two types of control strategies. Tentative control strategies essentially require the storage of more than one state at a time, which tends to render them unfeasible for state transition systems with complex states.

If the states of a state transition system represent logical formulae and the transitions are logical inference steps, it ranks as a *logical state transition system*. A logical state transition system together with a control strategy make up a *proof procedure*.

The following property characterizes a *commutative* state transition system: whenever two transition rules can be applied to some state, each of them remains applicable after application of the other, and the resulting state is independent of the order in which the two steps are performed. The advantage of commutative state transition systems lies in their automatic admittance of irrevocable control strategies, because the choice of an irrelevant rule only delays, but never prevents the "right" steps.

Most logical state transition systems happen to be commutative. For the first famous example, the lambda calculus, a weaker property bearing the name of its investigators Church and Rosser could be shown. Equivalently, a state transition system is *confluent*, if for all states S, S_1, S_2 with $S \overset{*}{\to} S_1$ and $S \overset{*}{\to} S_2$ there exists a state S' with $S_1 \overset{*}{\to} S'$ and $S_2 \overset{*}{\to} S'$. In other words, any two derivations from the same ancestor can be continued to a common successor. A less restrictive requirement than commutativity, confluence still allows for irrevocable control strategies, especially in Noetherian systems, where no derivation of infinite length exists.

The states of clause graph resolution are clause graphs. The inference rules Eisinger has considered are binary resolution and factoring, both including removal of the resolved link as an integral part, further merging of equal literals, purity removal, tautology removal and subsumption. Let $\emptyset$ denote the empty graph comprising neither clause nor links (the affirmation state of the logical state transition system) and let $\{\square\}$ denote the graph composed of just the empty clause (the refutation state of the logical state transition system). These are the final states of the clause graph resolution system, with the convention making subsumption compulsory on graphs that contain the empty clause. Only certain states are admissible as initial, namely clause graphs in which every possible link actually exists. Obviously each clause set S can be uniquely converted into such a clause graph, called the *initial* clause graph for S, denoted INIT(S).

3.6.2 Properties Relevant to the Inference System

The basic notions describing qualities of interest for logical state transition systems are soundness and completeness, further confluence and Noetherianess. For a more concise terminology (and in particular, as these traditional concepts do not hold in general for clause graphs)it appears useful to restrict these concepts to the reachable subsystems and even to subdivide the definitions according to the logical status of the initial state.

Given a clause graph S, clause graph resolution is called

refutation sound	iff	INIT(S) $\overset{*}{\to}$ $\{\square\}$ $\Rightarrow$ S is unsatisfiable;
refutation complete	iff	S is unsatisfiable $\Rightarrow$ INIT(S) $\overset{*}{\to}$ $\{\square\}$;
refutation complete	iff	S is unsatisfiable, INIT(S) $\overset{*}{\to}$ G_1 and INIT(S) $\overset{*}{\to}$ G_2 $\Rightarrow$ $G_1 \overset{*}{\to}$ G' and $G_2 \overset{*}{\to}$ G' for some G';
affirmation sound	iff	INIT(S) $\overset{*}{\to}$ $\emptyset$ $\Rightarrow$ S is satisfiable;
affirmation complete	iff	S is satisfiable $\Rightarrow$ INIT(S) $\overset{*}{\to}$ $\emptyset$;
affirmation complete	iff	S is satisfiable, INIT(S) $\overset{*}{\to}$ G_1 and INIT(S) $\overset{*}{\to}$ G_2 $\Rightarrow$ $G_1 \overset{*}{\to}$ G' and $G_2 \overset{*}{\to}$ G' for some G'.

The strongest property a deduction system should enjoy, is called the strong *completeness* property, which says that a (class of) control strategy(ies) exists that actually *finds* the refutation, in case one exists. This distinction is the major one to classical deduction calculi like resolution: completeness, i.e. the notion that there exists a refutation, suffices as a search can always be arranged (e.g. level saturation) such that this refutation will eventually be found by a proof procedure. This unfortunately does not hold for clause graph theorem proving.

Refutation soundness means that deriving$\{\square\}$ entails the unsatisfiability of the initial state. The dual property affirmation soundness says that from reaching an affirmation state the satisfiability of the start state follows. Refutation completeness expresses the existence of a refutation from each unsatisfiable initial state. Dually, affirmation completeness guarantees that there exists an affirmation state in the case of satisfiability. Clause set resolution is refutation complete, but of course affirmation complete only for certain decidable classes, e.g. the ground case, for the Herbrand class where each clause is a unit and for a certain class $S_1^+ \cup S_1^-$ of segregated formulae, where no clause contains literals of different sign and either all positive or all negative clauses are units.

Confluence, and in most cases even commutativity, is an immediate property of clause set resolution, for which the distinction between confluence and affirmation confluence is unnecessary. Refutation confluence carries weight in that it allows for irrevocable control strategies in the most relevant subsystem. Suppose some unsatisfiable set S is refutable, i.e. $\text{INIT}(S) \xrightarrow{*} \{\square\}$, and we perform an arbitrary series of inference steps leading to a graph G, i.e. $\text{INIT}(S) \xrightarrow{*} G$. Refutation confluence then ensures $G \xrightarrow{*} \{\square\}$, i.e. from G we can somehow proceed and reach the empty clause. In other words, any derivation can be continued to a refutation, and there are no dead ends that require a backup to earlier states in order to derive $\{\square\}$.Thus refutation confluence is absolutely indispensable for clause graph resolution to be of any practical value. The dual property affirmation confluence lacks this significance as long as the system is primarily intended to refute unsatisfiable clause sets.

3.6.3   Results for the Inference System

This section summarizes, which of the above properties clause graph resolution does or does not enjoy.

Clause graph resolution is
>   refutation sound,
>   refutation complete,
>   refutation confluent,
>   affirmation sound,
>   affirmation complete for the ground case, for the Herbrand class and for the class $S_1^+ \cup S_1^-$,
>   *not* affirmation confluent,
>   refutation complete, refutation confluent and affirmation sound for the unit refutable class with unrestricted tautology rule, i.e without the special "bridge link" condition [Smolka 82],
>   *neither* refutation confluent *nor* affirmation sound with unrestricted tautology rule.

No complete and sufficiently *simple* strategy is known, i.e. the strong completeness problem is still open. (Of course running an ordinary resolution theorem prover in parallel with the clause graph theorem prover and using its results to guide the link selection yields a complete strategy.)

Note that the system with an unrestricted tautology rule is the system as defined by Robert Kowalski. Thus with the original version of clause graph resolution there exists a refutation for each unsatisfiable formula, but an attempt to find it can lead to dead ends from which it can not recover to find the refutation. This result clearly overshadows the traditional notion of completeness alone, and in fact caused serious doubts about the traditional notions and analysis of deduction systems. See [Eisinger 86] for a discussion of the deep consequences. In the case of subsumption there is an example where $\emptyset$ is derived from an unsatisfiable unit refutable initial graph, using only resolution, purity removal and forward subsumption which violates the link condition.

N. Eisinger also investigated the properties of the clause graph versions of several classical control strategies like set-of-support, linear etc. (with mostly negative results). The traditional control strategies, however, are not that significant for the clause graph procedure, because heuristics exploiting the topological properties of the graph are much more important. Furthermore the practical incompleteness caused by time and memory limitations totally prevail the theoretical problems as is presumably the case for all deduction calculi.

4 EQUATIONAL REASONING

Equality is the most prominent relation to be built into an automated deduction system, as it provides an important representational basis for the formulation of first order theories [Tarski 68, Taylor 79]. Unfortunately it also posed some of the most formidable obstacles against its computational treatment. In particular, the explicit and naive use of the standard equality axioms (reflexivity, symmetry, transitivity and substitution axioms) turned out to be insufficient as a basis for computer oriented equational reasoning. Hence the last twenty years saw a proliferation of deduction techniques to incorporate the equality relation somehow directly into the inference machinery. There are currently at least four research communities with their own international conferences that support active research in equational reasoning: logic programming, unification theory, term rewriting systems and general automated deduction. Their respective aims are to develop:

(i) Special purpose unification algorithms for common equational axioms such as as associativity, commutativity, Boolean rings or Abelian Groups (see [Siekmann 89] for a survey of the field of unification theory and [Bürckert & Nutt 89] for the most recent workshop of this field)

(ii) General purpose unification algorithms based on narrowing [Nutt et al 89] or decomposition [Kirchner 87] that solve equality problems in a given *class* of equational theories. They are mainly developed for logic programming with equality [Smolka et al 89].

(iii) Demodulation and term rewriting systems (see [Huet & Oppen 80] and [Buchberger 87] for a survey and [Dershowitz 89] for the most recent proceedings of the conference in this field).

(iv) General inference rules to handle equality, usually in combination with a standard inference rule such as resolution (standard references are: [Wos et al 67] [Morris 69] [Sibert 69] [Anderson 70] [Brand 75] [Shostak 78] [Digricoli 79] [Lim & Henschen 85], the most recent proceedings of the conference supporting this field is [Lusk & Overbeek 88]).

This section presents a technique that integrates the second and the fourth approach with the essence of clause graph theorem proving and summarizes our experience in mechanizing equational reasoning: During a timespan of more than eight years, our original plans of building equality into our connection graph based deduction system [Siekmann &Wrightson 80] changed considerably in the light of experimental evidence.

Equational axioms - or the equality literals within a clause - have been used to pursue essentially two goals:

1. The *simplification* of a given state (i.e. a literal, a clause, a clause graph etc.)

2. The *reduction* of the *difference* between two given states

We like to classify the above techniques and logical calculi that incorporate equality according to these to criteria.

For example if an equation can be directed such that the right hand side becomes "smaller" or "simpler" in some sense than the left hand side, this directed equation can advantageously be used in a simplification task. Demodulation or term rewriting systems are good examples for this first approach.

The second goal rejects the indiscriminate use of equality and claims that no working mathematician would apply an equationally represented fact, unless used directly for some purpose, i.e. only *"if needed"*. In the context of a resolution based theorem prover this observation could be technically rephrased as using an equality axiom only for the purpose to remove the difference between two complementary literals. Both approaches, simplification and difference reduction, have their advantages and are worthy of investigation. However, whereas most of the current research is based with remarkable success on simplification techniques, especially on demodulation and term rewriting, difference reduction methods have found less attention, although they are at least as important for an automated deduction system.

The potential of difference reduction methods will be illustrated in this section.

4.1 Difference Reduction Methods

Among the various methods proposed within the fourth paradigm (iv), is *paramodulation* [Robinson & Wos 69]: given the solution rule of inference, the equality axioms become superfluous except for the revlexivity axiom [Rusinowitch 87].

More formally: the equality axioms

(E1)	$x=x$	(reflexivity)
(E2)	$x=y \Rightarrow y=x$	(symmetry)
(E3)	$x=y \wedge y=z \Rightarrow x=z$	(transitivity)
(E4)	$x_i=y \wedge P(x_1,...,x_i,...x_n) \Rightarrow P(x_1,...,y,...x_n)$	

(substitution for each n-ary predicate symbol)

(E5)	$x_i=y \wedge f(x_1,...,x_i,...x_n) \Rightarrow f(x_1,...,y,...,x_n)$	

(substitution for each n-ary function symbol)

can be replaced ("built-in") except for (E1) by one rule of inference, paramodulation, as defined in the introduction.

Although paramodulation was a substantial improvement compared to the axiomatical formalization of the equality relation, it still leads to enormous search spaces, as this inference rule can be applied almost everywhere in the clause space. For example the moderately difficult problem, that in an associative system with degree two (i.e. $x^2 = e$) we have commutativity, has an estimated space of about 12^{10} clauses [Bundy 83]. Hence the application of paramodulation must be tightly controlled.

The problem is:

(i) how to represent the information upon which this control can be based and

(ii) how to provide the equational reasoning system with a more goal oriented behaviour.

The analysis of the second problem led to a technique, *E-resolution,* that is a particularly good representative for the "if needed" approach.

The aim of J.B.Morris´ E-resolution [Morris 69] is to remove the differences between corresponding terms of two complementary literals (i.e. the same predicate symbol with opposite sign), such that an inference step by resolution becomes possible. E-resolution may be viewed as a sequence of paramodulation steps applied to two potentially resolvable literals until they are unifiable. It is then followed by the appropriate resolution step, and intermediate clauses are discarded.

This is a generalization of ordinary unification based resolution, and is a specialization of the more recently developed theory resolution technique of M.Stickel as defined in the introduction. Similar to the "means-end" analysis in GPS [Newell et all 59], the role of equality in E-resolution is that of a "means to an end" i.e. to produce favorable preconditions for a resolution step. Two potentially resolvable literals could be selected by some global search strategy, followed by the search for appropriate equations, these equations are applied if the differences can be completely removed.

E-resolution defines the cooperation between resolution and equality reasoning in general terms, but the calculus does not actually support the achievement of its goal: there remains the difficult search problem to find the appropriate equality steps. Since equality of two terms with respect to a given set of equations is undecidable in general, it is impossible to continue searching for equations until the potentially resolvable literals under consideration are definitely unifiable or not. Hence the main problem is: How to find the appropriate E-resolution steps?

The search problems of the paramodulation steps necessitated by E-resolution are hard to solve and led to the conclusion that in most cases an E-resolution step is just too large. Realizing this, V.J. Digricoli proposed a form of partial E-resolution, which he called RUE-resolution [Digricoli 85]. Essentially his calculus works similar to 'E-resolution', however, a deduction step is also possible if the differences between corresponding terms cannot be removed completely. The partial unifier is applied to the literals, the remaining disagreement pairs are computed and added as negated equations to the derived clause, similar to D. Brand's modification method [Brand 75].

Although this is currently the most interesting method to build in equality, there are problems: the major one is the *permissiveness* of this inference rule, as now any pair of literals with the same predicate symbol can be resolved and the burden of the deductive machinery is shifted upon the equality reasoning component, that handles the inequalities (actually in the most cases the weakest part of a deduction system).

Just as E-resolution has nothing to say on how the particular paramodulation steps are to be found, RUE-Resolution shifts the burden upon the inequality reasoning part and cannot say much about the resulting search problems.

Taking this point of view, we extended RUE-Resolution in the sense that graph-based information is available to guide the search for the appropriate equality units.

4.2 Paramodulated Clause Graphs

Our first attempt to build equality into the clause graph procedure tried to extend its essential idea to equational reasoning and represented the potential paramodulation steps as special links (we call them *"P-links"*) [Siekmann &Wrightson 80]. P-links connect one side of an equation with a unifiable subterm in another literal. There are two problems with this approach: The first one is inherent in the properties of equality: there are extremely many possibilities for paramodulation, and therefore extremely many P-links. Especially equations having variables at one or both sides, as for example the injectivity $\{f(x){\neq}f(y), x{=}y\}$, are connected with each subterm in the whole clause set. The total number of links can be reduced a little by erasing links from equations *into* variables. It is a natural restriction because paramodulation into a variable generates just an instantiation of the parent clause and is therefore unnecessary. The harder problem, however is the definition of a complete link inheritance mechanism for the paramodulated literal. This literal is no instance of its parent literal and may therefore be unifiable with a third literal which is itself not unifiable with the parent literal. Take for instance the three clauses $\{P(a)\}$, $\{a = b\}$ and $\{{\neg}P(b)\}$. After paramodulation into P(a) we obtain P(b) which is unifiable with ¬P(b) and has to be connected with a resolution link. This link can be generated using the information of the P-link connecting the right hand side of the equation with ¬P(b). That means we inherit a P-link as a resolution link. This works. What does not work is the mechanism for the restricted P-link set having no links into variables. As the example in figure 3-44 suggests it is necessary to generate a P-link into terms which are formed by an instantiation during a resolution, and these links should usually be inherited from links into variables.

Figure 4-1: A Link Inheritance Problem
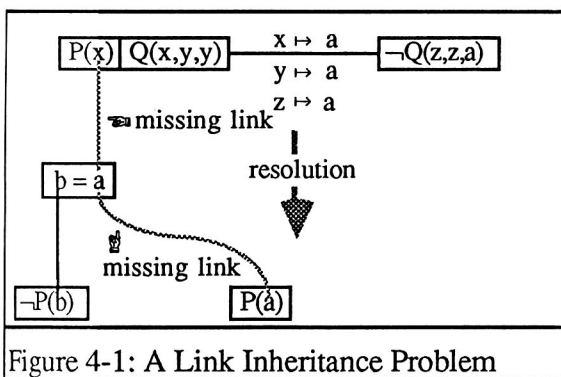
There are two other possibilities to get this link: Either we construct links not only into literals, but also *into* unifiers or we generally reconstruct all the links after each step, but then we loose the bookkeeping and purity generating effect of the link removal after an operation. Our current implementation allows to optionally construct the links into variables, but forbids paramodulation upon such a link.

All these problems with P-links and their inheritance have only technical character. Much deeper, however is the problem to select a paramodulation step. Our experience with many theorem proving

sessions has shown that the information contained in one P-link is far too weak to make a substantiated proposition for a paramodulation step. Except from simplifying rewrites, which should always be performed as soon as possible, it is necessary to analyze very long paramodulation chains for getting a good guess if they are useful or not, which we shall now present.

4.2.1   Constraints in Paramodulated Clause Graphs

Our second attempt to solve general equality problems was still based on paramodulated clause graphs, however they were only searched for compatible combinations of P-links. A *compatible* combination of P-links represents an executable paramodulation sequence, that modifies two literals such that they become resolvable [Bläsius 83]. Such paramodulation sequences can be displayed in a graph, which represents a solution of the given problem. The following example shows such a graph:

$$P(f(a))$$
$$|\,1$$
$$a\ =\ b$$
$$|\,2$$
$$f(b)\ =\ c$$
$$|\,3$$
$$\neg P(c)$$
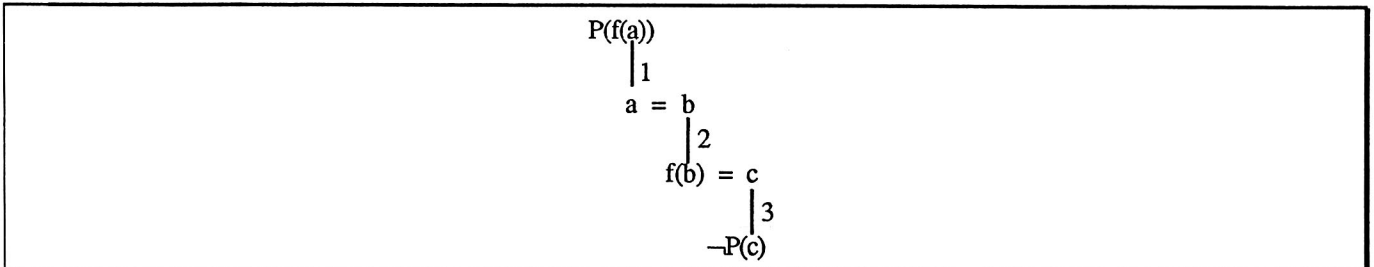
Figure 4-2: An executable Paramodulation Sequence

However not every graph represents an executable, i.e. compatible, sequence of paramodulation steps as the following example show:
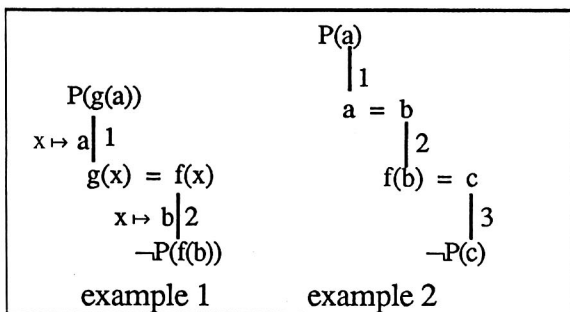
example 1   example 2

Figure 4-3: Incompatible Combinations

The combination of P-links 1 and 2 in example 2 is impossible because their unifiers {x ↦ a} and {x ↦ b} are incompatible. In example 2 the P-links 1 and 2 are incompatible, because after paramodulation on link 1, link 2 cannot be inherited to the paramodulant P(b) since the access depths do not coincide. An equality graph that cannot be executed is called *incompatible*, otherwise it is *compatible* .

The problem is to find compatible graph structures. To this end certain constraints can be stated, which are necessary but not sufficient for the compatibility of a graph and which are fast and inexpensive to test.

Some of the tested constraints are:

(1)   All unifiers of the involved P-links must merge to one most general unifier.

(2)   For each maximal chain of P-links in a graph the sum of all access depths must be equal to zero and each partial sum must be less than or equal to zero.

(3)   Each combination of P-links containing an incompatible substructure is incompatible too.

Practical experiments with a procedure based on the above constraint satisfaction method have shown however, that the set of potential graphs (i.e. combinations of P-links), which have to be created in order to test for compatibility, is still far too large, even in relatively simple examples. Especially P-links connecting variables (to everything else) make the procedure extremely inefficient. In most natural examples taken from mathematics the axioms contain many variables, and only the negated theorem contains Skolem-constants which may be effectual in constraints. Since variables can be instantiated to any term, many of the possible combinations of operations involving axioms are compatible. In termini of graph structures: most subgraphs (combination of P-links) are compatible, only when the subgraphs are combined to the potential

final solution graph, is the incompatibility detected. But for subgraphs which are in fact compatible, no constraints exist to detect an incompatibility.

The observation that such constraint satisfaction methods, albeit of great potential in artificial intelligence and logic programming, are not particularly valuable tools for equational reasoning tasks, was called the "Anti-Waltz-Effect" and discussed in [Bläsius 86]. The main reason for the failure of this approach is the lack of strong constraints in this application domain, which consequently do not sufficiently reduce the search space.

As a consequence the equality procedure was modified several times, and the experimental modifications finally led to an Equality Graph Construction procedure (EGC-procedure) to be presented in the following paragraph, which constructs compatible graphs without ever creating the enormous set of incompatible ones in the first place. The essential idea is this: Starting from some initial state (some graph) a sequence of transformations is performed on each subsequent state until a compatible graph is constructed.

4.3 Equality Graphs

As it turned out the EGC-procedure incorporates and extends the ideas underlying Digricoli's RUE-resolution and Morris' E-resolution, however it is based on a different form of partial unification. Let us demonstrate the structure of equality graphs and their construction with the following example: Let E be a set of equations

$$E = \{ g(x\ x) = h(x\ b),\ h(u\ v) = h(v\ u),\ h(b\ a) = f(b),\ b = c,\ c = e \}$$

and suppose we want to resolve upon two literals, say $P(g(a\ y)z)$ and $\neg P(f(e)b)$. While the second argument of P and $\neg P$ can be unified directly the first argument presents a problem. Let us concentrate upon the first corresponding argument terms by denoting this as our given equality problem: $<g(a\ y) =_E f(e)>$.

The initial equality graph for this problem can be displayed as:

g(a y)

⋮

f(e)

where dotted lines indicate problems to be solved. The only information in this graph is that the problem $<g(a\ y) =_E f(e)>$ is yet unsolved. The main discrepancies are the different toplevel symbols g and f, hence this difference must be removed by some equations. Two equations in E can be combined to form a *chain*: $g(x\ x) = h(x\ b)$ --- $h(b\ a) = f(b)$, which can be used for the removal of this discrepancy and which is therefore inserted into the graph:

g(a y)

/

g(x x) = h(x b)

h(b a) = f(b)

f(e)
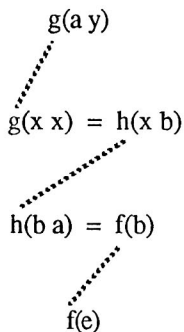
Now there are three subproblems to be solved:

$<g(a\ y) =_E g(x\ x)>, <h(x\ b) =_E h(b\ a)>$ and $< f(b) =_E f(e)>$.

In all three cases the heads of both terms are equal, but now the subterm may generate new subproblems, some of which are trivially solvable. We obtain the equality graph:

g(a y)

$\sigma = \begin{matrix} x \mapsto a \\ y \mapsto a \end{matrix}$ ——— * ———— $\sigma_1 = x \mapsto a$

———— $\sigma_2 = x \mapsto y$

g(x x) = h(x b)

———— $\tau_0 = x \mapsto b$

h(b a) = f(b)

f(e)
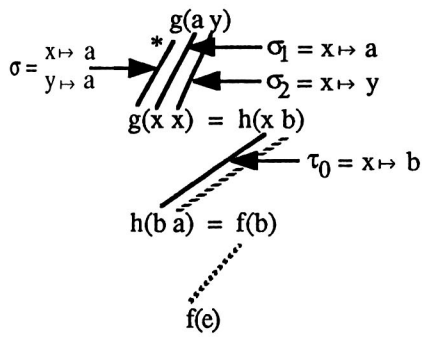
The solid links represent solved subproblems and are labelled with a substitution, empty substitutions are omitted. The substitutions σ_1 and σ_2 for the terms g(a y) and g(x x) must be checked for compatibility, i.e. they must be unifiable themselves. The result of this unification is represented as a separate link (*) labelled with the unifier $\sigma = \sigma_1 \cap \sigma_2$, where $\cap$ is the merge operation for substitutions (essentially the most general unifier of two substitutions).

Suppose we now select the unsolved subproblem b --- a: This subproblem is unsolvable, since we cannot build a chain of equations from the given set E to connect b and a. However instead of solving b --- a we can create a new subproblem at a higher term level: h(x b) --- h(b a). There exists an appropriate equation in E: h(u v) = h(v u) which is now inserted into the graph with the result:

g(a y)

$\sigma = \begin{matrix} x \mapsto a \\ y \mapsto a \end{matrix}$ ——— * ———— $\sigma_1 = x \mapsto a$

———— $\sigma_2 = x \mapsto y$

g(x x) = h(x b)

———— $\tau_1 = \begin{matrix} u \mapsto x \\ v \mapsto b \end{matrix}$

**

$\tau = \begin{matrix} x \mapsto a \\ u \mapsto a \\ v \mapsto b \end{matrix}$ ——— h(u v) = h(v u)

———— $\tau_2 = \begin{matrix} u \mapsto a \\ v \mapsto b \end{matrix}$

h(b a) = f(b)

f(e)
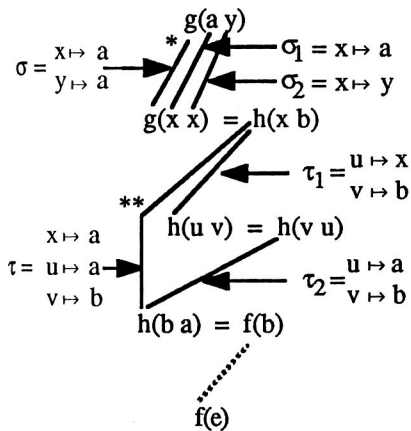
The substitution within each chain must be checked for compatibility and the result is again represented as a separate link. In our example the unit chain h(u v) = h(v u) connects the terms h(x b) and h(b a), the substitutions τ_1 and τ_2 are to be unified and the result is represented as the link (**), which is labelled with the unifier τ.

Finally the subproblem b --- e is selected and the chain b = c --- c = e can be inserted with the result:

g(a y)

$\sigma = \begin{matrix} x \mapsto a \\ y \mapsto a \end{matrix}$ ———— $\sigma_1 = x \mapsto a$

———— $\sigma_2 = x \mapsto y$

g(x x) = h(x b)

———— $\tau_1 = \begin{matrix} u \mapsto x \\ v \mapsto b \end{matrix}$

$\tau = \begin{matrix} x \mapsto a \\ u \mapsto a \\ v \mapsto b \end{matrix}$ ——— h(u v) = h(v u)

———— $\tau_2 = \begin{matrix} u \mapsto a \\ v \mapsto b \end{matrix}$

h(b a) = f(b)

$\varphi = \begin{matrix} x \mapsto a \\ u \mapsto a \\ v \mapsto b \\ y \mapsto a \end{matrix}$

b = c

c = e

f(e)
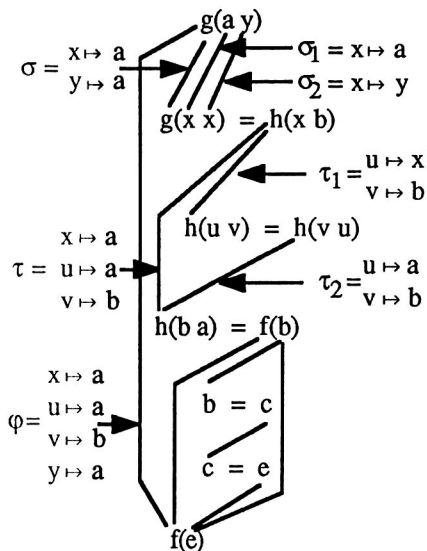
Since the substitutions σ and τ can be merged to $\varphi = \sigma \cap \tau$, the given equality problem is completely solved.

The transformation rules that transform a given partially solved graph (a *partial abstract equality plan*) into a successor graph with fewer unsolved subproblems are actually implemented as production rules, that fire when a particular situation in the graph is encountered; they constitute a *metareasoning calculus,* which can be shown to be sound and complete [Bläsius 87].

4.4 Experiments and Results

The EGC-procedure has not been integrated in the MKRP system, but a prototype implementation for handling pure equality problems with unit equations has been realized. We experimented with various control strategies and implementation techniques like sharing of common subgraphs etc. [Lotz 87].

Since the EGC-procedure is a general purpose inference system whose behaviour critically depends on control strategies, a final assessment largely depends on the state of the implementation and it is still too early for that (see [Bläsius & Siekmann 90] for the current discussion). So let us instead collect some impressions we have from running examples and developing heuristics.

The method is essentially a backward reasoning procedure. It allows to exploit syntactical and structural information about the problem to be solved, i.e. the terms to be made equal. On the other hand it has no built-in mechanism for exploiting structural information about the axioms. Equations can still be applied in both directions and in particular equations with variables on toplevel can be applied everywhere. Our experiments with the standard examples from group and ring theory where typical collapsing axioms like $x \cdot 1 = x$ occur have shown that the structural information exploited in the ordering of the equations reduces the search space much stronger. For this kind of examples - and these are usually the examples discussed in the literature - completion procedures are superior to the EGC-method. We have not yet been able to look at significant examples from other domains.

Nevertheless, the solution of really hard problems requires to exploit all information available to restrict the search space. We have found a good way to exploit information about the terms to be made equal. The next step will be to combine this source of information with information about the structure of the axioms, i.e. to combine ECG with completion techniques and theory unification algorithms. It is here where we expect significant results as the general representation provides a rich source of information. Furthermore the equality reasoning method has to be integrated into the resolution calculus. That means to find ways for handling conditional equations where the conditions may be arbitrary literals.

5 THE MARKGRAF KARL REFUTATION PROCEDURE

The actually implemented version of the Markgraf Karl Refutation Procedure [Karl Mark G. Raph 84] which was finished in 1984 is in the first place a tool for investigating the potential of the clause graph approach. However, in order to make it a useful and user friendly instrument for general ATP applications, it contains a number of additional service modules:
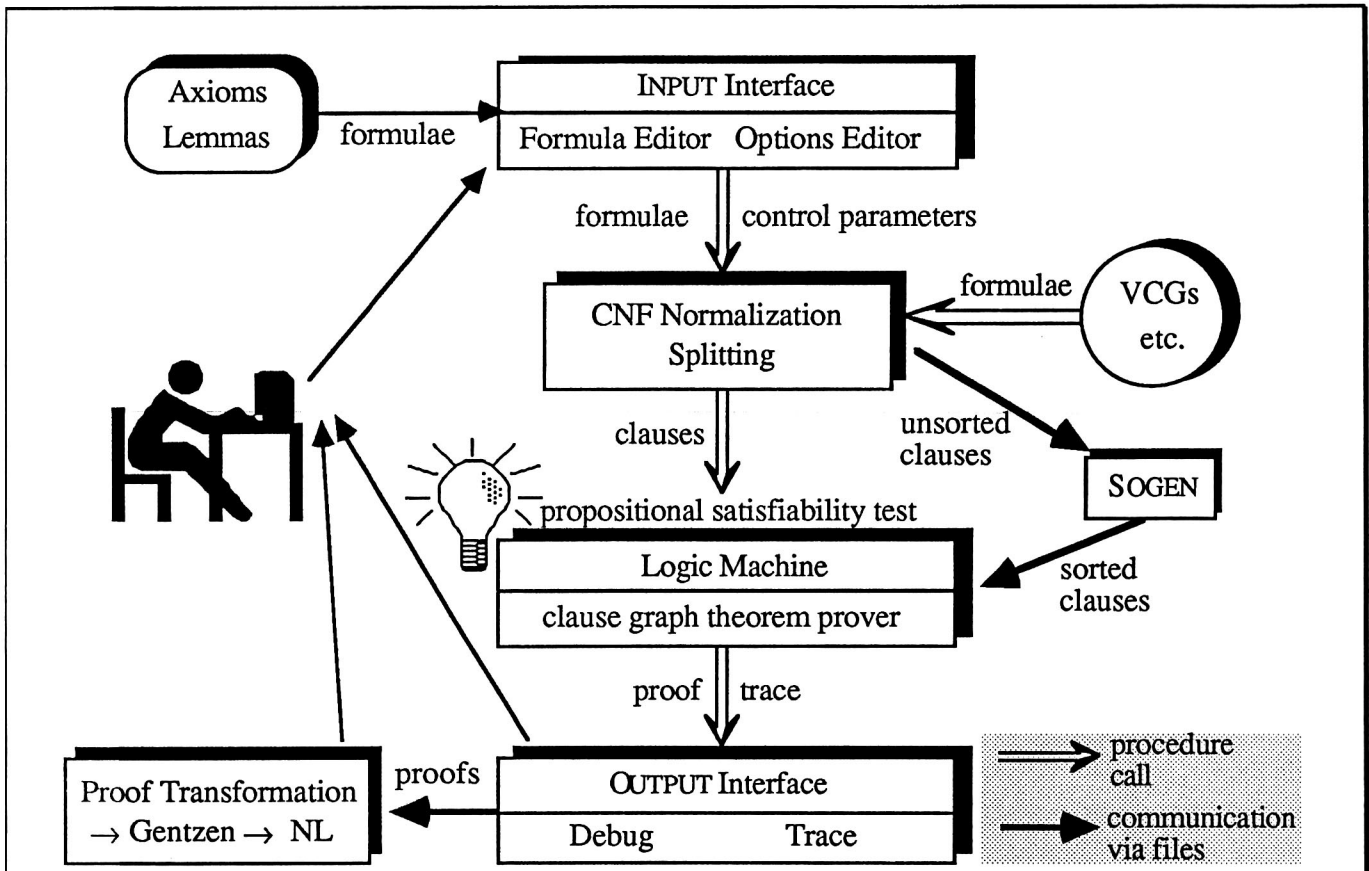
Figure 5-1: A Bird's Eye View of the MKRP System

The user communicates with the system via the INPUT and OUTPUT interface. The numerous input facilities essentially have the function to feed the system with logical formulae and control information in form of boolean and numerical parameters which influence the proof search (see the MKRP user manual [Beetz et al 88] for the many options that govern the search). Input to the system can of course also come from databanks with axioms and lemmas or from other programs like verification condition generators. The OUTPUT interface provides functions for tracing the behaviour of the system and for producing better readable resolution (and paramodulation) proofs. A separate proof transformer which has not been integrated into the MKRP system itself can read the resolution proofs, it constructs a refutation graph and uses it to generate a natural deduction proof [Lingenfelder 86, 89]. In a second step this natural deduction proof is translated into a better structured and more natural proof at a higher level of abstraction, stated at an intermediate language, with the goal to finally translate this via a third transformation into natural language [Huang 89].

The NORMALIZATION and SPLITTING module Skolemizes existential quantifications and transforms the formulae into conjunctive normal form (CNF) using an algorithm which is optimized for the elimination of equivalence signs [Eisinger & Weigele 83]. Disjunctions $\mathcal{F} \vee \mathcal{G}$ with no free variables in common, trigger case analysis. Technically this means splitting the clause set into two sets (and recursively into more), one containing $\mathcal{F}$ and the other one containing $\mathcal{G}$. Simple propositional simplifications like $\mathcal{F} \vee \mathcal{F} \rightarrow \mathcal{F}$ and others are also executed during the CNF normalization. After a propositional satisfiability check (where the

predicates´ arguments are ignored) with a Davis Putnam procedure, the generated clause sets are finally submitted to the logic machine.

Since all states of the transformation can be dumped on to files, other programs can manipulate these files before they are further processed by the MKRP. One such program is a prototype implementation of Manfred Schmidt-Schauss´ sort generation algorithm SOGEN [Schmidt-Schauss 88]. SOGEN takes unsorted or partly sorted clauses and transforms - if possible - unary predicates into sort relationships of the ΣRP* calculus. Automatic generation of sorts is helpful for proof problems generated by some application programs or when more sorts have to be introduced than the user is willing to define by himself. A sort structure representing the powerset of some finite set is an example where the number of sort symbols may explode.

Conversion of a unary predicate P into a sort S_P is only possible, when for every term $t = f(x_1,...,x_n)$ it is known whether $P(t)$ or $\neg P(t)$ holds. If for some term of this kind this information is not available at the beginning, but can be deduced, monitoring the proof search could enable further sort conversions. At the time being this is only possible by manually stopping the search, writing the current state on a file and invoking SOGEN. Automating this kind of metareasoning, however, is only a technical and a manpower problem.

5.1 The Architecture of the Logic Machine

The kernel of the MKRP system is a clause graph theorem prover. Since the design of the systems was developed before the theory resolution idea and the corresponding clause graph version came up, the actually implemented clause graphs admit only binary links (there is currently new implementation under way, that will eliminate this shortcoming).That means only finitary theory resolution algorithms and theories generated by two-literal clauses can be incorporated. We distinguish four main types of links, three of them are direct subtypes of binary I-links, the fourth type are P-links for supporting paramodulation. The three subtypes of I-links are R-links (Resolution links) connecting contradictory literals, S-links (Subsumption links) connecting a literal with an implied literal and T-links (Tautology links) connecting literals whose disjunction is tautologous. (For free literals, i.e. Robinson unifiable literals, R- and T-links are always parallel in the initial graph.) All links are labelled with sets of unifiers. Each link type is further divided into an external type connecting literals in different clauses and two internal types, one for connecting directly unifiable literals and the second one for connecting weakly unifiable literals.
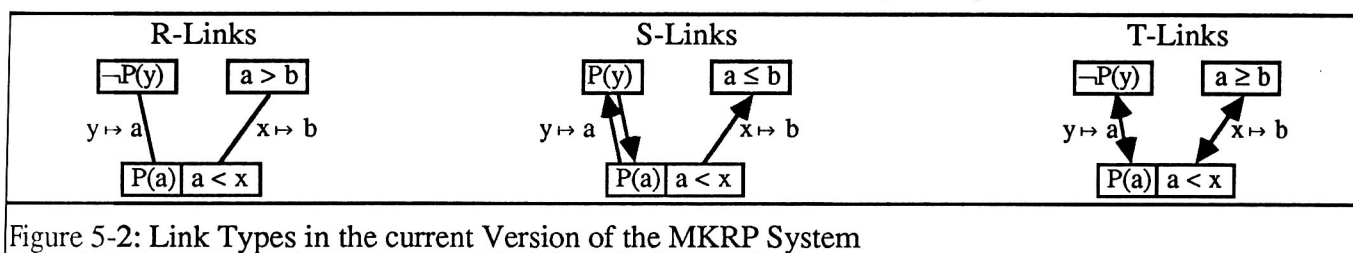
Figure 5-2: Link Types in the current Version of the MKRP System

Besides general datastructure modules for representing the elements of clause graphs and a unification module the logic machine contains 7 active modules:
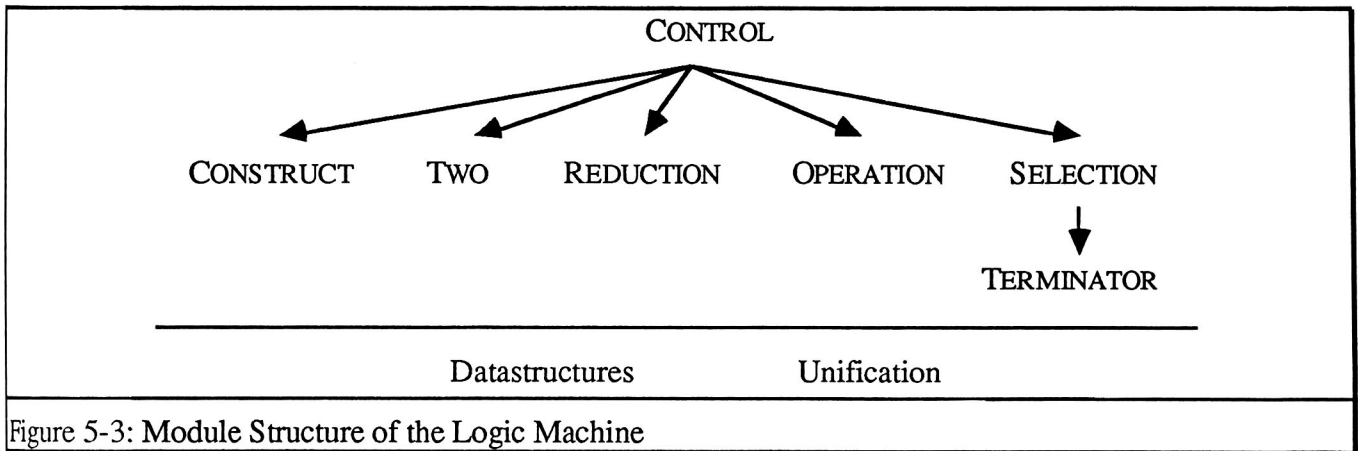
Figure 5-3: Module Structure of the Logic Machine

The CONSTRUCT module, as the name already suggests, constructs a clause graph from a set of clauses by inserting the clauses one by one into the graph and creating the links of the various types.

The TWO module realizes the link resolution operation (cf. chapter 3.3.1.4) for clauses with two literals by turning such clauses directly into link construction algorithms. That means we do not generate links to these clauses and then resolve them pairwise, but we generate the theory links directly.

The REDUCTION module performs most of the redundancy removing reductions mentioned in chapter 3.4, in particular purity removal, clause and link tautology, clause and link subsumption, link incompatibility checks, subsumption factoring, subsumption resolution and generalizing subsumption resolution.

The OPERATION module actually executes the resolution, factoring and paramodulation steps. It creates the new clause and inherits the links.

The SELECTION module selects the next deduction and reduction operation to be performed. It may call the TERMINATOR module for fast unit refutability checks.

The CONTROL module, as the name suggests, controls the interaction of all operations.

In the following sections we take a closer look at the key modules in the system.

5.1.1   The CONTROL Module

The CONTROL module controls the interaction of all operations. We distinguish two phases, the initialization phase where the graph is constructed, and the deduction phase where the search for the empty clause is performed.

The initialization phase:

    while new input clauses are available:
        select among those a clause with smallest number of literals
        if it is a non self-resolving clause and the corresponding option is set by the user
            then turn it into a link construction algorithm (call the TWO module)
        otherwise insert it into the graph (call the CONSTRUCT module) and
            reduce as long as possible (call the REDUCE module),
                but skip purity and link incompatibility reductions.
    After all clauses are inserted, do all possible purity and link incompatibility reductions.

Instead of first integrating all input clauses into a clause graph and then reducing this initial graph, we start from the empty graph and insert new clauses one by one, applying reduction rules after each insertion. This organization helps to keep the graph smaller: if a literal disappears, so do all its links; if it disappears before the insertion of later clauses, its links to these later clauses (which would afterwards disappear anyway) are not computed in the first place. The selection of the next input clause to be inserted is by increasing number of literals, because shorter clauses are more likely to subsume others. The link construction and initial reductions of a single clause are split into two subphases. First of all internal links are created, clause tautology and replacement factoring reductions are performed, and then the remaining links are constructed and the missing reductions follow. This avoids creating links for clauses or literals which are removed in the next step anyway.

The deduction phase:

> while empty clause not reached (and no other termination condition satisfied)
> select an applicable deduction operation (call the SELECTION module),
> create the corresponding clause (call the OPERATION module),
> select a corresponding reduction operation (call the SELECTION module again) and
> execute the reduction operation (call the REDUCTION module).

After creating a clause in the deduction phase, all reduction rules could in principle be applied automatically to the graph. It has, however, turned out to be more efficient to control the reduction operations explicitly by the selection module. The reason is that in some cases whole chains of operations are precomputed, for example in the TERMINATOR module, and have to be executed before any reduction should take place. Other strategies, in particular rewriting, require, if not performed destructively, the explicit deletion of clauses. Therefore the system can tell the REDUCTION module explicitly "delete this clause or link and follow potential snowball effects" or "apply all reduction rules to this particular clause" etc.

Besides the appearance of an empty clause, other termination criteria for the the deduction phase are resource limitations, collapse of the graph, the absence of a clause with only negative literals and a clause with only positive literals. The existence of a positive and a negative clause is a necessary condition for the unsatisfiability of the clause set.

5.1.2   The TWO Module

The TWO module turns two-literal clauses which are not self resolving, i.e. which have no internal R-links, into link construction algorithms for the corresponding predicates [Dixon 73, Ohlbach 83, Weigele 85]. A clause $\boxed{P(s) \mid Q(t)}$ for example is turned into an algorithm: "create R-links between literals $\neg P(s')$ and $\neg Q(t')$, create S-links between literals $\neg P(s')$ and $Q(t')$ as well as between literals $P(s')$ and $\neg Q(t')$ and create T-links between literals $P(s')$ and $Q(t')$, always if s,s' and t,t' are simultaneously unifiable". These clauses need not be inserted into the graph because their semantics is completely represented in the augmented link structure. All resolvents between these two-literal clauses, however, have to be created and either inserted into the graph or also turned into link construction algorithms. If there are infinitely many resolvents, they are created up to a certain level and the last level is inserted into the graph and not turned into link construction algorithms.

Since the theory links for the two-literal clauses are created already in the initialization phase when the graph is constructed, reduction operations may benefit from them even before the graph is completed. Carefully applied, i.e. applied only when there are not too many resolvents between these two-literal clauses themselves, this facility flattens the search space and therefore increases the power of the reduction algorithms and the look ahead potential of the link selection heuristics considerably. In other cases, however, it slows down the search by increasing the number of links in the initial graph exponentially.

5.1.3 The REDUCTION Module

The Reduction module performs most of the redundancy removing reductions mentioned in chapter 3.4 in particular purity removal, clause and link tautology, clause and link subsumption, link incompatibility checks, subsumption factoring, subsumption resolution and generalizing subsumption resolution [Präcklein 85, Eisinger et al 89]. What has not yet been implemented in the current version is the removal of redundant instances of clauses and links.

The interaction between the various reduction rules is quite complex. The application of one reduction rule may cause the applicability of another reduction rule that could not be applied before. It may, however, also destroy the applicability of another reduction rule that could be applied:
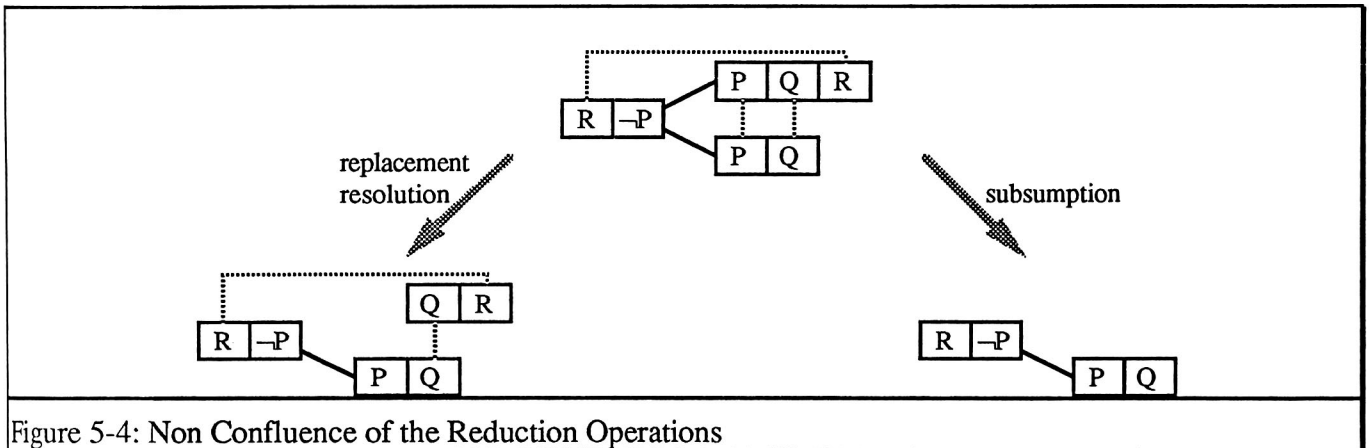
Figure 5-4: Non Confluence of the Reduction Operations

Assuming some further clauses that prevent purity but do not enable additional reductions, the application of either of the two competing reduction rules inhibits the other, and both of the resulting graphs are irreducible. The system defined by our reduction rules is Noetherian, but not confluent . That is, regardless of the order in which reduction rules are applied, an irreducible graph will always be reached after finitely many steps; however, different orders may result in different graphs.

So how do we select the next reduction step if several rules are applicable? In the example it would be better to select subsumption instead of replacement resolution, because that leads to a smaller irreducible graph. Unfortunately there are equally straightforward cases where it is just the other way around. There is no general criterion to find the smallest irreducible graph short of trying all sequences of rule application. But we have to keep in mind that reduction is a secondary operation taking place after every single deduction step. Its purpose is to make the graph smaller, and that's just what it does in any case. It would not seem wise to spend too much effort after each deduction step, just to avoid the possibility of suboptimal reductions.

Therefore we define a fixed precedence of the reduction rules based on a means-end-analysis that takes into account, for each rule, the cost for testing its applicability, the likelihood of its being indeed applicable, and the potential benefit from applying it [Präcklein 85]. Most of the estimates for these values were determined empirically (some of the factors are obvious, though: the benefit from removing an entire clause together with all its links is higher than the benefit from removing a single link.). The resulting precedence is as follows:

1.	merging of equal literals	6.	subsumption factoring
2.	purity removal	7.	subsumption resolution
3.	tautology removal	8.	removal of tautology links
4.	subsumption	9.	removal of incompatible links
5.	literal evaluation	10.	removal of subsumed links.

The whole reduction process is controlled by an agenda with entries of the form (rule, object/s) standing for the task "test whether the rule is applicable to the object/s, and if so, apply it". When a task is created, it is inserted into the agenda according to the precedence of the rule (in fact some more criteria influence the position of a task in the agenda; also, the precedence is not absolutely fixed but can be changed; see [Präcklein 85]). It is always the first task in the agenda that is executed. If the test succeeds, the corresponding rule is applied to the given object/s, and the current task creates new tasks which are inserted into the agenda before execution of the next task. The new tasks contain the information as to which rule might be applicable to which objects as a consequence of the current rule application. In order to determine these new tasks, a matrix is used that encodes the possible chain reactions and has the following (simplified) shape:

	Tautology	Link Tautology	Subsumption	Purity	Merging	Replacement Factoring	Replacement Resolution
Tautology				*			
Link Tautology				*			
Subsumption				*			
Purity				*			
Merging			◯	*			
Replacement Factoring			◯	*			
Replacement Resolution			◉	◯	⊛		◯

An entry in a cell of the table means that the application of the rule in the associated row may cause the applicability of the rule in the corresponding column. An asterisk says that the rule becomes applicable to objects that were adjacent or incident to the removed one, the circles denote the applicability to the changed objects themselves. If a rule becomes applicable due to replacement resolution, but only with generalization, this is indicated by shading the circle. The table can be refined to take into account that some rules like subsumption and replacement resolution are oriented: only one of the two objects involved is removed or changed. The matrix can also be extended to cover the various link conditions.

Figure 5-5: Potential Chain Reactions

Similar information is required to determine the tasks after a deduction step. Only newly inserted clauses and links can be tautologous or can be subsumed. The only possible replacement resolutions are those on the R-links connecting the new clause with the rest of the graph. The analogue holds for replacement factoring and SI-links. Any new subsumptions from the new clause back into the graph are indicated by new S-links.

This organization works sufficiently well and is rather flexible, as long as changes to the set of reduction rules are rare. The disadvantage of the scheme above is that each rule has to know about all other rules. If reduction rules are added or removed, one has to make sure that the matrix always reflects the actual rule set. Meanwhile we think that it might be better to trigger the creation of tasks not by rule applications, but by changes in the graph. For example, a purity can be caused by the removal of a link, regardless of the reason why this happened to the link. An object oriented approach seems to be most appropriate for such a new organization of the control.

5.1.4 The TERMINATOR Module

Todays logical calculi for mechanical deduction may be classified into those that start with a given set of logical formula and create new formula by the application of certain rules like resolution, paramodulation or by natural deduction rules until the theorem (forward reasoning) or a refutation (backward reasoning) has been derived.

More recently calculi of a different kind, like Andrews' matrix calculus [Andrews 68] or Bibel's connection method [Bibel 81] have been developed, which initially do not deduce any new formula, 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 some formulas. The clause graph proof procedure in its original formulation is of the first kind: A deduction is performed by the selection of a link, creating the corresponding resolvent, inserting it into the graph and deleting the selected link.

This proof procedure can be transformed into a calculus of the second kind using an idea originally proposed by S.Sickel [Sickel 76]: 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".

The method realized in the TERMINATOR module is very much in that spirit, but used for a 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 parent a unit, the clause graph has to contain a refutation tree or *terminator situation* (i.e. a special subtree), which just represents this chain of unit resolutions.

a) one level situation   b) two level situatiton   c) three level situation

Figure 5-6: "Terminator" Situations
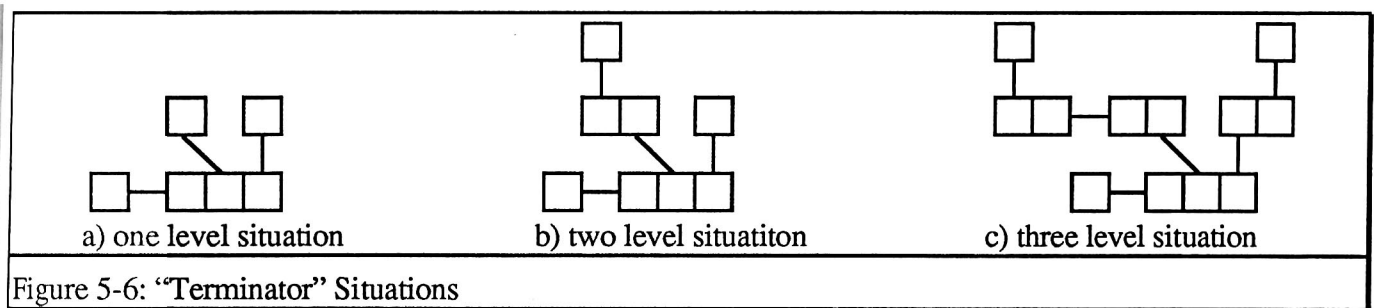
Every box in figure 5-6 represents a literal, a string of boxes is a clause and complementary unifiable boxes (literals) are connected by a link. If all unifiers attached to the link in figure 5-6,a are compatible, this represents a one-level terminator situation, since it immediately allows for the derivation of the empty clause. Similarly figure 5-6,b represents a two-level terminator situation (since its kernel is a one-level terminator situation). Figure 5-6,c is a three-level terminator situation, since its kernel is a two-level terminator.

Now every unit refutable clause set is characterized by this topological structure embedded into the corresponding clause graph. It is a tree with unit clauses as leaf nodes and the other nodes being non unit clauses. Links 1,2 and 4 in figure 5-7,a represent such a *refutation tree*. Three successive resolutions with these links (or rather their descendants by inheritance) will generate the empty clause. Links 1,3 and 4 do not constitute a refutation tree because the unifiers are not compatible. (Link 1 and 4 require the variable x to be instantiated with 'a' whereas link 3 instantiates x with 'b'.) Thus, the compatibility test for the unifiers has to be an essential part of the refutation tree extraction algorithm. A more complicated situation is shown in figure 5-7,b, which is particularly hard to detect, as it is usually embedded into the rest of the graph.
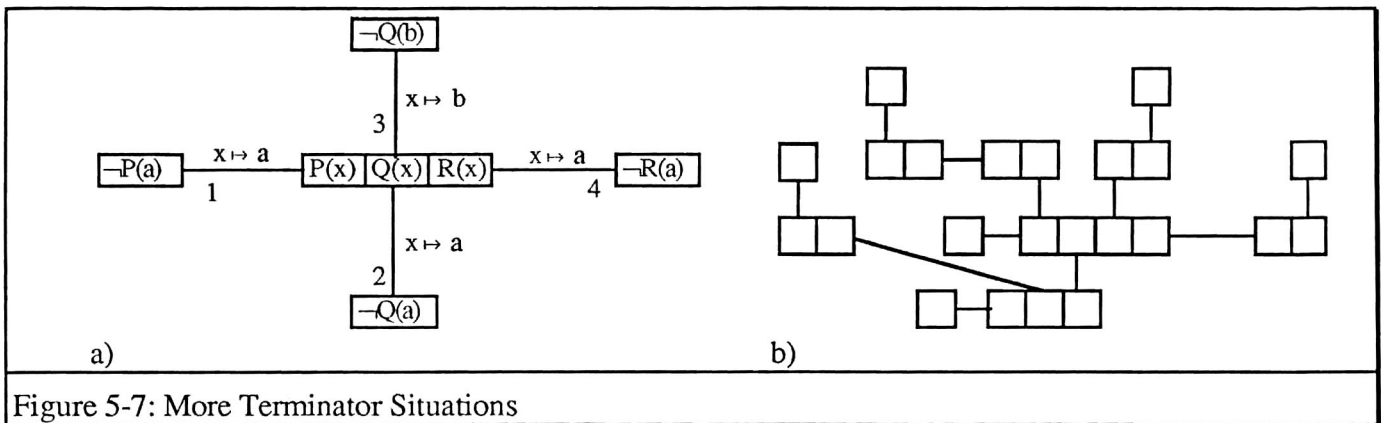
Figure 5-7: More Terminator Situations

The following important result is known about refutation trees [Harrison & Rubin 78]: A unit refutable clause set is unsatisfiable if there exists a refutation tree for its factored set.

As every successful search for the empty clause must reach a point where the remaining problem is unit refutable, (at least the very last step must be a resolution step between two unit clauses.), we know the TERMINATOR will eventually succeed.

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) had to be developed.

The first attempt to implement an N-level terminator algorithm used a recursive technique [Bläsius et al 81]: 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 units to resolve away all literals except the one connected to C. Now the one-level test for C was performed once again taking advantage of this new information.

The enormously positive effect of this algorithm came somewhat as a surprise and considerably enhanced the strength of the system, but unfortunately the algorithm itself was rather inefficient, because no result of former search steps calls could be reused. Therefore the same part of the clause graph had to be examined again and again and only the N = 2 case was ever within practical limits.

Whereas the old algorithm worked from inside the refutation tree to the leaf nodes (unit clauses), our current algorithm works just in the opposite direction and stores every information it once generated for later use. In particular it exploits some special properties of the unifiers attached to the links and is now capable to examine millions of resolution steps in a reasonable amount of time.

The TERMINATOR component of the MKRP-system is used in two ways: first it acts like a simple and fast theorem prover and is activated for clause sets which are known to be unit refutable. Horn clause sets and Horn renamable sets [Lewis 78] are typical examples. Secondly it is used to overcome the problem that the selection functions have the very limited horizon of a few steps ahead, since the computation af a further n-level look ahead for n ≥ 2, is prohibitively expensive and often outweighs the advantage. The TERMINATOR acts as a bottom-up techniques, which checks at tolerable cost if there is a proof within a predefined complexity bound. This use is the no-loop-requirement of [Sickel 76] and is akin to the n-level-look-ahead heuristic proposed by [Kowalski 75] and really constitutes a bidirectional search: the normal operations on the graph essentially constitute a topdown search, whereas the TERMINATOR is essentially a bottom-up search routine.

The implementation of the terminator algorithm has considerably increased the performance of the MKRP-system. Some very difficult examples other systems have not been able to solve until now or have serious difficulties which could be solved by our system because of this interaction between the top-down

and bottom-up approaches [Ohlbach 82], [Ohlbach&Wrightson 84]. A more detailed description of the TERMINATOR can be found in [Antoniou&Ohlbach 83].

5.1.5 The SELECTION Module

The real "intelligence" of an automated theorem prover is in its control facility for determining the next deduction step. Pure resolution theorem provers usually do brute force search within the boundaries of certain strategies, or filters, like set-of-support, unit-resolution, linear resolution etc. The search may further be ordered by preference operations according to some complexity measures like the size of terms and clauses. In the clause graph environment traditional filters can very easily be realized by declaring links active or passive and blocking operations on passive links. By an appropriate setting of the active-passive switch every traditional filter can be simulated. Seen from this point of view a traditional filter 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 filter is complete, the shaded parts are at least theoretically irrelevant.

On the other hand it is a well known 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 initial setting of the filter. Only if nothing better is known does it take the filter information into account and acts like a traditional theorem prover.

In contrast to the global search strategies and global heuristics some heuristic selection criteria are based on local structural information about the graph and the resolvent (paramodulant) respectively. Initially we experimented with about 20 different heuristic features, where every feature attaches a certain value to every link in the graph. 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 and there are the 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. Most of the information of the other features however was expressed as a real number in [0,1], where we experimented with several linear and nonlinear metrics [Bläsius et al 81]. 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 had been designed such that heuristic features could 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
   - number of added clauses
   - number of added links
   - number of removed predicate symbols

2. Complexity of the deduced clause
   - depth of the clause in the search space
   - number of literals in the clause
   - term complexity of the clause
   - degree of isolation of the clause,
       i.e. chance to produce a pure literal after further link deletions

3. Complexity of the parent clauses
   - degree of isolation of the parent clauses,
       i.e. chance to produce a pure literal after further link deletions.

The weights of the polynomial could be determined by the user. Although there was a default setting which was sufficient for simple examples, the user had to experiment with these parameters quite a lot in order to solve more complex problems.

Heuristic assessment of links with weighted polynomials seemed to be very natural for the clause graph procedure. Eventually, however, it turned out that its disadvantages outweigh its advantages. The disadvantages are:

- The computation of the polynomials is very expensive. Since some of the features depend on the local structure of the graph, i.e. on the other links connected to the same clauses as the given link, its heuristic value had to be recomputed each time a new link is attached to the clause. That means that in general after each step all active links have to be assessed again.

- The isolation features, i.e. resolvent isolation and parent isolation, which are intended to provoke purity deletions are significant only for graphs with many different predicates and few links per clause. But these examples are in general trivial anyway. The complex examples, however, are those with dense graphs, i.e. many links per clause; and in this case purity rarely occurs.

- The proper choice of the weights was very difficult for an unexperienced user. If you need 10 trials until you have found a successful parameter setting, you can´t really say that the proof was found automatically.

In order to overcome these difficulties we finally came up with a new control structure which does not need any user intervention at all and shows an even better performance than the old heuristics. This control structure was implemented in the SELECTION module. It is realized as a production system. The conditions of the production rules are characteristic patterns in the graph and the actions are sequences of resolution, factoring and paramodulations followed by applications of reduction operations to particular parts of the graph. The production rules are applied in a fixed order, but the action part of each rule may contain applications of other rules.

We describe the production rules in the order they are applied.

UNIT-CONFLICT
    Condition: There are two unit clauses connected by an R-link.
    Actions: Create the empty clause and stop.

TERMINATOR-1
    Condition: There is a one-level terminator situation, i.e. there is a non-unit clause whose literals are connected to unit clauses via R-links with compatible unifiers.
    Actions: Execute the sequence of unit resolutions and stop.

FACTORIZE
    Condition: The factoring option is set by the user and there is an active SI-link.
    Actions: Create a factor and reduce it immediately.

REDUCTIONS
    Condition: One of the following conditions holds for an R-link:
        - both parents become pure
        - the resolvent subsumes the parent clause
        - one parent clause becomes pure and the resolvent is shorter than this parent clause.
        - the resolvent becomes a unit clause by merging.
    Actions: Create the resolvent and reduce it immediately

ELIMINATE
Condition: An equation $f(x_1,\ldots x_n) = t$ where f does not occur in t has been deduced
Actions: Eliminate all occurrences of f by successive paramodulations.
Reduce all modified clauses afterwards.

MERGING
Condition: A resolvent becomes smaller than its smallest parent clause.
Actions: Create and reduce the resolvent.

TERMINATOR
Condition: There is an n-level terminator situation (n is determined by the user).
Actions: Execute the sequence of unit resolutions and stop.

TERMINATOR-UNITS
Condition: The corresponding option is set by the user.
Actions: Create all unit clauses the TERMINATOR module has found during its search for an n-level terminator situation.
Reduce all created unit clauses.

STRATEGIES
Condition: None
Actions: Select an active R- or P-link with minimum heuristic value according to the following formula:
$$2 * \text{depth\_in\_the\_search\_space} + 3 * \text{length\_of\_resolvent/paramodulant}$$

The following basic strategies are available:
BASIC   All R-links are activated
SET-OF-SUPPORT All R-links connected to clauses in the set of support are activated.
LINEAR   All R-links connected to the latest generated clause are activated.
P-links are activated according to the same rules as for R-links. Further restrictions can be imposed on P-link selection by the following paramodulation strategies:
COMPLETION   Knuth Bendix completion is simulated
ONCE ONLY   Each unit equation paramodulates only once into another unit equation.

There is a very important feature hidden in the notion "length of the resolvent/paramodulant". Since clauses may become shorter by merging of literals, replacement factoring and replacement resolution, it is almost impossible to anticipate the length of a new clause. A clause C, for example, may be reduced by itself, it may cause the reduction of a clause C_1 which may trigger reduction of a clause C_2 which may cause reduction of C again, i.e. there may be snowball effects and cycles of literal deletions in the graph. For the function that computes for a link the length of its resolvent or paramodulant it would be too expensive to follow all of these snowball effects. A compromise which works quite well is to estimate the size by considering only literal merging, replacement factoring and replacement resolution with a single unit clause.

Since the conditions are in general quite complicated and costly to test, it is not possible to check all conditions at once and then select among the applicable production rules. Only for very few examples this turned out to be a disadvantage, but it never caused complete failure. It should further be noted that a monitoring mechanism allows to check the conditions of each rule only for the part of the graph which has been changed since its last invocation.

6 EXPERIMENTS AND RESULTS

Over the many years we worked with the MKRP system we experimented with almost every example published in the literature as well as with problems generated by verification condition generators for PASCAL and FORTRAN. Furthermore we made three systematic studies of the performance of the system. The first one was to compare our system with the eight different proof procedures, Minker and Wilson have investigated in a comparative study of the the University of Maryland [Minker & Wilson 76]. 66 of the 87 examples were proved with the MKRP by unexperienced maths students with very little help from the experts [Bürckert et al 83].

The MKRP system was also used to prove the theorems of some textbooks. As an assignment the book by R. Smullyan "What's the name of this book?" [Smullyan 78], which contains a fair selection of logical puzzles, was given to a group of students. Within one term they were able to code and prove the whole book (except for a few cases that were not really intended for a deductive solution.)

The main testbed, however, was a textbook on automata theory: P. Deussen "Halbgruppen und Automaten" [Deussen 71], which gives a nice algebraic treatment of automata theory. The book is divided into three sections and we were able to prove every theorem in the first section (a little more than a third of the book).

From the experiences with these examples, which come from quite different areas, we think we are able to assess

- predicate logic as a language for representing mathematical and logical knowledge,

- the resolution calculus as an inference mechanism

- the clause graph approach

- our way for manipulating clause graphs and guiding the search.

6.1 Assessment of Predicate Logic for Representing Mathematical Knowledge.

Except for equality, predicate logic generally has no built-in concepts and no structuring mechanism. That means for example in order to prove $3+3 = 6$ one has to axiomatize natural numbers with the Peano axioms, axiomatize addition and apply a number of paramodulation steps to transform the term $+(s(s(s(0))), s(s(s(0))))$ into $s(s(s(s(s(s(0))))))$. The argument that frequently occurring axioms and lemmata like those for numbers or set theory could be kept in a library and automatically loaded when necessary is not really to the point. The real problem is that with this method all problems, trivial problems like adding two numbers as well as the problem to find the key idea for proving the theorem at hand, are on the same level. And that means for the designer of a theorem prover the impossible task to find a uniform mechanism which solves problems of all degrees of difficulties with the same efficiency. It is, however, hardly possible to find a single strategy or heuristic which guides a theorem prover in, say, proving Fermat's last theorem with the same certainty as adding two numbers.

To overcome these problems, we think two measures are necessary. First of all, frequently occurring concepts like numbers, lists, sets, but also algebraic properties like associativity, commutativity, symmetry etc. have to be built into the predicate logic and the resolution calculus in order to free the global search control from the simple algorithmically solvable problems. Sorted logics as a representation of simple set relationships and theory unification as an operationalization of algebraic properties of function and predicate symbols are a first step. Sorts helped in encoding logical puzzles from Smullyan's book [Ohlbach & Schauss 87], but for real world examples with more complex statements about sets, sorts are too weak. Built-in arithmetic is absolutely necessary for solving any nontrivial verification condition, but of course also for algebraic problems, where numbers are used for example in indices and the like.

The second measure is to develop a better input language which allows to represent statements at a much higher level of abstraction as well as heuristics for choosing the right axiomatization for the problem at hand and for breaking down complex problems into smaller ones. For making the MKRP system prove the theorems in the automata book, for example, we had to experiment with various versions of axiomatizations and in particular we had to break down manually even relatively simple theorems into several subproblems. The rules for preparing the problems for submitting it to the theorem prover were sometimes purely syntactical, usually domain dependent and quite often model oriented.

Because of these difficulties we stopped the experiment with the automata book and started the development of more appropriate tools. As a first step a mathematical knowledge representation language was developed and the concepts, definitions and the general knowledge that is required to prove the theorems in R. Artin "Galois theory" [Artin 42] were represented in this language to test its adequacy. The final evaluation of this new line of enterprise however may have to wait for some time to come, as we need more experience in how to use this mathematical knowledge in guiding the search.

6.2 Assessment of the Resolution Calculus as an Inference Mechanism

There are at least two objections against the resolution calculus: it needs the unnatural and potentially explosive conjunctive normal form; and it is a purely syntactical rule and therefore too lowlevel for reasoning about complex mathematical concepts.

To avoid the potential explosion of literals during the transformation into clause form resolution calculi for arbitrary formulae have been developed [Murray & Rosenthal 87]. We tend to think that this is in general not worth the effort (there may be special situations, where it is important), because

- there is a linear transformation algorithm which avoids the duplication of literals by introducing artificial predicates as abbreviations for complex formulae,

- reduction operations, in particular tautology and subsumption deletions are absolutely essential for solving any nontrivial problem. In particular subsumption algorithms are already quite expensive for clauses [Gottlob & Leitsch 85]. They are almost impossible for arbitrarily structured formulae.

- The real hard problems do not come from the duplication of literals, but from the distribution of terms in the clauses. There are clause sets with 5 literals in total (typically in the implication calculus), requiring hundreds of resolvents in the proof. The additional mechanisms in nonclausal resolution methods do not help at all to solve these problems.

For the removal of redundancies introduced by duplication of literals, the reduction operations we integrated in the MKRP system turned out to be quite effective. A remarkable case of this kind is the "challenge problem" suggested by Peter Andrews during the fourth workshop on automated deduction in 1979. The problem is to show the validity of the formula

$$([\exists x \forall y (P(x) \Leftrightarrow P(y))] \quad \Leftrightarrow \quad [(\exists x \, Qx) \Leftrightarrow (\forall y \, P(y))])$$

$$\Leftrightarrow$$

$$([\exists x \forall y (Q(x) \Leftrightarrow Q(y))] \quad \Leftrightarrow \quad [(\exists x \, P(x)) \Leftrightarrow (\forall y \, Q(y))])$$

with a resolution system. Because of the deep nesting of equivalences this formula results in a rather sizable clause set containing 128 eight-literal clauses (actually a straightforward conversion is more likely to produce 1024 clauses; Markgraph Karl's conversion algorithm avoids certain redundancies in handling equivalences). The total clause graph for these 128 clauses contains about 30,000 links, which does represent a challenge for any selection component. When we applied the reduction rules right after each insertion of a clause during the initialization loop, the actual graph never exceeded a size of 20 clauses and

1153 links. After insertion of the 125th input clause the empty clause was derived, and the deduction loop was not even activated.

The objection against the syntactical nature of resolution can partly be invalidated by using theory resolution, i.e. incorporating the semantics of special symbols into theory unification and special purpose resolution algorithms. But still, general inference rules like resolution or even theory resolution are too lowlevel for solving really hard problems. Advanced methods for structuring mathematical knowledge and manipulating it on a more abstract level as well as model based reasoning are certainly necessary.

6.3 Assessment of the Clause Graph Appraoch

The clause graph datastructure contains valuable explicit information about the structure of the problem to be solved. The various reduction algorithms and selection heuristics can use this information for removing redundancies and guiding the search, most of them would be unfeasible without having access to this extra information.

Maintaining the links however is very expensive, for many examples too expensive. We were never able to keep more than about 200 clauses and about 20000 links at a time. That means clause sets with little internal structure, dense graphs and with few obvious redundancies are not very suitable for the treatment with clause graph resolution. Their search space is unstructured and has to be explored by generating sometimes thousands or even millions of clauses. This is impossible with clause graph resolution.

The literals in the resolvents and the inherited links are essentially instances of the original ones and therefore their explicit generation introduces new redundancies which may become predominant to the extra information they contain. In particular our experience with the TERMINATOR algorithm which does not generate any resolvents explicitly, but works on the initial graph, has shown that this kind of operation on clause graphs can speed up the search by many orders of magnitudes compared to cg-resolution with explicit generation of resolvents.

Therefore our conclusion is: the clause graph datastructure with R- and S-links is an excellent basis for all kinds of algorithms in the theorem prover. The search for the refutation, however, should avoid the explicit generation of resolvents as far as possible. (Parallelizing clause graph resolution would not be of much help either if it only speeds up the process of filling the memory with clauses and links.)

Paramodulated clause graphs with P-links for equality handling on the other hand turned out to be a blind alley. There are simply too many P-links and the heuristics based on P-links are much too shortsighted to solve any significant equality problem. It is too early for a final evaluation of our current developments for equality reasoning.

6.4 Assessment of our Way to Manipulate Clause Graphs

Tests with hundreds of examples have shown the following overall behaviour: The reduction operations are powerful enough to reduce the given problem very quickly to the hard kernel. Many examples (including all proposition formula sets) can therefore already be solved during the initial reduction phase. If the remaining kernel problem is unit refutable it is mostly solved by the TERMINATOR module, if not, there is a class of examples where a pulsation of the graph can often be observed that is caused by some relatively blind resolution steps, followed by a sequence of reductions after the "right" clause has been created. It is quite possible, but scarcely necessary, to make hundreds of deductions without too much growth of the graph. Although each step is relatively expensive and hence the LIPS rating (logical inferences per second) is rather low, the system penetrates deeply into the search space because large parts are pruned very early. For this class of examples we can say that the system really behaves like a human mathematician. It finds quite straightforwardly a successful path through the search space, without generating millions of alternatives.

The following table which compares the behaviour of the MKRP system with the results of the Minker and Wilson study confirms the goal directed behaviour of the system.

	Maryland Refutation Procedures			MKRP		
	NOC-P	NOC-G	NOC-P/NOC-G	NOC-P	NOC-G	NOC-P/NOC-G
ANCES	19 18	62 943	.306 .019	7	7	1
BW 3	19 21	63 2585	.302 .008	8	16	.5
Prim	21 20	89 221	.236 .09	12	27	.44
Wos 3	7 7	17 154	.412 .045	3	3	1
Wos 7	13 12	241 244	.054 .049	9	10	.9
Wos 8	12 12	210 360	.057 .033	6	9	.67
LS-17	20 14	98 1203	.204 .011	4	7	.57
LS-21	12 12	252 684	.048 .018	7	12	.58
LS-35	14 14	335 1521	.042 .009	6	9	.67
LS-65	17 17	48 880	.354 .019	11	58	.19
LS-115	13 13	20 227	.65 .057	7	11	.63
LS-121	31	536	.058	12	30	.4

NOC-P = Number of clauses in the proof
NOC-G = Number of clauses generated

The table is to be understood as follows: the first column gives the name of the set of axioms in [Minker & Wilson 76]. The next three columns quote the findings of Minker and Wilson, 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 of the proof procedure that performed best. The final three columns give the corresponding values of the MKRP system. To give some more findings for more difficult examples: Argonne's new OTTER theorem prover generates 5041 clauses for proving Sam's lemma (using UR-resolution) whereas the MKRP system generates only about 190 unit clauses. The proof needs 19 UR-steps.

These numbers are the statistical corroboration of the first two claims put forward in the introduction: it is indeed possible to build a theorem proving system, that it will display an active and directed behaviour and it will not generate a search space of many thousands of irrelevant clauses, but will find a proof with comparatively few redundant derivation steps.

There is however another class of examples, where the graph explodes after a few steps and there are no reductions which cause shrinking of the graph to a manageable size. For this class, as already mentioned earlier, the explicit generation of resolvents and links is a serious disadvantage.

To Summarize our Conclusions:

A "next generation theorem prover" has to consist of two levels. The lowlevel part should be based on a logic with as many built-in concepts as possible. The calculus should be theory resolution, but with different cooperating theories. This is a very general scheme which covers all kinds of special purpose algorithms. Clause graphs (with theory links of course) are the right datastructure, but explicit generation of resolvents should be avoided.

The highlevel part has to represent mathematical knowledge, factual as well as heuristical, in a more abstract and domain dependent way. It should be able to break complex problems down into smaller pieces which can be handled by the lowlevel part. In order to eventually achieve the performance of a human mathematician, we think that the only way, and therefore the greatest challenge, is the development of model based guidance for the theorem prover.

ACKNOWLEDGEMENTS

References and Bibliography of the MKRP Project

Anderson 70	Anderson, R., *Completeness Results for E-Resolution.* Proc Spring Joint Conf., pp. 653-656, 1970.
Andrews 68	Andrews, P., *Resolution with Merging.* JACM, Vol 15, No 3, 1968.
Andrews 81	Andrews, P.B., *Theorem Proving via General Matings.* JACM 28:2, pp. 193-214, 1981.
Antoniou&Ohlbach 81*	Antoniou, G., Ohlbach, H.J., *Terminator.* Proc. of 8th IJCAI, Karlsruhe 1983.
Artin 42	Artin, E., *Galois Theory.* Univ. of Notre Dame Press, Notre Dame, London, 1942.
Beetz et al 88	Beetz, M., Freitag, H., Klug, J., Lingenfelder, C., *The MKRP User Manual.* SEKI Working Paper SWP-88-01.
Bibel 81	Bibel, W., *On Matrices with Connections.* JACM 28:4, pp. 633-645, 1981.
Biundo et al 86	Biundo, S., Hummel, B., Hutter, D., Walther, C., *The Karlsruhe Induction Theorem Proving System.* Proc. of 8th CADE, Springer LNCS 230, 1986.
Bläsius et al 81*	Bläsius, K., Eisinger ,N., Siekmann, J., Smolka, G., Herold, A., Walther, C., *The Markgraph Karl Refutation Procedure.* Proc. of IJCAI-81, Vancouver, 1981.
Bläsius 83*	Bläsius, K.H., *Equality-Reasoning in clause graphs.* Proc. of IJCAI-83, Karlsruhe, pp. 936-939, 1983.
Bläsius 86*	Bläsius, K.H., *Against the 'Anti Waltz' Effect in Equality Reasoning.* Proc. of German Workshop on Artificial Intelligence, Springer Informatik-Fachberichte 124, pp. 230-241, 1986.
Bläsius 86a*	Bläsius, K.H., *Construction of equality graphs.* SEKI-REPORT SR-86-01, FB Informatik, Univ. Kaiserslautern, 1986.
Bläsius 87*	Bläsius, K.H., *Equality Reasoning Bases on Graphs.* Dissertation, Fachbereich Informatik, Univ. Kaiserslautern, 1987.
Bläsius&Bürckert 89*	Bläsius, K.H., Bürckert, H.-J., *Deduction Systems in Artificial Intellicence.* Ellis Horwood Series in AI, 1989.
Bläsius&Siekmann 88*	Bläsius, K.H., Siekmann, J., *Partial Unification for Graph Based Equational Reasoning.* Proc. of 9th CADE, Springer LNCS 310,pp. 397-414, 1988.
Bläsius&Siekmann 90*	Bläsius, K.H., Siekmann, J., *Equality Reasoning Based on Equality Graphs.* forthcoming.
Boyer&Moore 79	Boyer, R.S., Moore, J.S., *A Computational Logic.* Academic Press, 1979.

Brand 75	Brand, D., *Proving Theorems with the Modification Method.* SIAM Journal of Comp., Vol 4, No 4, 1975.
Bruynooghe 75	Bruynooghe, M., *The Inheritance of Links in a Connection Graph.* Report CW 2 Applied Mathematics and Programming Division, Katholieke Universiteit Leuven, 1975.
Buchberger 87	Buchberger, B., *History and Basic Features of the Critical Pair/Completion Procedure.* Journal of Symbolic Computation, Vol 3, Nos 1&2, pp 3-38, 1987.
Bürckert et al 83*	Bürckert, H.-J., Wang, H., Zheng, R., *MKRP: A Performance Test by Working Mathematicians.* Interner Bericht 19/83, Fak. f. Informatik, Univ. of Karlsruhe, 1983.
Bürckert&Nutt	Bürckert, H.-J., Nutt, W., *Unif'89: Extended Abstracts of the third International Workshop on Unification Theory.* SEKI Report , FB. Informatik, Univ. of Kaiserslautern, 1989.
Bundy 83	Bundy, A., *The Computer Modelling of Mathematical Reasoning.* Academic Press, London 1983.
Chang&Lee 73	Chang, C.-L., Lee, R.C.-T., *Symbolic Logic and Mechanical Theorem Proving.* Computer Science and Applied Mathematics Series (Editor Werner Rheinboldt), Academic Press, New York, 1973.
Cohn 87	Cohn, A., *A More Expressive Formulation of Many Sorted Logic.* Journal of Automated Reasoning, vol. 3, No. 2, pp. 113-200, 1987.
Davis&Putnam 60	Davis, M., Putnam, H., *A computing procedure for quantification theory.* J.ACM 7, pp. 201-215, July 1960.
Dershowitz 89	Dershowitz, N. (ed.), *Rewriting Techniques and Applications.* Proc. of 3rd Int. Conference on Rewriting Techniques, RTA-89, Springer LNCS 355, 1989.
Deussen 71	Deussen, P., *Halbgruppen und Automaten.* Springer Verlag, 1971.
Digricoli 79	Digricoli, V.J., *Resolution by Unification and Equality.* Proc. 4th Workshop on Automated Deduction, Texas, 1979.
Digricoli 81	Digricoli, V.J., *The Efficacy of RUE Resolution, Experimental Results and Heuristic Theory.* Proc of IJCAI-81, Vancouver, 1981.
Digricoli 85	Digricoli, V.J., *The Management of Heuristic Search in Boolean Experiments with RUE Resolution.* Proc. of IJCAI-85, Los Angeles, 1985.
Dixon 73	Dixon, J.K., *Z-Resolution: Theorem Proving with Compiled Axioms.* J.ACM, 20,1, 1973.
Eisinger 81*	Eisinger, N., *Subsumption and Connection Graphs.* Proc. of IJCAI-81, pp. 480-486, Vancouver, 1981.
Eisinger 86*	Eisinger, N.,*What You always Wanted to Know about Clause Graph Resolution.* Proc. of 8th CADE, Springer LNCS, Vol. 230, pp. 316-336, 1986.
Eisinger 88*	Eisinger, N., *Completeness, Confluence, and Related Properties of Clause Graph Resolution.* SEKI Report SR-88-07, FB Informatik, Univ. Kaiserslautern, 1988.
Eisinger&Weigele 83*	Eisinger, N., Weigele, M., *A Technical Note on Splitting and Clausal Form Algorithms.* Proc. of GWAI-83, Springer Fachberichte, 1983.
Eisinger&Ohlbach 86*	Eisinger, N., Ohlbach, H.J., *The Markgraf Karl Refutation Procedure (MKRP).* Proc. 8th CADE, Springer LNCS, Vol. 230, pp. 681-682, 1986.
Eisinger&Ohlbach 87*	Eisinger, N., Ohlbach, H.J., *Deduktionssysteme-Grundlagen und Beispiele.* in Deduktionssysteme: Automatisierung des logischen Denkens, K.H. Bläsius/ H.-J. Bürckert (Hrsg.), Oldenbourg Verlag, München, pp. 22-833, 1987. English Translation: Deduction Systems for Artificial Intelligence, K.H. Bläsius/H.-J. Bürckert (editors), Ellis Horwood Ltd., Chichester, 1989.
Eisinger et al 89*	Eisinger, N., Ohlbach, H.J., Präcklein, A., *Elimination of Redundancies in Clause Sets and Clause Graphs.* SEKI Report SR-89-10, FB. Informatik, Univ. of Kaiserslautern.
Gottlob&Leitsch 85	Gottlob, G., Leitsch, A.,*On the Efficiency of Subsumption Algorithms.* JACM, Vol. 32, No. 2, pp. 280-295, 1985.

Gottlob&Leitsch 85   Gottlob, G., Leitsch, A., *Fast Subsumption Algorithms*.
                      Proc. EUROCAL, Springer LNCS, Vol. 204, 1985.

Harrison&Rubin 78   Harrison, M.C., Rubin, N., *Another Generalization of Resolution*.
                      JACM, Vol. 25, No. 3, pp. 341.-351,July 1978.

Herold 85*   Herold, A., *Combination of Unification Algorithms*.
                      SEKI Report SR-85-VIII-KL, FB. Informatik, Univ. of Kaiserslautern.

Herold 87*   Herold, A., *Combination of Unification Algorithms in Equational Theories*.
                      Thesis, FB. Informatik, Univ. of Kaiserslautern, 1987.

Hewitt 72   Hewitt, C., *Description and Theoretical Analysis of PLANNER*.
                      AI-TR-258, MIT, 1972.

Joyner 73   Joyner, W., *Automatic Theorem-Proving and the Decision Problem*.
                      Report 7/73, Center Research Comp. Tech., Harvard University, 1973.

Karl Mark G. Raph 84* Karl Mark G. Raph, *The Markgraf Karl Refutation Procedure*.
                      Interner Bericht, Memo-SEKI-MK-84-01, FB Informatik, Universität Kaiserslautern, 1984.

King 69   King, J., *A Program Verifier*.
                      PhD thesis, Carnegie Mellon, 1969.

Kirchner 85   Kirchner, C., *Méthodes et outils de conception systématique d'algorithmes d'únification dans
                      les théories équationelles*. Thèse d'état de l'Université de Nancy, 1985.

Kirchner 87   Kirchner, C., *Methods and Tools for Equational Unification*.
                      Proc. of Colloq. on Equations in Algebraic Structures, Lakeway, Texas, 1987.

Knuth&Bendix 70   Knuth, D., Bendix, P., *Simple Word Problems in Universal Algebras*.
                      in: Computational Problems in Abstract Algebra. Ed. Leech I., Pergamon Press, pp. 263-
                      297,1970.

Kowalski 75   Kowalski, G., *A Proof Procedure Using Connection Graphs*.
                      J.ACM, Vol. 22, No. 4, 1975.

Kowalski 79   Kowalski, R., *Logic for Problem Solving*.
                      Artificial Intelligence Series, (Nils J. Nilsson, Editor), Vol. 7, North-Holland, New York,
                      1979.

Lewis 78   Lewis, H.R., *Renaming a Set of Clauses as Horn Set*.
                      J.ACM 25, 1, 1978.

Lim&Henschen 85   Lim, Y., Henschen, L.J., *A New Hyperparamodulation Strategy for the Equality Relation*.
                      Proc. of IJCAI-85, Los Angeles, 1985.

Lingenfelder 86*   Lingenfelder, C., *Transformation of Refutation Graphs into Natural Deduction Proofs*.
                      SEKI-Report, SR-86-10, FB. Informatik, Univ. of Kaiserslautern.

Lingenfelder 89*   Lingenfelder, C., *Structuring Computer Generated Proofs*.
                      Proc. of IJCAI-89, Detroit, pp. 378-383, 1989.

Lotz 87*   Lotz, V., *Heuristische Kontrolle des Aufbaus von Gleichungsgraphen*.
                      Diploma thesis, FB. Informatik, Univ. of Kaiserslautern, 1987.

Loveland 78   Loveland, D., *Automated Theorem Proving: A Logical Basis*.
                      Fundamental Studies in Computer Science, Vol. 6, North-Holland, New York, 1978.

Lusk&Overbeek 80   Lusk, E.L., Overbeek, R.A., *Data Structure and Control Architecture for Implementation of
                      Theorem-Proving Programs*. Proc. 5th CADE, Springer LNCS 87, pp. 232-249, 1980.

Lusk&Overbeek 84   Lusk, E.L., Overbeek, R.A., *A Short Problem Set for Testing Systems That Include Equality
                      Reasoning*. Argonne National Laboratory, Argonne, Illinois 600439, 1984.

Lusk&Overbeek 80   Lusk, E.L., Overbeek, R.A. (eds.), *9th International Conference on Automated Deduction*.
                      Springer LNCS 310, 1988.

Lusk et al 82   Lusk, L., McCune, W., Overbeek, R.,
                      Logic Machine Architecture: Kernel Functions and Inference Rules.
                      Proc. of CADE-82, Springer LNCS 138, 1982.

Milner 77   Milner, R., *A Theory of Type Polymorphism in Programming*.
                      Journal of Computer Systems Sciences, 17, pp. 348-375, 1977.

Minker&Wilson 76   Minker, J., Wilson, *Resolution Refinements and Search Strategies: A Comparative Study*.
                      IEEE Transactions on Computers, vol. C-25, no. 8, 1976.

Morris 69   Morris, J.B., *E-Resolution: An Extension of Resolution to include the Equality Relation*.
                      Proc. of IJCAI-69, pp. 287-294, 1969.

Munch 88   Munch, K.H., *A New Reduction Rule for the Connection Graph Proof Procedure.*
                     Journal of Automated Reasoning, Vol. 43, No. 4, pp. 425-444, 1988.

Murray&Rosenthal 87 Murray, N.V., Rosenthal, E., *Inference with Path Resolution and Semantic Graphs.*
                     J.ACM, 34/2, 1987.

Newell et al 59   Newell, A., Shaw, J.C., Simon, H.,
                     Report on a General Problem Solving Program.
                     Proc. of Int. Conf. Information Processing (UNESCO). Paris, 1959.

Nutt et al 89   Nutt, W., Rety, P., Smolka, G., *Basic Narrowing Revisited.*
                     Journal of Symbolic Computation, vol. 7, pp. 295-317, 1989.

Ohlbach 82*   Ohlbach H.J., *The Logic Engine.*
                     MEMO-SEKI-82-II, FB Informatik, Univ. of Karlsruhe, 1982.

Ohlbach 83*   Ohlbach, H.J., *Ein Regelbasiertes Klauselgraphverfahren.*
                     Proc. of GWAI-83, Informatik Fachberichte, Springer Verlag, 1983.

Ohlbach&Wrightson 84* Ohlbach, H.J., Wrightson, G.,
                     Solving a Problem in Relevance Logic with an Automated Theorem Prover.
                     Proc. of 7th CADE, Napa, Springer LNCS 170, pp. 496-508, 1984.

Ohlbach 87*   Ohlbach, H.J., *Link Inheritance in Abstract Clause Graphs.*
                     Journal of Automated Reasoning, Vol. 3, No. 11, pp. 1-34, 1987.

Ohlbach&Schmidt-Schauss 85* Ohlbach, H.J., Schmidt-Schauss, M., *The Lion and the Unicorn.*
                     Journal of Automated Reasoning Vol. 1, 1985, pp. 327-332, 1985.

Ohlbach&Siekmann 88* Ohlbach, H.J., Siekmann, J.,
                     Using Automated Reasoning Techniques for Deductive Databases.
                     SEKI Report SR-88-06, FB Informatik, Univ. Kaiserslautern, 1988.

Ohlbach 88*   Ohlbach,H.J., *A Resolution Calculus for Modal Logics.*
                     Proc. of 9th CADE, Springer LNCS 310, pp. 500-516, 1988.
                     SEKI Report SR-88-08, FB. Informatik, Univ. of Kaiserslautern, 1988.

Ohlbach 89*   Ohlbach, H.J., *Context Logic.*
                     SEKI Report SR-89-08, FB. Informatik, Univ. of Kaiserslautern, 1989.

Peterson & Stickel 81 Peterson, G.E., Stickel, M.E: *Complete Sets of Reductions for some Equational Theories.*
                     Journal of the ACM, Vol. 28,2 1981.

Plotkin 72   Plotkin, G., *Building in Equational Theories.*
                     Machine Intelligence 7, 1972.

Präcklein 85*   Präcklein, A., *Ein Reduktionsmodul für einen automatischen Beweiser.*
                     Diploma Thesis, University of Karlsruhe, 1985.

Robinson &Wos 69   Robinson, G., Wos, L., *Paramodulation and TP in first order theories with equality.*
                     Machine Intelligence 4, pp. 135-150, 1969.

Robinson 65   Robinson, J.A., *A Machine-Oriented Logic Based on the Resolution Principle.* J.ACM, Vol.
                     12, No. 1, pp. 23-41, 1965.

Rusinowitch 87   Rusinowitch, M., *Démonstration Automatique par des Techniques de Réécriture*
                     Thèse d´état, CRIN, Centre de Recherche en Informatique de Nancy, 1987.

Schmidt-Schauss 88* Schmidt-Schauss, M., *Computational aspects of an order sorted logic with term declarations.*
                     Thesis, FB. Informatik, University of Kaiserslautern, 1988.

Schmidt-Schauss 88a* Schmidt-Schauss, M., *Implication of Clauses is Undecidable.*
                     Journal of Theoretical Computer Science, 59, pp. 287-296, 1988.

Schmidt-Schauss 88b* Schmidt-Schauss, M.,*Unification in a Combination of Arbitrary Disjoint Equational Theories.*
                     Proc. of 9th CADE, Springer LNCS 310, pp 378-396, 1988.
                     SEKI Report SR-87-16, FB. Informatik, Univ. of Kaiserslautern.

Shostak 76   Shostak, R.E., *Refutation Graphs.*
                     Artificial Intelligence 7, pp. 51-64, 1976.

Shostak 78   Shostak, R.E., *An Algorithm for Reasoning about Equality.*
                     J.ACM,, Vol. 2211, No. 7, 1978.

Sibert 69   Sibert, E.E., *A machine-oriented Logic incorporating the Equality Axiom.*
                     Machine Intelligence 4, pp. 103-133, 1969.

Sickel 76   Sickel, S., *A Search Technique for Clause Interconnectivity Graphs.*
                     IEEE Trans. on Computers C-25(8), pp. 823-835, 1976.

Siekmann 89*	Siekmann, J., *Unification Theory*. Journal of Symbolic Computation, Special Issue on Unification, C. Kirchner(ed.), vol 7, pp. 207-274, 1989.
Siekmann&Wrightson 80*	Siekmann, J., Wrightson, G., *Paramodulated Connection Graphs*. Acta Informatica, No. 13, pp. 67-86, 1980.
Smullyan 68	Smullyan, R.M., *First-Order Logic*. Springer Verlag, Berlin 1968.
Smullyan 78	Smullyan, R.M., *What is the name of this book?* Prentice-Hall, Englewood Cliffs, New Jersey, 1978.
Smolka 82*	Smolka, G., *Completeness and Confluence Properties of Kowalski´s Clause Graph Calculus*. Univ. Karlsruhe, Techn. Report 31/82, 1982.
Smolka et al 89	Smolka, G., Nutt, W., Goguen J., Meseguer, J.,*Order-sorted equational computation*. in M. Nivat, H. Ait-Kace (eds.) Resolution of Equations in Algebraic Structures, vol. 2, Academic Press, 1989.
Smolka 89	Smolka, G., *Logic Programming over Polymorphically Order-Sorted Types*. Thesis, FB. Informatik, Univ. of Kaiserslautern, 1989.
Socher-Ambrosius 89*	Socher-Ambrosius, R., *Reducing the Derivation of Redundant Clauses in Reasoning Systems*. SEKI Report SR-89-04, FB Informatik, Univ. Kaiserslautern, 1989.
Stickel 81	Stickel, M.E., *A unification algorithm for associative-commutative functions*. JACM 28 (3), pp. 423-434, 1981.
Stickel 85	Stickel, M.E., *Automated Deduction by Theory Resolution*, Journal of Automated Reasoning, Vol. 1, No. 4, pp. 333-356, 1985.
Stickel 87	Stickel, M.E., *A comparison of the variable-abstraction and the constant-abstraction method for the associative-commutative unification*. JAR 3, pp. 285-289, 1987.
Tarski 68	Tarski, A., *Equational Logic and Equational Theories of Algebra*. in Schmidt et al: Contribution to Mathematical Logic, North Holland 1968.
Taylor 79	Taylor, W., *Equational Logic*. Houston Journal of Mathematics, 5, 1979.
Tepp 89*	Tepp, M., *Kombinationsverfahren für Unifikationsalgorithmen*. Diploma thesis, FB. Informatik, Univ. Kaiserslautern, 1989
Walther 81*	Walther, Ch., *Elimination of Redundant Links in Extended Connection Graphs*. Proc. GWAI-81, Springer Informatik Fachberichte, Vol. 47, pp. 201-213, Springer, 1981.
Walther 82*	Walther, Ch., *PLL - A First Order Language for an Automated Theorem Prover*. Bericht 35/82, Univ. Karlsruhe, 1982.
Walther 83*	Walther, Ch., *A Many Sorted Calculus based on Resolution and Paramodulation*. Fak. Informatik, Bericht 34/82, Univ. Karlsruhe, 1983.
Walther 87*	Walther, Ch. *A Many-sorted Calculus based on Resolution and Paramodulation*. Research Notes in Artificial Intelligence, Pitman Ltd., London 1987.
Walther 88	Walther, Ch. *Argument Bounded Algorithms as a Basis for Automation of Termination Proofs*. Proc. of 9th CADE, Springer LNCS 310, pp. 602-621, 1988.
Weigele 85*	Weigele, M. *Ein Regelbasiertes Klauselgraphverfahren*. Diploma Thesis, University of Karlsruhe, 1985.
Wos et al 67	Wos, L., Carson, D., Robinson, G.,Shallar, L., *The Concept of Demodulation in Automated Theorem Proving*. JACM, Vol. 14, No. 4, pp. 698-709, 1967.
Wos et al 84	Wos, L., Overbeek, R., Lusk, E., Boyle, J., *Automated Reasoning - Introduction and Applications*. Prentice-Hall, Englewood Cliffs, NJ, 1984.
Yelick 87	Yelick, K.A., *Unification in Combinations of Collapse-free Regular Theories*. J. of Symbolic Computation 3, pp. 153-181, 1987.

* Publications originating from the MKRP project.