CHAPTER 28

Model Elimination and Connection Tableau Procedures

Reinhold Letz

Gernot Stenz

SECOND READERS: Peter Baumgartner and Uwe Petermann.

Contents

HANDBOOK OF AUTOMATED REASONING
Edited by Alan Robinson and Andrei Voronkov

1. Introduction

The last years have seen many efforts in the development, implementation and application of automated deduction systems. Currently, the most successful theorem provers for classical first-order logic are either based on resolution or on model elimination [Sutcliffe and Suttner 1998]. While resolution is treated in other chapters of this Handbook, see [Bachmair and Ganzinger 2001, Weidenbach 2001] (Chapters 2 and 27), this chapter presents the state-of-the-art of theorem proving using model elimination. Historically, model elimination has been presented in different conceptual frameworks. While the very first paper [Loveland 1968] used a tree-oriented format, a restricted and more resolution-oriented chain notation [Loveland 1969, Loveland 1978] has become the standard for some twenty years.

This changed about ten years ago, when it was recognized that it is more natural to view model elimination as a particular refinement of the tableau calculus, in which connections are employed as a control mechanism for guiding the proof search. In order to emphasize this approach, we introduced the term "connection tableaux" [Schumann and Letz 1990, Letz, Schumann, Bayerl and Bibel 1992, Letz 1993, Letz, Mayr and Goller 1994]. This view had a very fruitful effect on the research in the field. In the meantime, many calculi and proof procedures developed in automated deduction, such as SLD-Resolution, the connection method or systems like Satchmo and MGTP, have been reformulated in tableau style. As a positive result of these activities, the similarities and differences between many calculi with formerly unclear relations could be identified. Furthermore, new calculi have been developed which are based on tableaux and integrate connections in different manners, see, e.g., [Hähnle 2001] (Chapter 3 of this Handbook). The main advantages of viewing model elimination as a tableau calculus are the following. On the one hand, more powerful search pruning mechanisms can be identified. On the other hand, the completeness proofs are simplified significantly.

Since model elimination is one of the main paradigms in automated deduction, a wealth of methods for improving the basic proof procedure have been developed. Many of these methods, however, have a very limited effect, since they improve the performance only for few, often pathological examples. Therefore, we have concentrated on methods that have generally proven successful. Furthermore, we have put emphasis on the presentation of the main paradigms for an efficient implementation of model elimination and its refinements.

When trying to classify the methods of redundancy elimination developed for model elimination, one naturally ends up with three categories. Since in model elimination or connection tableaux the manipulated inferential objects are not clauses but tableaux, which are entire deductions, a significant number of techniques have been developed which permit to identify certain deductions as redundant just because of their internal structures. These techniques, among which the property of regularity is most important, constitute the first class of improvements, which may be termed as *structural* or *local* methods of search pruning. Another source of redundancy in proof search results from the fact that typically certain deductions are redundant in the presence of other ones. These *global* approaches of inter-tableau

pruning form the second class of methods for redundancy elimination. A prominent example of such a method is failure caching. The final improvement of pure model elimination has to do with the length of proofs. Even minimal proofs may become rather long when compared with other calculi. This is because pure model elimination is *cut-free* and the generated deduction objects are trees. Consequently, a proof may contain the same subproof more than once. A further difference between model elimination and calculi such as resolution is that free variables in a tableau are normally considered as *rigid*, in the sense that every variable can be used in one instantiation only. We describe the main methods to overcome these deficiencies, which are controlled cuts and universal variables.

When it comes to the implementation of a theorem prover based on model elimination, we have a very special situation. This is because model elimination is very close to SLD-resolution, which is the basic inference system of the programming language Prolog. Consequently, one can take advantage of this proximity by using as much as possible from the implementation techniques developed for Prolog. One successful such approach is to extend abstract machine technology from the Horn case to the full clausal case. Another possibility consists in taking Prolog itself as a programming language, by which reasonably efficient implementations of model elimination can often be obtained with no or only very little implementational effort. Both of these approaches will be described in detail. But the close relation of model elimination to Prolog may also have negative effects. Especially, it is very difficult to implement refinements of model elimination which do not fit with the basic working principles of Prolog. For example, there is no easy way of integrating equality handling or theory reasoning into an abstract machine. Or, when using Prolog as programming language, it is very difficult to implement a different backtracking mechanism. This has motivated us to develop and present a further implementation architecture which is more modular and flexible and which strongly differs from the ones based on Prolog or Prolog techniques. The key idea for achieving high efficiency in this approach is the extensive re-use of the results of expensive operations.

The development of a redundancy elimination technique is one thing, its efficient implementation is another one. Fortunately, many of the refinements developed for model elimination may be formulated in a uniform general setting, as conditions on the instantiations of variables, so-called disequation constraints. In the final section, we develop the general framework of disequation constraints including universal variables and normalization, and we describe in detail how efficient constraint handlers may be implemented.

2. Clausal Tableaux and Connectedness

2.1. Preliminaries

Before starting with the presentation of clausal tableaux, the meaning of some basic concepts and notations has to be defined. We are working on formulae in

clause logic and use standard conventions for denoting logical symbols and formulae. Our alphabet consists of individual *variables, function* and *predicate symbols* with arities ≥ 0, the logical *connectives* $\neg$ (negation), $\vee$ (disjunction) and $\wedge$ (conjunction), the *universal* and the *existential quantifiers* $\forall$ respectively $\exists$, plus the comma and the parentheses as punctuation symbols. A *term* is either a variable or a string of the form $\alpha(t_1, \ldots, t_n)$ where α is a function symbol of arity n and the t_i are terms. An *atomic formula* is a string of the form $\alpha(t_1, \ldots, t_n)$ where α is a predicate symbol of arity n and the t_i are terms. Expressions of the form $\alpha()$ are conveniently abbreviated by writing just α. First-order formulae, occurrences of expressions in other expressions, the scope of quantifier occurrences, and what it means that a variable occurs free or bound in an expression are defined as usual.

We emphasize the special notions used in this work. The *complement* of a formula F is G if F is of the form $\neg G$, and $\neg F$ otherwise; the complement of a formula is abbreviated as $\sim F$. A *literal* is either an atomic formula or an atomic formula with a negation sign in front of it. A *clause* is a disjunction of literals, i.e., either a literal, called a *unit clause*, or a string of the form $L_1 \vee \cdots \vee L_n$ where the L_i are literals. A *Horn clause* is a clause containing at most one positive literal. A *clausal formula* is a conjunction of clauses, i.e., either a clause or a string of the form $c_1 \wedge \cdots \wedge c_n$ where the c_i are clauses.

A *substitution* σ is any mapping of variables to terms; for any (sequence of) expression(s) F, we abbreviate with $F\sigma$ the (sequence of) expression(s) obtained by simultaneously replacing every free occurrence of a variable in F with its value under σ; $F\sigma$ is called an *instance* of F. The tuples $\langle x, t\rangle$ in a substitution with $x \neq t$ are called *bindings* and abbreviated by writing x/t. Normally, a substitution will be denoted by giving the set of its bindings. If $\{x_1/t_1, x_2/t_2, x_3/t_3, \ldots\}$ is (the set of bindings of) a substitution σ, then $\{x_1, x_2, x_3, \ldots\}$ and $\{t_1, t_2, t_3, \ldots\}$ are called the *domain* and the *range* of σ, respectively. Given two (sequences of) expressions F and G, when a substitution σ satisfies $F\sigma = G\sigma$, then σ is called a *unifier* for F and G. Some specializations of the unifier concept are of high importance. A unifier σ for F and G is termed a *most general unifier* if, for every unifier τ for F and G, there is a substitution θ with $\sigma\theta = \tau$; σ is a *minimal unifier* if the set of its bindings has minimal cardinality. To illustrate the difference, given three distinct variables x, y, z, then $\{x/y\}$ is a minimal unifier for the terms x and y whereas the substitution $\{x/y, z/x\}$ is a most general unifier, but no minimal unifier. Finally, a unifier σ is *idempotent* if $\sigma = \sigma\sigma$. It holds that any minimal unifier is idempotent and most general. Since nearly all unification algorithms return minimal unifiers, we will prefer that term throughout this chapter.

Subsequently, we will preferably employ special meta-expressions for the denotation of formulae and their components. For individual variables we will normally use the letters u, v, w, x, y, z; *constants* are nullary function symbols and are denoted with the letters a, b, c, d; for function symbols of arity > 0 we use f, g, h; predicate symbols are denoted with P, Q, R, nullary predicate symbols are denoted with lower case letters; subscripts will be used when needed.

2.2. *Inference Rules of Clausal Tableaux*

Clausal tableaux are trees labelled with literals (and other control information) inductively defined as follows.

2.1. DEFINITION *(Clausal tableau).* Let S be a set or conjunction of clauses $c_1, \ldots, c_n$. A tree consisting of just one unlabelled node is a *clausal tableau for* S. The single branch of this tree is considered as *open*. If B is an open branch in a clausal tableau T for S with leaf node N (called *subgoal*), then the formula trees obtained from the following two inference rules are *clausal tableaux for* S:

(Expansion rule) Select a clause c_i in S and simultaneously replace all its variables with distinct new variables not occurring in T. Let $L_1 \vee \cdots \vee L_n$ be the resulting clause. Then attach n new nodes as successors of the subgoal N and label them with the literals $L_1, \ldots, L_n$, respectively. The new branches are considered as *open*.

(Closure or reduction rule) If the subgoal N with literal label K has an ancestor node N' with literal L on the branch B such that there exists a minimal unifier σ for K and the complement $\sim L$ of L, then obtain the tableau $T\sigma$, that is, apply the substitution σ to the literals in T. Now the branch B is considered as *closed*.[1]

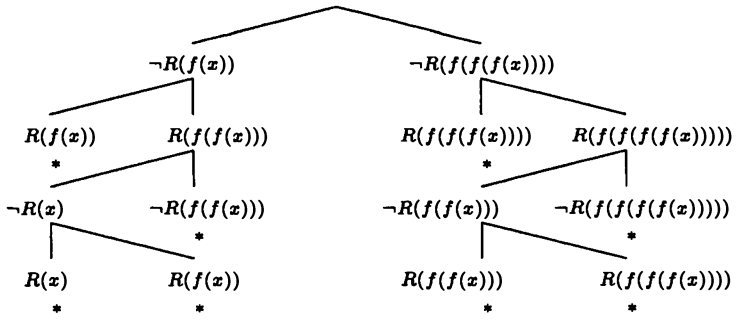


Figure 1: A closed clausal tableau for the formula S consisting of the two clauses $R(x) \vee R(f(x))$ and $\neg R(x) \vee \neg R(f(f(x)))$.

Figure 1 displays a closed clausal tableau, i.e., a tableau with the closure rule applied to all its branches, which we indicate with an asterisk at the end of each branch. In the figure, the unifiers resulting from the closure steps are already applied to the tableau. In general, variables in clausal tableaux are considered as *rigid*, i.e., just as place holders for arbitrary ground terms. In Section 5, it will be shown that this condition can be weakened for certain variables. The example shows the

[1] A branch is closed by applying an inference rule. There are no implicit branch closures.

necessity of renaming variables. Without renaming it would be impossible to unify the second literal in the first clause with the complement of the second literal in the second clause, which is done, for example, in the second closure step on the left. Furthermore, multiple copies of the same input clauses are needed.

Let us make some remarks regarding the peculiarities of this definition as compared with the more familiar definition of tableaux. For one thing, we carry the input set or formula S alongside the tableau and do not put its members at the beginning of the tableau, we leave the root unlabelled instead. This facilitates the comparison of tableaux for different input sets. For example, one tableau may be an instance of another tableau, even if their input sets differ. Also, a branch is considered as closed only if the closure rule was explicitly applied to it, all other branches are considered as open, even when they are complementary. This precaution simplifies the presentation, in particular, of the proof of the Lifting Lemma (Lemma 6.5), and more adequately reflects the actual situation when implementing tableaux.

The clausal tableau calculus is *sound* and *complete*, that is, for every set of clauses S, there exists a closed clausal tableau for S if and only if S is unsatisfiable. Furthermore, the clausal tableau calculus is *(proof) confluent*, i.e., every clausal tableau for an unsatisfiable input formula S can be completed to a closed clausal tableau for S.

Clausal tableaux provide a large potential for refinements, i.e., for imposing additional restrictions on the tableau construction. For instance, one can integrate *ordering restrictions* [Klingenbeck and Hähnle 1994] as they are successfully used in resolution-based systems (see also Chapter 3 in this Handbook). The most important structural refinement of clausal tableaux with respect to automated deduction, however, is to use links or *connections* to guide the proof search.

2.3. Connection Tableaux

Some additional notation will be useful.

2.2. DEFINITION *(Tableau clause)*. For any non-leaf node N in a clausal tableau, the set of nodes $N_1, \ldots, N_m$ immediately below N is called the node *family below* N; if the nodes $N_1, \ldots, N_m$ are labelled with the literals $L_1, \ldots, L_m$ respectively, then the clause $L_1 \vee \cdots \vee L_m$ is named the *tableau clause below* N; The tableau clause below the root node is called the *start* or *top clause* of the tableau.

A closer look at the tableau displayed in Figure 1 reveals an interesting structural property. In every node family below the start clause, at least one node has a complementary ancestor. This property can be formulated in two variants, a weaker one and a stronger one.

2.3. DEFINITION *(Path connectedness, connectedness).*

1. A clausal tableau is said to be *path connected* (or *weakly connected*) iff, in every node family below the start clause, there is one node with a complementary ancestor.
2. A clausal tableau is said to be *connected* (or *tightly connected*) iff, in every node family below the start clause, there is one node which is complementary to its predecessor.

A (path) connected tableau is also called *(path) connection* tableau.

With the connection conditions, every clause has a certain relation to the start clause. This allows a goal-oriented form which may be used to guide the proof search.

Let us make some brief historical remarks on the rôle of connections in tableaux. The notion of a connection is a central concept in automated deduction whereas tableau calculi, traditionally, have no reference to connections—as an example, note that the notion does not even occur in [Fitting 1996]. On the other hand, it was hardly noticed in the field of automated deduction and logic programming that calculi like *model elimination* [Loveland 1968, Loveland 1978], the *connection calculi* in [Bibel 1987], or *SLD-resolution* [Kowalski and Kuehner 1970] should proof-theoretically be considered as tableau calculi. This permits, for instance, to view the calculi as *cut-free* proof systems. The relation of these calculi to tableaux has not been recognized, although, for example, the original presentation of model elimination [Loveland 1968] is clearly in tableau style. The main reason for this situation may be that until recently both communities (tableaux and automated deduction) were almost completely separated. As a further illustration of this fact, note that unification was not really used in tableaux before the end of the eighties [Reeves 1987, Fitting 1990]. In Section 2.7, we will clarify the relation of connection tableaux with model elimination, SLD-resolution, and the connection method.

In order to satisfy the connectedness conditions, for every tableau expansion step except the first one, the closure rule has to be applied to one of the newly attached nodes. This motivates to amalgamate both inference rules into a new macro inference rule.

2.4. DEFINITION *((Path) extension rule).* The *(path) extension rule* is defined as follows: perform a clausal expansion step immediately followed by a closure step unifying one of the newly attached literals, say L, with the complement of the literal at its predecessor node (at one of its ancestor nodes); the literal L and its node are called *entry* or *head literal* and *entry* or *head node*, respectively.

The building of such macro inference rules is a standard technique in automated deduction to increase efficiency. With these new rules, the clausal tableau calculi can be reorganized.

2.5. DEFINITION *((Path) connection tableau calculus).* The *(path) connection tableau calculus* consists of the following three inferences rules:

- the (path) extension rule,
- the closure or reduction rule,
- and the *start rule*, which is simply the expansion rule, but restricted to only one application, namely the attachment of the start clause.

A fundamental proof-theoretical property of the two connection tableau calculi is that they are not proof confluent, as opposed to the general clausal tableau calculus. This can easily be recognized, for instance, by considering the unsatisfiable set of unit clauses $S = \{p, q, \neg q\}$. If we select p as start clause, then the tableau cannot be completed to a closed tableau without violating the (path) connectedness condition. In other terms, using the (path) connectedness condition, one can run into dead ends. The important consequence to be drawn from this fact is that, for those tableau calculi, systematic branch saturation procedures of the type presented in [Smullyan 1968] do not exist. Since an open connection tableau branch that cannot be expanded does not guarantee the existence of a model, connection tableaux are therefore not suited for *model generation*. Weaker connection conditions that are compatible with model generation are described in [Billon 1996, Baumgartner 1998, Baumgartner, Eisinger and Furbach 1999] and in [Hähnle 2001] (Chapter 3 of this Handbook).

2.4. Proof Search in Connection Tableaux

When using non-confluent deduction systems like the connection tableau calculi, in order to find a proof, in general, all possible deductions have to be enumerated in a fair manner until the first proof is found. The *search space* for a tableau enumeration procedure can be defined as a tree of tableaux.

2.6. DEFINITION *((Tableau) search tree)*. Let S be a set of formulae and C a tableau calculus. The *corresponding (tableau) search tree* is a tree $\mathcal{T}$ labelled with tableaux defined by induction on its depth.

1. The root of $\mathcal{T}$ is labelled with the trivial tableau, consisting just of a root node.
2. Every non-leaf node $\mathcal{N}$ of depth n in $\mathcal{T}$ has as many successor nodes as there are successful applications of a single inference step in the tableau calculus C applied to the tableau at the node $\mathcal{N}$ and using formulae from S; the successor nodes of $\mathcal{N}$ of depth $n+1$ are labelled with the resulting tableaux, respectively.

The leaf nodes of a (tableau) search tree can be partitioned into two sets of nodes, the ones labelled with tableaux that are closed, called *success nodes*, and the others which are labelled with open tableaux to which no successful inference steps can be applied, called *failure nodes*. Closed tableaux occurring in a search tree are *proofs*.

It is important to note that the search spaces of tableau calculi cannot be represented by familiar *and-or-trees* in which the and-nodes represent the tableau clauses and the or-nodes the alternatives for expansion. Such a more compact representation is not possible in the first-order case, because the branches in a free-variable tableau cannot be treated independently.

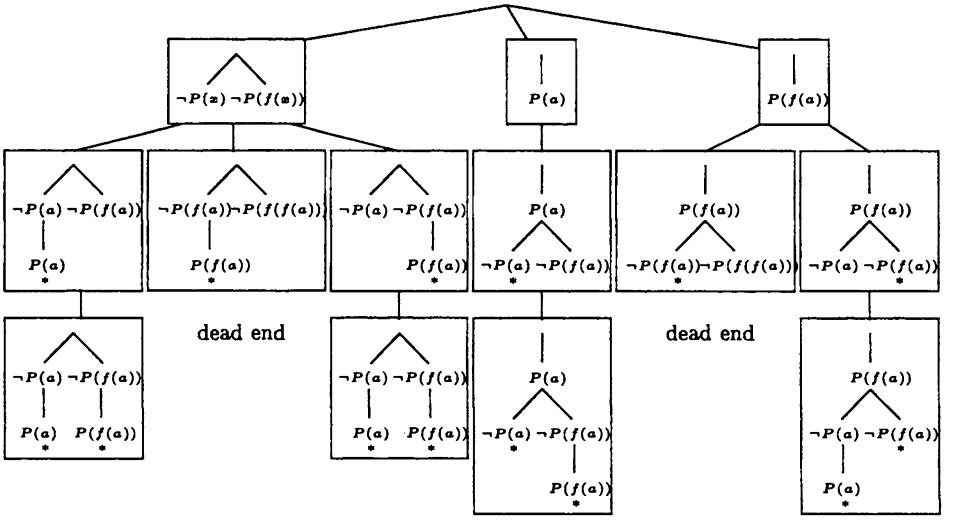


Figure 2: The connection tableau search tree for the set S consisting of the three clauses $\neg P(x) \vee \neg P(f(x))$, $P(a)$, and $P(f(a))$.

In Figure 2, the complete connection tableau search tree for a set of clauses is given. For this simple example, the search tree is finite. Note that the search space of the general clausal tableau calculus (without a connection condition) is infinite for S. This is but one example for the search pruning effect achieved by the connection conditions.

In order to find a connection tableau proof, the corresponding (normally infinite) search tree has to be explored. This can be done *explicitly* by constructing all tableaux in a *breadth-first* manner and working down the search tree level-wise from top to bottom, such a calculus was discussed in [Baumgartner and Brüning 1997]. The explicit construction of all tableaux, however, suffers from an enormous consumption of memory, since the number and the sizes of the generated proof objects significantly grow during the proof process. Furthermore, the computational effort for creating new tableaux increases with the depth of the search tree, since the sizes of the tableaux increase. In contrast, for resolution procedures the *number* of new proof objects (clauses) is generally considered the critical parameter. This sufficiently demonstrates why an explicit tableau enumeration approach should not be pursued in practice.

The customary and successful paradigm therefore is to explore a tableau search tree in an *implicit* manner, using *consecutively bounded depth-first iterative deepening search* procedures. In this approach, iteratively larger finite initial parts of a search tree $\mathcal{T}$ are explored, by imposing so-called *completeness bounds* on the structure of the permitted tableaux. Due to the construction process of tableaux from the root to the leaves, many tableaux have identical or structurally identical

subparts. This suggests exploring finite initial segments in a *depth-first* manner, by employing *structure sharing* techniques and using *backtracking*. More precisely, at each time only one tableau is in memory, which is extended following the branches of the search tree; backtracking occurs when a leaf node of the current initial segment of the search tree has been reached. If no proof was found on one level, then the next level of the iteration is started. A more elaborate description of this proof search procedure will be given in Section 7. Even though according to this methodology initial parts of the search tree are explored several times, no significant efficiency is lost if the initial segments increase exponentially [Korf 1985]. The advantage is that, due to the application of Prolog techniques, very high inference rates can be achieved.

2.5. Completeness Bounds

In this section, different important completeness bounds will be introduced. Formally, a completeness bound can be viewed as a particular size function on tableaux.

2.7. DEFINITION *(Completeness bound).* A *size bound* is a total mapping s assigning to any tableau T a non-negative integer n called the *s-size* of T. A size bound s is called a *completeness bound* for a (clausal) tableau calculus C if, for any finite set S of formulae (clauses) and any $n \geq 0$, the search tree corresponding to C and S contains only finitely many tableaux with s-size less or equal to n.

The finiteness condition qualifies completeness bounds as suitable for iterative deepening search. Given a completeness bound s and an iterative deepening level with size limit n, an implicit deduction enumeration procedure works as follows. Whenever an inference step is applied to a tableau, it is checked whether the s-size of the new tableau is $\leq n$, otherwise backtracking is performed.

2.5.1. Inference bound

The most natural completeness bound is the so-called *inference bound* which counts the number of inference steps that are needed to construct a closed tableau. Using the inference bound, the search tree is explored level-wise; that is, for size n, the search tree is explored until depth $\leq n$. The search effort can be reduced by using *look-ahead* information as follows. As soon as a clause is selected for attachment, its length is taken into account for the current inference number, since obviously, for every subgoal of the clause at least one inference step is necessary to solve it. This enables us to detect the exceeding of the current size limit as early as possible. For example, considering the search tree given in Figure 2, with inference limit 2, one can avoid an expansion step with the first clause $\neg P(x) \vee \neg P(f(x))$, since any closed tableau with this clause as start clause will at least need 3 inference steps. This method was first used in [Stickel 1988].

2.5.2. Depth bound

A further simple completeness bound is the *depth bound*, which limits the length of the branches of the tableaux considered in the current search level. In connection tableaux, one can relax this bound so that it is only checked when non-unit clauses are attached. This implements a kind of unit preference strategy. An experimental comparison of the inference bound and the relaxed depth bound is contained in [Letz et al. 1992].

Both of the above bounds have certain deficiencies in practice. Briefly, the inference bound is too optimistic, since it implicitly assumes that subgoals which are not yet processed may be solved with just one inference step. The weakness of the depth bound, on the other hand, is that it is too coarse in the sense that the number of tableaux in a search tree with depth $\leq n+1$ is usually much larger than the number of tableaux with depth $\leq n$. In fact, in the worst case, the increase function is doubly exponential whereas, in the case of the inference bound, the increase function is exponential at most. Furthermore, both bounds favour tableaux of completely different structures. Using the inference bound, trees containing few long branches are preferred, whereas the depth bound prefers symmetrically structured trees.

2.5.3. A divide-and-conquer optimization of the inference bound

In [Harrison 1996], the following method was applied for avoiding some of the deficiencies of the inference bound. In order to comprehend the essence of the method, assume N_1 and N_2 to be two subgoals (among others) in a tableau and let the number of remaining inferences be k. Now it is clear that one of the two subgoals must have a proof of $\leq k/2$ inferences in order to meet the size limit. This suggests the following two-step algorithm. First, select the subgoal N_1 and attempt to solve it with inference limit $k/2$; if this succeeds, solve the rest of the tableau with whatever is left over from k. If this has been done for all solutions of the subgoal N_1, repeat the entire process for N_2. The advantage of this method is that the exploration of N_1 and N_2 to the full limit k is often avoided. Its disadvantage is that pairs of solutions of the subgoals with size $\leq k/2$ will be found twice, which increases the search space. In order to keep this method from failing in the recursive case, methods of failure caching as presented in Section 4.3 are needed. In practice, this method performs better if smaller limits like $k/3$ or $k/4$ are used instead of $k/2$, although those do not guarantee that all proofs on the respective iterative deepening level can be found. A possible explanation for this improved behavior is that the latter methods tend to prefer short or unit clauses which is a generally successful strategy in automated deduction (see also Section 2.6.2 where a similar effect may be achieved with a method based on a different idea).

2.5.4. Clause dependent depth bounds

Other approaches aim at improving the depth bound. The depth bound is typically implemented as follows. For a given tableau depth limit, say k, every node in the tableau is labelled with the value $k-d$ where d is the distance from the root node. If this value of a node is 0, then no tableau extension is permitted at this node.

Accordingly, one may call this value of a node its *resource*. This approach permits a straightforward generalization of the depth bound. Instead of giving the open successors $N_1, \ldots, N_m$ of a tableau node N with resource i the resource $j = i - 1$, the resource j of each of $N_1, \ldots, N_m$ is the value of a function r of two arguments, the resource i of N and the number m of new subgoals in the attached clause. We call such bounds *clause dependent depth bounds*. With clause dependent depth bounds a smoother increase of the iterative deepening levels can be obtained. Two such clause dependent depth bounds have been used in practice, one defined by $r(i,m) = i - m$ (this bound is available in the system SETHEO since version V.3 [Goller, Letz, Mayr and Schumann 1994]) and the other by $r(i,m) = (i-1)/m$ (this bound was called *sym* in [Harrison 1996]).

2.5.5. (Inference) weighted depth bounds

Although a higher flexibility can be obtained with clause dependent depth bounds, all these bounds are pure depth bounds in some sense, since the resource j of a node is determined at the time the node is attached to the tableau. In order to increase the flexibility and to permit an integration of features of the inference bound, the so-called *weighted depth bounds* have been developed. The main idea of the weighted depth bounds is to use a bound such as the clause dependent depth bound as a basis, but to take the inferences into account when eventually allocating the resource to a subgoal. In detail, this is controlled by three parameterized functions w_1, w_2, w_3 as follows. When entering a clause with m subgoals from a node with resource i, first, the maximally available resource j for the new subgoals is computed according to a clause dependent depth bound, i.e., $j = w_1(i,m)$. Then, the value j is divided into two parts, a *guaranteed* part $j_g = w_2(j,m) \leq j$ and an *additive* part $j_a = j - j_g$. Whenever a subgoal is selected, the additive part is modified depending on the inferences Δi performed since the clause was attached to the tableau,[2] i.e., $j'_a = w_3(j_a, \Delta i)$. The resource finally allocated for a selected subgoal then is $j_g + j'_a$.

Depending on the parameter choices for the functions w_1, w_2, w_3, the respective weighted depth bound can simulate the inference bound ($w_1(i,m) = i - m$, $w_2(j,m) = 0$, $w_3(j_a, \Delta i) = j_a - \Delta i$) or the (clause dependent) depth bound(s) or any combination of them.

A parameter selection which represents a simple new completeness bound combining inference and depth bound is, for example, $w_1(i,m) = i - 1$, $w_2(j,m) = j - (m-1)$, $w_3(j_a, \Delta i) = j_a/(1+\Delta i)$. For certain formula classes, this bound turned out to be much more successful than each of the other bounds [Moser, Ibens, Letz, Steinbach, Goller, Schumann and Mayr 1997]. One reason for the success of this strategy is that it also performs a unit preference strategy.

[2] We assume that the look-ahead optimization is used, according to which reduction steps and extension steps into unit clauses do not increase the current inference value. This implies that $\Delta i = 0$ if no extension steps into non-unit clauses have been performed on subgoals of the current clause.

2.6. *Subgoal Processing*

There is a source of indeterminism in the clausal tableau calculi presented so far that can be removed without any harm. This indeterminism concerns the selection of the next subgoal at which an expansion, extension, or closure step is to be performed.

2.8. DEFINITION *(Subgoal selection function).* A *(subgoal) selection function* ϕ is a mapping assigning an open branch with subgoal N to every open tableau T. Let ϕ be a subgoal selection function and $S = T_1, \ldots, T_n$ a sequence of tableaux. If each tableau T_{i+1} in S can be obtained from T_i by performing an inference step on the subgoal $\phi(T_i)$, then we say that S and T_n are *constructed according to* ϕ.

Most complete refinements and extensions of clausal tableau calculi developed to date are *independent of the subgoal selection*, i.e., the completeness holds for any subgoal selection function (for exceptions see Section 6.2 and Section 7 in [Letz et al. 1994]). If a calculus has this property, then it is possible to choose one subgoal selection function ϕ in advance and ignore all tableaux in the search tree that are not constructed according to ϕ. This way the search effort can be reduced significantly. As an illustration of this method of *search pruning*, consider the search tree displayed in Figure 2. For this simple tree, one can only distinguish two subgoal selection functions ϕ_1 and ϕ_2. ϕ_1 selects the left subgoal and ϕ_2 the right subgoal in the start clause. When deciding for ϕ_2, the three leftmost lower boxes will vanish from the search tree. In case ϕ_1 is used, only two boxes will be pruned away.

For the clausal tableau calculi presented up to this point, even the following stronger independence property holds.

2.9. PROPOSITION (Strong independence of subgoal selection). *Given any closed (path) (connection) tableau T for a set of clauses S constructed with n inference steps, then for any subgoal selection function ϕ, there exists a sequence $T_0, \ldots, T_n$ of (path) (connection) tableaux constructed according to ϕ such that T_n is closed and T is an instance of T_n, i.e. $T = T_n\sigma$ for some substitution σ.*

PROOF. See [Letz 1999]. □

In case a calculus is strongly independent of the subgoal selection, not only completeness is preserved, but minimal proof lengths as well. Furthermore, if a completeness bound of the sort described above is used, then the iterative deepening level on which the first proof is found is always the same, independent of the subgoal selection. Note that the strong independence of the subgoal selection (and hence minimal proof lengths) will be lost for certain extensions of the clausal tableau calculus such as *folding up* [Letz et al. 1994] and the *local* closure rule which is discussed below.

One particular useful form of choosing subgoals is *depth-first* selection, i.e., one always selects the subgoal of an open branch of maximal length in the tableau. *Depth-first left-most/right-most* selection always chooses the subgoal on the left-most/right-most open branch (which automatically has maximal depth). Depth-

first left-most selection is the built-in subgoal selection strategy of Prolog. Depth-first selection has a number of advantages, the most important being that the search is kept relatively local. Furthermore, very efficient implementations are possible.

2.6.1. Subgoal reordering

The order of subgoal selection has influences on the size of the search space, as shown by the search tree above. This is because subgoals normally share variables and thus the solution substitutions of one subgoal have an influence on the solution substitutions of the other subgoals.

A general *least commitment* paradigm is to prefer subgoals that produce fewer solutions. In order to identify a non-closable connection tableau as early as possible, the solutions of a subgoal should be exhausted as early as possible. Therefore, subgoals for which probably only few solutions exist should be selected earlier than subgoals for which many solutions exist. This results in the *fewest-solutions* principle for subgoal selection.

Depth-first selection means that all subgoal alternatives stem from one clause of the input set. Therefore, the selection order of the literals in a clause can be determined *statically*, i.e., once and for all before starting the proof search, as in [Letz et al. 1992]. But subgoal selection can also be performed *dynamically*, whenever the literals of the clause are handled in a tableau. The static version is cheaper (in terms of performed comparisons), but often an optimal subgoal selection cannot be determined statically, as can be seen, for example, when considering the transitivity clause $P(x,z) \vee \neg P(x,y) \vee \neg P(y,z)$. Statically, none of the literals can be preferred. Dynamically, however, when performing an extension step entering the transitivity clause from a subgoal $\neg P(a,z)$, the first subgoal $\neg P(x,y)$ is instantiated to $\neg P(a,y)$. Since it contains only one variable now, it should be preferred according to the fewest-solutions principle. Entering the transitivity clause from a subgoal $\neg P(x,a)$ leads to preference of the second subgoal $\neg P(y,a)$.

2.6.2. Subgoal alternation

When a subgoal in a tableau has been selected for solution, a number of complementary unification partners are available, viz. the connected path literals and the connected literals in the input clauses. Together they form the so-called *choice point* of the subgoal. One common principle of standard backtracking search procedures in model elimination (and in Prolog) is that, whenever a subgoal has been selected, its choice point must be completely finished, i.e., when retracting an alternative in the choice point of a subgoal, one has to stick to the subgoal and try another alternative in its choice point. This standard methodology has an interesting search-theoretic weakness.

This can be shown by the following generic example, variants of which often occur in practice. Given the subgoals $\neg P(x,y)$ and $\neg Q(x,y)$ in a tableau, assume the following clauses to be in the input.

(1) $P(a,a)$,
(2) $P(x,y) \vee \neg P'(x,z) \vee \neg P'(y,z)$,

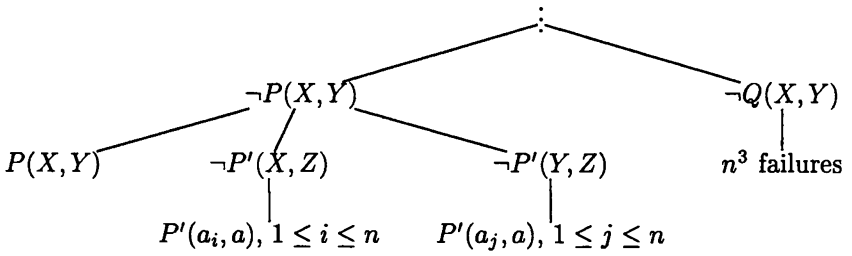


Figure 3: Effort in case of standard subgoal processing.

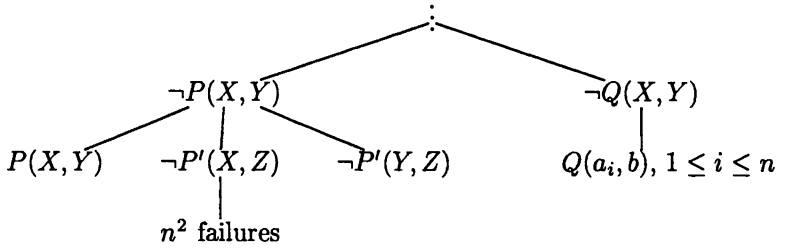


Figure 4: Effort when switching to another subgoal.

(3) $P'(a_i,a), \quad 1 \leq i \leq n,$
(4) $Q(a_i,b), \quad 1 \leq i \leq n.$

Suppose further we have decided to select the first subgoal and perform depth-first subgoal selection. The critical point, say at time t, is after unit clause (1) in the choice point was tried and no compatible solution instance for the other subgoal was found. Now we are forced to enter clause (2). Obviously, there are n^2 solution substitutions (unifications) for solving clause (2) (the product of the solutions of its subgoals). For each of those solutions, we have to perform n unifications with the Q-subgoal, which all fail. Including the unifications spent in clause (2), this amounts to a total effort of $1+n+n^2+n^3$ unifications (see Figure 3). Observe now what would happen when at time t we would not have entered clause (2), but would switch to the Q-subgoal instead. Then, for each of the n solution substitutions $Q(a_i,b)$, one would jump to the P-subgoal, enter clause (2) and perform just n failing unifications for its first subgoal. This sums up to a total of just $n+n(1+n) = 2n+n^2$ unifications (see Figure 4).

It is apparent that this phenomenon is related to the fewest-solutions principle. Clause (2) generates more solutions for the subgoal $\neg P(X,Y)$ than the clauses in the choice point of the subgoal $\neg Q(X,Y)$. This shows that taking the remaining alternatives of *all* subgoals into account provides a choice which can better satisfy the fewest-solution principle. As a general principle, *subgoal alternation* always switches to the subgoal the current connected clause of which is likely to produce

the fewest solutions.

One might argue that with a different subgoal selection, selecting the Q-subgoal first could also avoid the cubic effort. But it is apparent that the example could be extended so that the Q-subgoal would additionally have a longer clause as an alternative, so that the total number of its solutions would be even larger than that of the P-subgoal. In this case, with subgoal alternation one could jump back to the P-subgoal and try clause (2) next, in contrast to standard subgoal selection. Another possibility of jumping to the Q-subgoal *after* having entered clause (2) would be free subgoal selection. In fact, subgoal alternation under depth-first subgoal selection comes closer to standard free subgoal selection, but both methods are not identical.

The question is, when it is worthwhile to stop the processing of a choice point and switch to another subgoal? As a matter of fact, it cannot be determined in advance, how many solutions a clause in the choice point of a subgoal produces for that subgoal. A useful criterion, however, is the *shortest-clause* principle, since, in the worst case, the number of subgoal solutions coming from a clause is the product of the numbers of solutions of its subgoals.[3]

In summary, subgoal alternation works as follows. The standard subgoal selection and clause selection phases are combined and result in a single selection phase that is performed before each derivation step. The selection yields the subgoal for which the most suitable unification partner exists wrt. the number of solutions probably produced. This is done by comparing the unification partners of all subgoals with each other using, for instance, the shortest-clause principle. If more than one unification partner is given the mark of 'best choice', their corresponding subgoals have to be compared due to the principles for standard subgoal selection, namely the first-fail principle and the fewest-solutions principle.

In order to compare the way subgoal alternation (using the shortest-clause principle) works to the standard non-alternating variant, consider two subgoals A and B with clauses of lengths 1,3,5 and 2,4,6 in their choice points, respectively. Table 1 illustrates the order in which clauses are tried.

Subgoal alternation has a number of interesting effects when combined with other methods in model elimination. First note that the method leads to the preference of short clauses. A particularly beneficial effect of preferring short clauses, especially the preference of unit clauses, is the early instantiation of variables. Unit clauses are usually more instantiated than longer clauses, because they represent the "facts" of the input problem, whereas longer clauses in general represent the axioms of the underlying theory. Since normally variables are shared between several subgoals, the solution of a subgoal by a unit clause usually leads to instantiating variables in other subgoals. These instantiations reduce the number of solutions of the other subgoals and thus reduce the search space to be explored when selecting them. Advantage is also taken from subgoal alternation when combined with local failure caching considered in Section 4.3. Failure caching can only exploit information from *closed*

[3] Also, the number of variables in the *calling* subgoal and in the *head* literal of a clause matter for the number of solutions produced.

standard backtracking	subgoal alternation
A1 B2	A1 B2
A1 B4	A1 B4
A1 B6	A1 B6
A3 B2	≀ B2 A3
A3 B4	B2 A5
A3 B6	≀ A3 B4
A5 B2	A3 B6
A5 B4	≀ B4 A5
A5 B6	≀ A5 B6

Table 1: Order of tried clauses for subgoals A and B with clauses of lengths 1,3,5 and 2,4,6 in their choice points, respectively. ≀ indicates subgoal alternations.

sub-tableaux, thus a large number of small subproofs provides more information for caching than a small number of large sub-tableaux that cannot be closed. Since subgoal alternation prefers short clauses and hence small subproofs, the local failure caching mechanism is supported.

Subgoal alternation leads to the simultaneous processing of several choice points. This provides the possibility of computing *look-ahead information* concerning the minimal number of inferences still needed for closing a tableau. A simple estimate of this inference value is the number of subgoals plus the number of all subgoals in the shortest alternative of each subgoal. In general, when using standard subgoal selection, every choice point with literal L except the current one contains connected path literals and connected unit clauses, that is, possibly L can be solved in one inference step. Using subgoal alternation, the reduction steps and the extension steps with unit clauses have already been tried at several choice points, so that only the unification partners *in non-unit clauses* are left in the choice points of several subgoals. Thus more information about the required inference resources can be obtained than in the standard procedure. This *look-ahead information* can be used for search pruning, whenever the number of inferences has an influence on the search bound.[4]

However, under certain circumstances alternating between subgoals may be disadvantageous. If a subgoal cannot be solved at all, switching to another subgoal may be worse than sticking to the current choice point, since the latter may lead to an earlier retraction of the whole clause. This is important for ground subgoals in particular, because they have at most one solution substitution in the Horn case. Since ground subgoals do not contain free variables, they normally cannot profit from early instantiations achieved by subgoal alternation, i.e., switching to brother

[4]This technique is also used by the SETHEO system [Moser et al. 1997].

subgoals and instantiating their free variables cannot lead to instantiations within a ground subgoal. Therefore, when processing a ground subgoal, the fewest-solutions principle for subgoal selection becomes more important than the shortest-clause principle for subgoal alternation. For this reason, subgoal alternation should not be performed when the current subgoal is ground.

2.7. Connection Tableaux and Related Calculi

Due to the fact that tableau calculi work by building up tree structures whereas other calculi derive new formulae from old ones, the close relation of tableaux with other proof systems is not immediately evident. There exist similarities of tableau proofs to deductions in other calculi. In order to clarify the interdependencies, it is helpful to reformulate the process of tableau construction in terms of formula generation procedures. There are two natural formula interpretations of tableaux which we shall mention and which both have their merits.

2.10. DEFINITION. The *branch formula* of a formula tree T is the disjunction of the conjunctions of the formulae on the branches of T.

Another finer view is preserving the underlying tree structure of the formula tree.

2.11. DEFINITION *(Formula of a formula tree (inductive)).*

1. The *formula* of a one-node formula tree labelled with the formula F is simply F.
2. The *formula* of a complex formula tree with root N (with label F) and immediate formula subtrees $T_1, \ldots, T_n$, in this order, is $F \wedge (F_1 \vee \cdots \vee F_n)$ (or simply $F_1 \vee \cdots \vee F_n$ if N is unlabelled) where F_i is the formula of T_i, for every $1 \leq i \leq n$.

Evidently, the branch formula and the formula of a formula tree are equivalent. Futhermore, it is clear that the following proposition holds, from which, as a corollary, also follows the soundness of the method of clausal tableaux.

2.12. PROPOSITION. *If F is the (branch) formula of a clausal tableau for a set of clauses S, then F is a logical consequence of S.*

PROOF. Trivial. □

With the formula notation of tableaux, one can identify a close correspondence of tableau deductions to calculi of the *generative* type. This way, the relation between tableaux and Gentzen's *sequent system* was shown in [Smullyan 1968] using so-called *block tableaux*. We are interested in recognizing similarities to calculi from the field of automated deduction. For this purpose, it is helpful to only consider the *open parts* of tableaux, which we call *goal trees*.

2.13. DEFINITION *(Goal tree).* The *goal tree of* a tableau T is the formula tree obtained from T by cutting off all closed branches.

The goal tree of a tableau contains only the open branches of a tableau. Obviously, for the continuation of the refutation process, all other parts of the tableau may be disregarded without any harm.

2.14. DEFINITION *(Goal formula).*

1. The *goal formula* of any closed tableau is the falsum $\bot$.
2. The *goal formula* of any open tableau is the formula of the goal tree of the tableau.

Using the goal formula interpretation, the tableau construction can be viewed as a linear deduction process in which a new goal formula is always deduced from the previous one until eventually the falsum is derived. In Example 2.15, we give a goal formula deduction that corresponds to the construction of the tableau in Figure 1, under a branch selection function ϕ that always selects the right-most branch.

2.15. EXAMPLE *(Goal formula deduction).* The set of clauses $S = \{R(x) \vee R(f(x)), \neg R(x) \vee \neg R(f(f(x)))\}$ has the following goal formula refutation.

$\neg R(x) \vee \neg R(f(f(x)))$
$\neg R(x) \vee (\neg R(f(f(x))) \wedge R(f(f(f(x)))))$
$\neg R(x) \vee (\neg R(f(f(x))) \wedge R(f(f(f(x)))) \wedge \neg R(f(x)))$
$\neg R(x) \vee (\neg R(f(f(x))) \wedge R(f(f(f(x)))) \wedge \neg R(f(x)) \wedge R(f(f(x))))$
$\neg R(x)$
$\neg R(f(x)) \wedge R(f(f(x)))$
$\neg R(f(x)) \wedge R(f(f(x))) \wedge \neg R(x)$
$\neg R(f(x)) \wedge R(f(f(x))) \wedge \neg R(x) \wedge R(f(x))$
$\bot$

2.16. PROPOSITION. *The goal formula of any clausal tableau T is logically equivalent to the formula of T.*

PROOF. Trivial. □

2.7.1. Model elimination chains

Using the goal tree or goal formula notation, one can easily identify a close similarity of connection tableaux with the model elimination calculus as presented in [Loveland 1978], which we will discuss in some more detail. As already mentioned, model elimination was originally introduced as a tree-based procedure with the full generality of subgoal selection in [Loveland 1968], although the deductive object of a tableau is not explicitly used in this paper. As Don Loveland has pointed out, the linearized version of model elimination presented in [Loveland 1969, Loveland 1978] was the result of an adaptation to the resolution form. Here, we treat a subsystem of model elimination without factoring and lemmata, called *weak model elimination* in

[Loveland 1978], which is still refutation-complete. The fact that weak model elimination is indeed a specialized subsystem of the connection tableau calculus becomes apparent when considering the goal formula deductions of connection tableaux. The weak model elimination calculus can be considered as the special case of the connection tableau calculus where the selection of open branches is performed in a *depth-first right-most* or *left-most* manner, i.e., always the right-most (left-most) open branch has to be selected. Let us choose the right-most variant for now. Due to this restriction of the subgoal selection, a one-dimensional "chain" representation of goal formulae is possible in which no logical operators are necessary. The transformation from goal formulae with a depth-first right-most selection function to model elimination chains works as follows. To any goal formula generated with a depth-first right-most selection function, apply the following operation: replace every conjunction $L_1 \wedge \cdots \wedge L_n \wedge F$ with $[L_1 \cdots L_n]F$ and delete all disjunction symbols.

In a model elimination chain, the occurrences of bracketed literals denote the non-leaf nodes and the occurrences of unbracketed literals denote the subgoals of the goal tree of the tableau. For every subgoal N corresponding to an occurrence of an unbracketed literal L, the bracketed literal occurrences to the left of L encode the ancestor nodes of N. The model elimination proof corresponding to the goal formula deduction given in Example 2.15 is depicted in Example 2.17.

2.17. EXAMPLE *(Model elimination chain deduction).* The set $\{R(x) \vee R(f(x)), \neg R(x) \vee \neg R(f(f(x)))\}$ has the following model elimination chain refutation.

$\neg R(x)\ \neg R(f(f(x)))$
$\neg R(x)\ [\ \neg R(f(f(x)))\ ]\ R(f(f(f(x))))$
$\neg R(x)\ [\ \neg R(f(f(x)))\ R(f(f(f(x))))\ ]\ \neg R(f(x))$
$\neg R(x)\ [\ \neg R(f(f(x)))\ R(f(f(f(x))))\ \neg R(f(x))\ ]\ R(f(f(x)))$
$\neg R(x)$
$[\ \neg R(f(x))\ ]\ R(f(f(x)))$
$[\ \neg R(f(x))\ R(f(f(x)))\ ]\ \neg R(x)$
$[\ \neg R(f(x))\ R(f(f(x)))\ \neg R(x)\ ]\ R(f(x))$
$\perp$

It is evident that weak model elimination is a refinement of the connection tableau calculus, in which a fixed depth-first selection function is used. Viewing chain model elimination as a tableau refinement has various proof-theoretic advantages concerning generality and the possibility of defining extensions and refinements of the basic calculus. Also the soundness and completeness proofs of chain model elimination are immediate consequences of the soundness and completeness proofs of connection tableaux, which are very short and simple if compared with the rather involved proofs in [Loveland 1978]. Subsequently, we will adopt the original and more general view of model elimination as intended by Don Loveland [Loveland 1968] and use the terms connection tableaux and model elimination synonymously.

It is straightforward to recognize that SLD-resolution, although traditionally introduced as a resolution refinement, can also be viewed as a restricted form of model

elimination where the reduction steps are omitted. If the underlying formula is a *Horn formula*, i.e., contains Horn clauses only, then it is obvious that this restriction on model elimination preserves completeness.[5]

2.7.2. The connection method

Another framework in automated deduction which is related with tableaux is the *connection method* [Andrews 1981, Bibel 1987]. We briefly mention the fundamental concepts of the connection method here, since they will be used for a search pruning technique presented in Section 4.1.

2.18. DEFINITION *(Path, connection, mating, spanning property).* Given a set of clauses $S = \{c_1, \ldots, c_n\}$, a *path through* S is a set of n literal occurrences in S, exactly one from each clause in S. A *connection in* S is a two-element subset of a path through S such that the corresponding literals are complementary. Any set of connections in S is called a *mating in* S. A mating M is said to be *spanning for* S if every path through S is a superset of a connection in M.

A set of ground clauses S is unsatisfiable if and only if there is a spanning mating for S. The most natural method for finding and identifying a spanning mating as such is to use a tree-oriented path checking procedure which decomposes the formula, much the same as in the tableau framework, but guided by connections. The majority of those connection *calculi* in [Bibel 1987] can therefore be considered as connection *tableau* calculi, using the weaker path connectedness condition. Thus, every closed (path) connection tableau for a set S determines a spanning mating for S. In the first-order case, the notions of multiplicities and unification come into play, which we will not treat here. For a more detailed comparison, see [Letz et al. 1994, Letz 1998*b*].

3. Further Structural Refinements of Clausal Tableaux

The pure calculus of connection tableaux is only moderately successful in automated deduction. This is because the corresponding search trees are still full of redundancies. In general, there are different methodologies for reducing the search effort of tableau search procedures. In this section, we consider methods which attempt to restrict the tableau *calculus*, that is, disallow certain inference steps if they produce tableaux of a certain *structure*—note that the connection condition is such a structural restriction on general clausal tableaux. The effect on the tableau search tree is that the respective nodes together with the dominated subtrees can be ignored so that the branching rate of the tableau search tree decreases. These *structural* methods of redundancy elimination are *local* pruning techniques in the sense that they can be performed by looking at single tableaux only.

[5] When starting with an all-negative clause, reduction steps are not possible for syntactic reasons.

3.1. Regularity

A fundamental structural refinement of tableaux is the so-called regularity condition.

3.1. DEFINITION *(Regularity).* A clausal tableau is *regular* if on no branch a literal occurs more than once.

The term "regular" has been used to emphasize the analogy to the definition of *regular* resolution [Tseitin 1970]. Imposing the regularity restriction has some important consequences. First, for general clausal tableaux, every closed tableau of minimal size is guaranteed to be regular. Therefore, regularity preserves minimal proof lengths. Furthermore, using regularity the tableau search space of any ground formula becomes finite. While the latter condition also holds for connection tableaux, minimal closed connection tableaux may not be regular. In [Letz et al. 1994] it is shown that regular connection tableaux cannot even polynomially simulate connection tableaux. Nevertheless, a wealth of experimental results clearly shows that this theoretical disadvantage is more than compensated for by the strong search pruning effect of regularity [Letz et al. 1992], so that this refinement is indispensable for any model elimination proof procedure.

3.2. Tautology Elimination

Normally, it is a good strategy to eliminate certain clauses from the input set which can be shown to be redundant for finding a refutation. Tautological clauses are of such a sort.[6] In the ground case, tautologies may be identified once and for ever in a preprocessing phase and can be eliminated before starting the actual proof search. In the first-order case, however, it may happen that tautologies are generated dynamically. Let us demonstrate this phenomenon with the example of the clause $\neg P(x,y) \vee \neg P(y,z) \vee P(x,z)$ expressing the transitivity of a relation. Suppose that during the construction of a tableau this clause is used in an extension step (for simplicity renaming is neglected). Assume further that after some subsequent inference steps the variables y and z are instantiated to the same term t. Then a tautological instance $\neg P(x,t) \vee \neg P(t,t) \vee P(x,t)$ of the transitivity formula has been generated. Since no tautological clause is relevant in a set of formulae, connection tableaux with tautological tableau clauses need not be considered when searching for a refutation. Therefore the respective tableau and any extension of it can be disregarded.

Please note that the conditions of tautology-freeness and regularity are partially overlapping. More specifically, the non-tautology condition on the one hand covers all occurrences of identical predecessor nodes, but not the more remote ancestors. The regularity condition on the other hand captures all occurrences of tautological

[6] Of course, tautologies may facilitate the construction of smaller tableau proofs, since they can be used to simulate the cut rule. Yet, an *uncontrolled* use of cuts is not desirable at all.

clauses for backward reasoning with Horn clauses (i.e. with negative start clauses only), but not for non-Horn clauses.

3.3. Tableau Clause Subsumption

An essential pruning method in resolution theorem proving is *subsumption deletion*, which during the proof process deletes any clause that is subsumed by another clause, and this way eliminates a lot of redundancy. Although no new clauses are generated in the tableau approach, a restricted variant of clause subsumption reduction can be used in the tableau framework, too. First, we briefly recall the definition of subsumption between clauses.

3.2. DEFINITION *(Subsumption for clauses).* Given two clauses c_1 and c_2, we say that c_1 *subsumes* c_2 if there is a variable substitution σ such that the set of literals contained in $c_1\sigma$ is a subset of the set of literals contained in c_2.

Similar to the dynamic generation of tautologies, it may happen, that a clause which has been attached in a tableau step during the tableau construction process is instantiated and then subsumed by another clause from the input set. As an example, suppose the transitivity clause from above and a unit clause $P(a,b)$ are contained in the input set. Now, if the transitivity clause is used in a tableau and if after some inference steps the variables x and z are instantiated to a and b, respectively, then the resulting tableau clause $\neg P(a,y) \vee \neg P(y,b) \vee P(a,b)$ is subsumed by $P(a,b)$. Obviously, for any closed tableau using the former tableau clause a closed tableau exists which uses the latter clause instead.

Again there is the possibility of a pruning overlap with the regularity and the non-tautology conditions. Note that, strictly speaking, avoiding tableau clause subsumption is not a pure *tableau structure* restriction, since a case of subsumption cannot be defined by merely looking at the tableau. Additionally, it is necessary to take the respective input set into account.

3.4. Strong Connectedness

When employing an efficient transformation from the general first-order format to clausal form, new predicates are sometimes introduced which are used to abbreviate certain formulae [Eder 1984, Plaisted and Greenbaum 1986, Boy de la Tour 1990]. Assume, for instance, we have to abbreviate a conjunction of literals $a \wedge b$ with a new predicate d by introducing a biconditional $d \leftrightarrow a \wedge b$. This rewrites to the three clauses $\sim a \vee \sim b \vee d$, $a \vee \sim d$, and $b \vee \sim d$. Interestingly, every resolvent between the three clauses is a tautology. Applied to the tableau construction, this means that whenever one of these clauses is immediately below another one, then a hidden form of a tautology has been generated as shown in Figure 5. (This example also illustrates that the effect of the cut rule can be simulated by suitable definitions.)

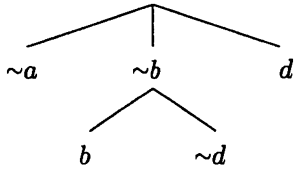


Figure 5: Hidden tautologies in tableaux.

Please note that certain cases of such hidden tautologies may be avoided. For this purpose the notion of connectedness was strengthened to *strong connectedness* in [Letz 1993].

3.3. DEFINITION *(Strong connectedness).* Two clauses c_1 and c_2 are *strongly connected* if there is a substitution σ such that $c_1\sigma$ contains exactly one literal whose complement occurs in $c_2\sigma$, i.e. c_1 and c_2 can be used as the parent clauses of a non-tautological resolvent.

In Section 6.1, it will be proven that, for all pairs of adjacent tableau clauses, strong connectedness may be demanded without losing completeness. However, it is essential that the two clauses are adjacent, i.e. one must be located immediately below the other. For more distant pairs of tableau clauses one dominated by the other, the condition that they be strongly connected is not compatible with the condition of regularity. An example for this is given in Figure 6. This figure shows the only closed strongly connected regular tableau with top clause $\{\neg p, \neg q\}$. Note that the top clause and the clause $\{p, q\}$ have only tautological resolvents. This cannot be avoided even when additional inference rules like factorization or folding up (see Section 5) are available.

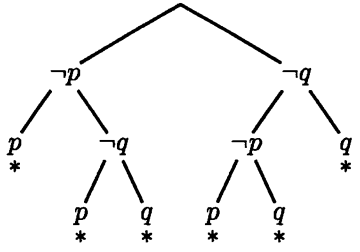


Figure 6: A strongly connected tableau for $\{\{p, q\}, \{p, \neg q\}, \{\neg p.q\}, \{\neg p.\neg q\}\}$.

3.5. Use of Relevance Information

By using *relevance information*, the set of possible start clauses can be minimized.

3.4. DEFINITION *(Essentiality, relevance, minimal unsatisfiability).* A formula F is called *essential in* a set S of formulae if S is unsatisfiable and $S \setminus \{F\}$ is satisfiable. A formula F is named *relevant in* S if F is essential in some subset of S. An unsatisfiable set of formulae S is said to be *minimally unsatisfiable* if each formula in S is essential in S.

As will be shown in Section 6, the connection tableau calculus is complete in the strong sense that, for every relevant clause in a set S, there exists a closed connection tableau for S with this clause as the start clause. Since in any unsatisfiable set of clauses, some negative clause is relevant, it is sufficient to consider only negative clauses as start clauses. The application of this default pruning method achieves a significant reduction of the search space. In many cases, one has even more information concerning the relevance of certain clauses. Normally, a satisfiable subset of the input is well-known to the user, namely, the clauses specifying the theory axioms and the hypotheses. Such relevance information is also provided in the TPTP library [Sutcliffe, Suttner and Yemenis 1994]. A goal-directed system can enormously profit from the relevance information by considering only those clauses as start clauses that stem from the conjecture. As an example, consider an axiomatization of set theory containing the basic axiom that the empty set contains no set, which is normally expressed as a negative unit clause. Evidently, it is not very reasonable to start a refutation with this clause.

It is important to note, however, that when relevance information is being employed, then *all* conjecture clauses have to be tried as start clauses and not only the all-negative ones. Relevance information is normally more restrictive than the default method except when *all* negative clauses are stemming from the conjecture, in which case obviously the default mode is more restrictive.

4. Global Pruning Methods in Model Elimination

4.1. Matings Pruning

As already mentioned in Section 2.7.2, a mating, i.e. a set of connections, can be associated with any (path) connection tableau. However, this mapping is not injective in general. So one and the same mating may be associated with different tableaux. This means that the matings concept provides a more abstract view of the search space and enables us to group tableaux into equivalence classes. Under certain circumstances, it is not necessary to construct *all tableaux* in such a class but only *one representative.* In order to illustrate this, let us consider the set of propositional clauses

$$\{\neg P_1 \vee \neg P_2, \neg P_1 \vee P_2, P_1 \vee \neg P_2, P_1 \vee P_2\}.$$

As shown in Figure 7, the set has 4 closed regular connection tableaux with the all-negative start clause $\neg P_1 \vee \neg P_2$. If, however, the involved sets of connections are inspected, it turns out that the tableaux all have the same mating consisting of

6 connections. The redundancy contained in the tableau framework is that certain tableaux are *permutations* of each other corresponding to different possible ways of *traversing* a set of connections. Obviously, only one of the tableaux in such an equivalence class has to be considered.

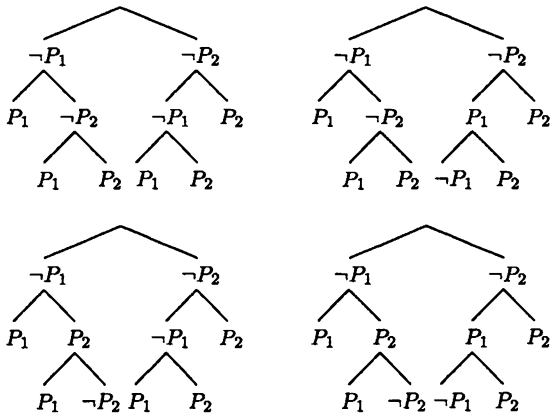


Figure 7: Four closed connection tableaux for the same spanning mating.

The question is, how exactly this redundancy can be avoided. A general line of development would be to store all matings that have been considered during the tableau search procedure and to ignore all tableaux which encode a mating which was already generated before. This approach would require an enormous amount of space. Based on preliminary work in [Letz 1993], a method was developed in [Letz 1998*b*] which can do with very little space and avoid the form of duplication shown in Figure 7. To comprehend the method, note that, in the example above, the source of the redundancy is that a certain connection can be used both in an extension step and in a reduction step. This causes the combinatorial explosion. The idea is now to block certain reduction steps by using an ordering $\prec$ on the occurrences of literals in the input set which has to be respected during the tableau construction, as follows. Assume, we want to perform a reduction step from a node N to an ancestor node N'. Let $N_1, \ldots, N_n$ be the node family below N'. The nodes $N_1, \ldots, N_n$ were attached by an extension step "into" a node complementary to N', say N_i. Now we do not permit the reduction step from N to N' if $N_i \prec N$ where the ordering $\prec$ is inherited from the literal occurrences in the input set to the tableau nodes. As can easily be verified, for any total ordering, in the example above, only one closed tableau can be constructed with this proviso. As shown in [Letz 1998*b*], using this method a super-exponential reduction of the search space can be achieved with almost no overhead.

On the other hand, there may be problems when combining this method with other search pruning techniques.

4.1.1. Matings pruning and strong connectedness

For instance, the method is not compatible with the condition of strong connectedness presented in Section 3.4. As a counterexample, consider the set of the four clauses given in Example 4.1.

4.1. EXAMPLE. $\{P \vee Q(a), P \vee \neg Q(a), \neg P \vee Q(a), \neg P \vee \neg Q(x)\}$.

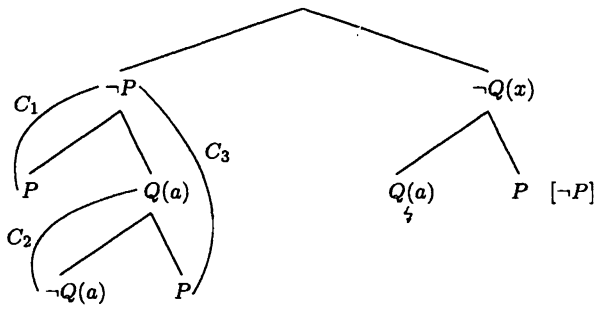


Figure 8: Deduction process for Example 4.1.

If we take the fourth clause, which is relevant in the set, as top clause, enter the first clause, then the second one by extension, and finally, perform a reduction step, then the closed subtableau on the left-hand side encodes the mating $\{C_1, C_2, C_3\}$. Now, any extension step at the subgoal labelled with $\neg Q(x)$ on the right-hand side immediately violates the strong connectedness condition. Therefore, backtracking is required up to the state in which only the top clause remains. Afterwards, the second clause must be entered, followed by an extension step into the first one. But now the mating pruning forbids a reduction step at the subgoal labelled with P, since it would produce a closed subtableau encoding the same mating $\{C_3, C_2, C_1\}$ as before. Since extension steps are impossible because of the regularity condition, the deduction process would fail and incorrectly report that the clause set is satisfiable.

4.2. Tableau Subsumption

A much more powerful application of the idea of subsumption between input clauses and tableau clauses consists in generalizing subsumption between clauses to subsumption between entire tableaux. For a powerful concept of subsumption between formula trees, the following notion of *formula tree contractions* proves helpful.

4.2. DEFINITION *((Formula) tree contraction).* A (formula) tree T is called a *contraction* of a (formula) tree T' if T' can be obtained from T by attaching n (formula) trees to n non-leaf nodes of T, for some $n \geq 0$.

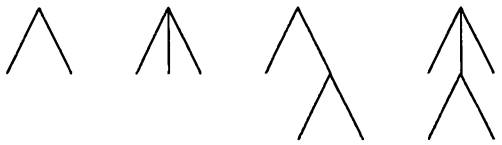


Figure 9: Illustration of the notion of tree contractions.

In Figure 9, the tree on the left is a contraction of itself, of the second and the fourth tree but not a contraction of the third one. Furthermore, the third tree is a contraction of the fourth one, which exhausts all contraction relations among these four trees. Now subsumption can be defined easily by building on the instance relation between formula trees.

4.3. DEFINITION *(Formula tree subsumption).* A formula tree T *subsumes* a formula tree T' if some formula tree contraction of T' is an instance of T.

The subsumption relation can be extended considerably by considering the branches as sets, or even more by employing the positive refinement technique discussed in [Baumgartner and Brüning 1997].

Since the exploitation of subsumption between entire tableaux has not enough potential for reducing the search space, we favour the following form of subsumption deletion.

4.4. DEFINITION *(Subsumption deletion).* For any pair of different nodes $\mathcal{N}$ and $\mathcal{N}'$ in a tableau search tree $\mathcal{T}$, if the goal tree of the tableau at $\mathcal{N}$ subsumes the goal tree of the tableau at $\mathcal{N}'$, then the whole subtree of the search tree with root $\mathcal{N}'$ is deleted from $\mathcal{T}$.

With subsumption deletion, a form of *global* redundancy elimination is achieved which is complementary to the purely tableau *structural* pruning methods discussed so far. In [Letz et al. 1994] it is shown that, for many formulae, cases of goal tree subsumption inevitably occur during proof search. Since this type of redundancy cannot be identified with tableau structure refinements like connectedness, regularity, or allies, methods for avoiding tableau subsumption are essential for achieving a well-performing model elimination proof procedure.

4.2.1. Tableau subsumption vs. regularity

Similar to the case of resolution where certain refinements of the *calculus*, i.e., restrictions of the resolution *inference rule*, become incomplete when combined with subsumption deletion, such cases also occur for refinements of tableau calculi. Formally, the compatibility with subsumption deletion can be expressed as follows.

4.5. DEFINITION *(Compatibility with subsumption).* A tableau calculus is said to be *compatible with subsumption* if any of its tableau search trees $\mathcal{T}$ has the following

property. For arbitrary pairs of nodes $\mathcal{N}$, $\mathcal{N}'$ in $\mathcal{T}$, if the goal tree S of the tableau T at $\mathcal{N}$ subsumes the subgoal tree S' of the tableau T' at $\mathcal{N}'$ and if $\mathcal{N}'$ dominates a success node, then $\mathcal{N}$ dominates a success node.

The (connection) tableau calculus is compatible with subsumption, but the integration of the regularity condition, for example, poses problems.

4.6. PROPOSITION. *The regular connection tableau calculus is incompatible with subsumption.*

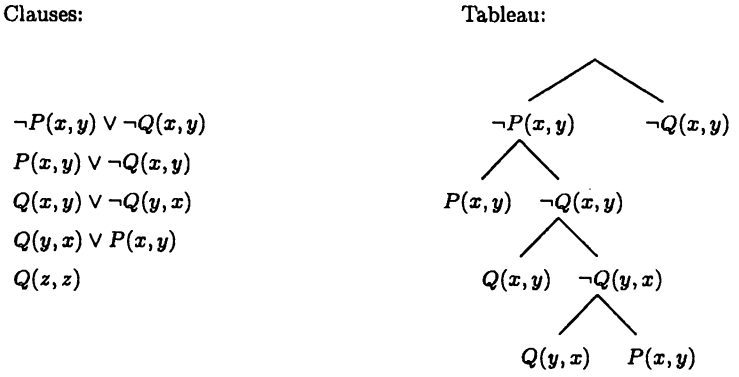


Figure 10: The incompatibility of subsumption and regularity.

PROOF. We use the unsatisfiable set of clauses displayed on the left of Figure 10. Taking the first clause as top clause and employing the depth-first left-most selection function, the first subgoal N labelled with $\neg P(x,y)$ can be solved by deducing the tableau T depicted on the right of the figure. Since N has been solved optimally, i.e., without instantiating its variables, the subgoal tree of T subsumes the subgoal trees of all other tableaux working on the solution of N. Hence, all tableaux competing with T can be removed by subsumption deletion. But T cannot be extended to a solved tableau, due to the regularity condition, the crucial impediment being that an extension step into $Q(z,z)$ is not permitted, since it would render the already solved subtableau on the left irregular. To obtain a formula in which subsumption is fatal for *any* top clause, one can employ the *duplication trick* used in [Letz et al. 1994]. □

The obvious problem with regularity is that it applies to entire tableaux whereas tableau subsumption considers only their subgoal trees. A straightforward solution therefore is to restrict regularity to subgoal trees, too. The respective weakening of regularity is called *subgoal tree regularity* . Similar incompleteness results can be achieved when combining tableau subsumption with tautology deletion. Here, a remedy is to ignore the non-tautology conditions of a tableau clause if some of its literals do not occur in the current subgoal tree. The same argument applies to the combination of tableau clause subsumption with formula tree subsumption.

4.3. Failure Caching

The observation that cases of subsumption inevitably will occur in practice suggests organizing the enumeration of tableaux in such a manner that cases of subsumption can really be detected. This could be achieved with a proof procedure which explicitly constructs competitive tableaux and thus investigates the search tree in a breadth-first manner. However, as already mentioned, the explicit enumeration of tableaux or goal trees is practically impossible. But when performing an implicit enumeration of tableaux by using iterative-deepening search procedures, only one tableau is in memory at any one time. This renders it very difficult to implement subsumption techniques in an adequate way. However, we discuss two methods by which a restricted concept of subsumption deletion can be implemented. The first paradigm employs *intelligent backtracking* [Neitz 1995]. This refinement of the standard Prolog backtracking technique searches the branch for the substitution that prevents a subgoal from being solved and backtracks to that point. Yet, this is difficult to do in the non-Horn case or in combination with other pruning mechanisms. The other paradigm is the use of so-called "failure caching" methods. The idea underlying this approach is to avoid the repetition of subgoal solutions which apply the same or a more special substitution to the respective branch. There are two approaches, one of which uses a permanent cache [Astrachan and Loveland 1991] and the other one uses a temporary cache [Letz et al. 1994]. We describe the latter method, which might be called "local failure caching" in more detail, because it turned out to be more successful in practice. Subsequently, we assume that only depth-first branch selection functions are used.

4.7. DEFINITION *(Solution -, failure substitution).* Given a tableau search tree $\mathcal{T}$ for a tableau calculus and a depth-first branch selection function, let $\mathcal{N}$ be a node in $\mathcal{T}$, T the tableau at $\mathcal{N}$ and N the selected subgoal in T.

1. If $\mathcal{N}'$ with tableau T' is a node in the search tree $\mathcal{T}$ dominated by $\mathcal{N}$ such that all branches through N in T' are closed, let $\sigma' = \sigma_1 \cdots \sigma_n$ be the composition of substitutions applied to the tableau T on the way from $\mathcal{N}$ to $\mathcal{N}'$. Then the substitution $\sigma = \{x/x\sigma' \in \sigma' : x \text{ occurs in } T\}$, i.e., the set of bindings in σ' with domain variables occurring in the tableau T, is called a *solution (substitution) of N at $\mathcal{N}$ via $\mathcal{N}'$.*
2. If $\mathcal{T}'$ is an initial segment of the search tree $\mathcal{T}$ containing no proof at $\mathcal{N}'$ or below it, then the solution σ is named a *failure substitution for N at $\mathcal{N}$ via $\mathcal{N}'$ in $\mathcal{T}'$.*

Briefly, when a solution of a subgoal N with a substitution σ does not permit to solve the rest of the tableau under a given size bound, then this solution substitution is a failure substitution. We describe how failure substitutions can be applied in a search procedure which explores tableau search trees in a depth-first manner employing structure sharing and backtracking.

4.8. DEFINITION *(Generation, application, and deletion of a failure substitution).* Let $\mathcal{T}$ be a finite initial segment of a tableau search tree.

1. Whenever a subgoal N selected in a tableau T at a search node $\mathcal{N}$ in $\mathcal{T}$ has been closed via (a sub-refutation to) a node $\mathcal{N}'$ in the search tree, then the computed solution σ is stored at the tableau node N. If the tableau at $\mathcal{N}'$ cannot be completed to a closed tableau in $\mathcal{T}'$ and the proof procedure backtracks over $\mathcal{N}'$, then σ is turned into a failure substitution.
2. In any alternative solution process of the tableau T below the search node $\mathcal{N}$, if a substitution $\tau = \tau_1 \cdots \tau_m$ is computed such that one of the failure substitutions stored at the node N is more general than τ, then the proof procedure immediately backtracks.
3. When the search node $\mathcal{N}$ (at which the tableau node N was selected for solution) is backtracked, then all failure substitutions at N are deleted.

	action	subgoals	substitution	fail.subs.
T_0	start step	$\neg P(x), \neg Q(y), \neg R(x)$	$\emptyset$	$\emptyset$
T_1	$P(a)$ entered	$\neg Q(y), \neg R(a)$	$\{x/a\}$	$\emptyset$
T_2	$Q(a)$ entered	$\neg R(a)$	$\{x/a, y/a\}$	$\emptyset$
	unification failure	$\neg R(a)$	$\{x/a, y/a\}$	$\emptyset$
	retract step 2	$\neg Q(y), \neg R(a)$	$\{x/a\}$	$\{y/a\}$
T_3	$Q(b)$ entered	$\neg R(a)$	$\{x/a, y/b\}$	$\{y/a\}$
	unification failure	$\neg R(a)$	$\{x/a, y/b\}$	$\{y/a\}$
	retract step 3	$\neg Q(y), \neg R(a)$	$\{x/a\}$	$\{y/a\}$
	retract step 1	$\neg P(x), \neg Q(y), \neg R(x)$	$\emptyset$	$\{x/a\}$
T_4	$P(x) \vee \neg Q(x)$ entered	$\neg Q(x), \neg Q(y), \neg R(x)$	$\{z/x\}$	$\{x/a\}$
T_5	$Q(a)$ entered	$\neg Q(y), \neg R(a)$	$\{z/a, x/a\}$	$\{x/a\}$
T_5	T_5 subsumed by T_1	$\neg Q(y), \neg R(a)$	$\{z/a, x/a\}$	$\{x/a\}$
	retract step 5	$\neg Q(x), \neg Q(y), \neg R(x)$	$\{z/x\}$	$\{x/a\}$
T_6	$Q(b)$ entered	$\neg Q(y), \neg R(b)$	$\{z/b, x/b\}$	$\{x/a\}$
T_7	$Q(a)$ entered	$\neg R(b)$	$\{z/b, x/b, y/a\}$	$\{x/a\}$
T_8	$R(b)$ entered		$\{z/b, x/b, y/a\}$	$\{x/a\}$

Figure 11: Proof search using failure substitutions.

4.9. EXAMPLE. In order to understand the mechanism, we show the method at work on a specific example.

Let S be the set of the five clauses

$$\neg P(x) \vee \neg Q(y) \vee \neg R(x), \quad P(a), \quad P(z) \vee \neg Q(z), \quad Q(a), \quad Q(b), \quad R(b).$$

A possible tableau construction for this clause set is documented in Figure 11. Assume, we start with the first clause in the set S and explore the corresponding

tableau search tree using a depth-first left-to-right subgoal selection function, just like in Prolog. Accordingly, in inference step 1, the subgoal $\neg P(x)$ is solved using the clause $P(a)$. With this substitution, the remaining subgoals cannot be solved. Therefore, when backtracking step 1, the failure substitution $\{x/a\}$ is stored at the subgoal $\neg P(x)$. The search pruning effect of this failure substitution shows up in inference step 5 when the failure substitution is more general than the computed tableau substitution, i.e., the tableau T_5 is subsumed by the tableau T_1. Without this pruning method, steps 2 to 4 would have to be repeated.

Note also that one has to be careful to delete failure substitutions under certain circumstances, as expressed in step 3 of the procedure. This provision applies, for example, to the failure substitution $\{y/a\}$ generated at the subgoal $\neg Q(y)$ after the retraction of inference step 2. When the choice point of this subgoal is completely exhausted, then $\{y/a\}$ has to be deleted. Otherwise, it would prevent the solution process of the tableau when, in step 7, this subgoal is again solved using the clause $Q(a)$.

As already noted in Example 4.9, when the failure substitution $\{x/a\}$ at $\neg P(x)$ is more general than an alternative solution substitution of the subgoal, then the subgoal tree of the former tableau subsumes the one of the tableau generated later. The described method preserves completeness for certain completeness bounds, as stated more precisely in the following proposition.

4.10. PROPOSITION. *Let $\mathcal{T}$ be the initial segment of a connection tableau search tree defined by some subgoal selection function and the depth bound (Section 2.5.2) or some clause-dependent depth bound (Section 2.5.4) with size limitation k. Assume a failure substitution σ has been generated at a node N selected in a connection tableau T at a search node $\mathcal{N}$ in $\mathcal{T}$ via a search node $\mathcal{N}'$ according to the procedure in Definition 4.8. If T_c is a closed connection tableau in the search tree $\mathcal{T}$ below the search node $\mathcal{N}$ and τ the composition of substitutions applied when generating T_c from T, then the failure substitution σ is not more general than τ.*

PROOF. Assume indirectly, that σ is more general than τ, i.e., $\tau = \sigma\theta$. Let S_c and S be the subtableaux with root N in T_c respectively in the connection tableau at $\mathcal{N}'$. Then, replacing S_c in T_c with $S\theta$ results in a closed connection tableau T_c'. Furthermore, it is clear that T_c' satisfies the size limit k of the completeness bound used. Now the connection tableau calculus is strongly independent of the selection function. Consequently, a variant T_c'' of the tableau T_c' must be contained in the search tree $\mathcal{T}$ below the search node $\mathcal{N}'$. Since T_c'' must be closed, too, there must be a closed tableau below the search node $\mathcal{N}'$. But this contradicts the assumption of σ being a failure substitution. □

The failure caching method described above has to be adapted when combined with other completeness bounds. While for the (clause-dependent) depth bounds exactly the method described above can be used, care must be taken when using the inference bound so as not to lose completeness. It may happen that a subgoal solution with solution substitution σ exhausts almost all available inferences so that

there are not enough inferences left for the remaining subgoals, and there might exist another, smaller solution tree of the subgoal with the same substitution which would permit the solution of the remaining subgoals. Then the failure substitution σ would prevent this. Accordingly, in order to guarantee completeness, the number of inferences needed for a subgoal solution has to be attached to a failure substitution, and only if the solution tree computed later is greater or equal to the one associated with σ, σ may be used for pruning.

Furthermore, when using failure caching together with structural pruning methods such as regularity, tautology deletion, or tableau clause subsumption, phenomena like the one discussed in Proposition 4.6 may lead to incompleteness. A remedy is to restrict the structural conditions to the subgoal tree of the current tableau. But even if this requirement is complied with, completeness may be lost as demonstrated by the following example.

4.11. EXAMPLE. Let S be the set of the seven clauses

$$\neg P(x,b)\vee\neg Q(x),\ P(x,b)\vee\neg R(x)\vee\neg P(y,b),\ P(a,z),\ P(x,b),\ R(a),\ Q(a),\ R(x).$$

Using the first clause as start clause and performing a Prolog-like search strategy, the clause $P(x,b) \vee \neg R(x) \vee \neg P(y,b)$ is entered from the subgoal $\neg P(x,b)$. Solving the subgoal $\neg R(x)$ with the clause $R(a)$ leads to a tableau structure violation (irregularity or tautology) when $\neg P(y,b)\{x/a\}$ is solved with $P(a,z)$. This triggers the creation of a failure substitution $\{x/a\}$ at the subgoal $\neg R(x)$. The alternative solution of $\neg R(x)$ (with $R(x)$) and of $\neg P(y,b)$ (with $P(a,z)$) succeeds, so that the subgoal $\neg P(x,b)$ in the top clause is solved with the empty substitution $\emptyset$. The last subgoal $\neg Q(x)$ in the top clause, however, cannot be solved using the clause $Q(a)$ due to the failure substitution $\{x/a\}$ at $\neg R(x)$. This initiates backtracking, and the solution substitution $\emptyset$ at the subgoal $\neg P(x,b)$ is turned into a failure substitution. As a consequence, any alternative solution of this subgoal will be pruned, so that the procedure does not find a closed tableau, although the set of clauses is unsatisfiable. The problem is that the first tableau structure violation encountered has mutated to a failure substitution $\{x/a\}$. The reason for the failure is that the substitution $\{x/a\}$ leading to success (i.e. a regular tableau) is blocked by the previous computation of a *different* tableau that runs into a regularity violation. One possible solution is to simply ignore the fatal failure substitution x/a when the respective node $\neg P(x,b)$ is solved. In a more general sense, this results in the following modification of Definition 4.8.

4.12. DEFINITION *(Generation, application, and deletion of failure substitutions).* Items 1 and 3 are as in Definition 4.8, item 2 has to be replaced as follows.

2. In any alternative solution process *of the subgoal* N below the search node $\mathcal{N}$, if a substitution $\tau = \tau_1 \cdots \tau_m$ is computed such that one of the failure substitutions stored at the node N is more general than τ, then the proof procedure immediately backtracks.

In other terms, the failure substitutions at a subgoal have to be deactivated when the subgoal has been solved. This restricted usage of failure substitutions for search pruning preserves completeness. It would be interesting to investigate which weaker restrictions on failure caching and the structural tableau conditions would still guarantee completeness. With the failure caching procedure described in Definition 4.12 a significant search pruning effect can be achieved, as confirmed by a wealth of experimental results [Letz et al. 1994, Moser et al. 1997].

4.3.1. Comparison with other methods

The *caching* technique proposed in [Astrachan and Stickel 1992] suggests to record the solutions of subgoals independently of the path contexts in which the subgoals appear. Then, cached solutions can be used for solving subgoals by *lookup* instead of *search*. In the special case in which no solutions for a cached subgoal exist, the cache acts in the same manner as the local failure caching mechanism. One difference is that failure substitutions take the path context into account and hence are compatible with subgoal tree regularity whereas the caching technique mentioned above is not. On the other hand, permanently cached subgoals without context have more cases of application than the temporary and context-dependent failure substitutions. The main disadvantage of the context-ignoring caching technique, however, is that its applicability is restricted to the Horn case. Note that the first aspect of the caching technique mentioned, that is replacing search by lookup, cannot be captured with a temporary mechanism as described above, since lookup is mainly effective for *different* subgoals whereas failure substitutions are merely used on different solutions of *one and the same* subgoal.

In [Loveland 1978] a different concept of subsumption was suggested for model elimination chains. Roughly speaking, this concept is based on a proof transformation which permits to ignore certain subgoals if the set of literals at the current subgoals is subsumed by an input clause. Such a replacement is possible, for example, if the remaining subgoals can be solved without reduction steps into their predecessors. In terms of tableaux, Loveland's subsumption *reduces* the current subgoal tree while our approach tries to *prune* it.

5. Shortening of Proofs

The analytic tableau approach has proven successful, both proof-theoretically and in the practice of automated deduction. It is well-known, however, since the work of Gentzen [Gentzen 1935] that the purely analytic paradigm suffers from a fundamental weakness, namely, the poor *deductive power*. That is, for very simple examples, the smallest tableau proofs may be extremely large if compared with proofs in other calculi. In this section, we shall present methods which can remedy this weakness and lead to significantly shorter proofs. The methods we consider are of two different types. First, we describe mechanisms that amount to adding additional inference rules to the tableau calculus. The mechanisms are centered around a controlled integration of the (backward) cut rule. Those mechanisms have

the widest application, since they already improve the behaviour of tableaux for propositional logic. Furthermore, we consider an improvement of tableaux which is first-order by its very nature, since it can be effective for tableaux with free variables only. It results from the fact that free variables in tableaux need not necessarily be treated as rigid. This results in a system in which the complexity of proofs can be significantly smaller than the so-called *Herbrand complexity*, which for a given set S of clauses is the complexity of the minimal unsatisfiable set of ground instances of clauses in S (see [Baaz and Leitsch 1992]).

5.1. *Factorization*

The *factorization* rule was introduced to the model elimination format in [Kowalski and Kuehner 1971] (see also [Loveland 1972]) and used in the connection calculus [Bibel 1987, Chapter III.6], but was applied to depth-first selection functions only due to format restrictions. On the general level of the (connection) tableau calculus, which permits arbitrary node selection functions, the rule can be motivated as follows. Consider a closed tableau containing two nodes N_1 and N_2 with the same literals as labels. Furthermore, suppose that all ancestor nodes of N_2 are also ancestors of N_1. Then, the closed tableau part T below N_2 could have been reused as a solution and attached to N_1, because all expansion and reduction steps performed in T under N_2 are possible in T under N_1, too. This observation leads to the introduction of *factorization* as an additional inference rule. Factorization permits to mark a subgoal N_1 as solved if its literal can be unified with the literal of another node N_2, provided that the set of ancestors of N_2 is a subset of the set of ancestors of N_1; additionally, the respective substitution has to be applied to the tableaux.

5.1. EXAMPLE. For any set $\{A_1, \ldots, A_n\}$ of distinct propositional atoms, let S_n denote the set of all 2^n clauses of the shape $L_1 \vee \ldots \vee L_n$ where $L_i = A_i$ or $L_i = \neg A_i$, $1 \leq i \leq n$.

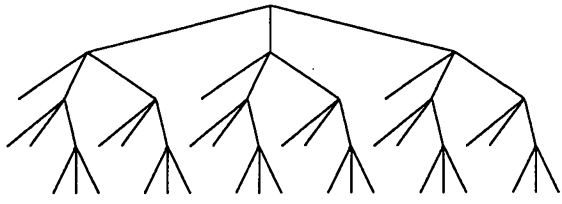


Figure 12: Tree structure of a minimal closed tableau for Example 5.1, $n = 3$.

As illustrated with Figure 12 the standard *cut-free* tableau calculi are intractable for this class of formulae [Letz 1993, Letz et al. 1994]. Used on the set of clauses

$$\{p \vee q, p \vee \neg q, \neg p \vee q, \neg p \vee \neg q\}$$

which denotes an instance of Example 5.1, for $n = 2$, factorization yields a shorter proof, as shown in Figure 13. Factorization steps are indicated by arcs. Obviously, in order to preserve soundness the rule must be constrained to prohibit solution cycles. Thus, in Figure 13 the factorization of subgoal N_4 on the right hand side with the node N_3 with the same literal on the left hand side is not permitted after the first factorization (node N_1 with node N_2) has been performed, because this would involve a reciprocal, and hence unsound, employment of one solution within the other. To avoid the cyclic application of factorization, tableaux have to be supplied with an additional factorization dependency relation.

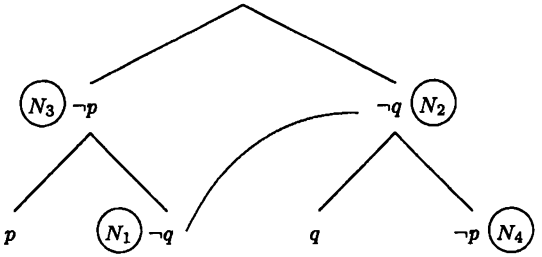


Figure 13: Factorization step in a connection tableau for Example 5.1, $n = 2$.

5.2. DEFINITION *(Factorization dependency relation).* A *factorization dependency relation on a tableau* T is a strict partial ordering $\prec$ on the tableau nodes ($N_1 \prec N_2$ means that the solution of N_2 depends on the solution of N_1).

5.3. DEFINITION *(Tableau factorization).* Given a tableau T and a factorization dependency relation $\prec$ on its nodes. First, select a subgoal N_1 with literal L and another node N_2 labelled with a literal K such that

1. there is a minimal unifier σ: $L\sigma = K\sigma$,
2. N_1 is dominated by a node N which has the node N_2 among its immediate successors, and
3. $N_3 \not\prec N_2$, where N_3 is the brother node of N_2 on the branch from the root down to and including N_1.[7]

Then mark N_1 as closed. Afterwards, modify $\prec$ by first adding the pair of nodes $\langle N_2, N_3 \rangle$ and then forming the transitive closure of the relation; finally, apply the substitution σ to the tableau. We say that the subgoal N_1 has been *factorized with* the node N_2. The tableau construction is started with an empty factorization dependency relation, and all other tableau inference rules leave the factorization dependency relation unchanged.

Applied to the example shown in Figure 13, when the subgoal N_1 is factorized with the node N_2, the pair $\langle N_2, N_3 \rangle$ is added to the previously empty relation $\prec$,

[7]Note that N_3 may be N_1 itself.

thus denoting that the solution of the node N_3 depends on the solution of the node N_2. After that, factorization of the subgoal N_4 with the node N_3 is not possible any more.

It is clear that the factorization dependency relation only relates brother nodes, i.e., nodes which have the same immediate predecessor. Furthermore, the applications of factorization at a subgoal N_1 with a node N_2 can be subdivided into two cases. Either, the node N_2 has been solved already, or the node N_2 or some of the nodes dominated by N_2 are not yet solved. In the second case we shall speak of an *optimistic* application of factorization, since the node N_1 is marked as solved *before* it is known whether a solution exists. Conversely, the first case will be called a *pessimistic* application of factorization.

Similar to the case of ordinary (connection) tableaux, if the factorization rule is added, the order in which the tableau rules are applied does not influence the structure of the tableau.

5.4. PROPOSITION (Strong node selection independency of factorization). *Any closed (connection) tableau with factorization for a set of clauses constructed with one selection function can be constructed with any other selection function.*

PROOF. See [Letz et al. 1994]. □

Switching from one selection function to another may mean that certain optimistic factorization steps become pessimistic factorization steps and vice versa. If we are working with subgoal trees, i.e., completely remove solved parts of a tableau, as done in the chain format of model elimination, then for all *depth-first* selection functions solely optimistic applications of factorization can occur. Also, the factorization dependency relation may be safely ignored, because the depth-first procedure and the removal of solved nodes render cyclic factorization attempts impossible. It is for this reason, that the integration approaches of factorization into model elimination or into the connection calculus have not mentioned the need for a factorization dependency relation. Note also that if factorization is integrated into the chain format of model elimination, then the mentioned strong node selection independency does not hold, since pessimistic factorization steps cannot be performed.

The addition of the factorization rule permits the generation of significantly smaller (connection) tableaux proofs. Thus, for the critical formula class given in Example 5.1, for which no polynomial tableau proofs exist (see Figure 12), there exist linear closed connection tableaux with factorization, as shown in Figure 14. With the factorization rule connection tableaux linearly simulate truth tables. In fact, the factorization rule can be considered as being a certain restricted version of the *cut rule*, which permits to add two new nodes labelled with arbitrary formulae F and $\neg F$, respectively, to any tableau branch. We speak of an *atomic cut* when F is an atomic formula.

5.5. PROPOSITION. *Atomic cut tableaux can linearly simulate tableaux with fac-*

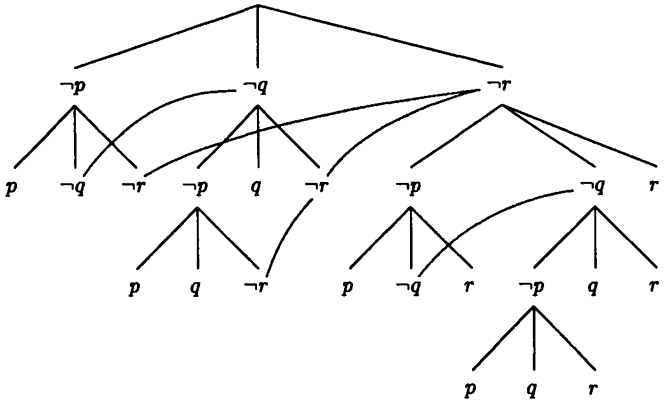


Figure 14: Linear closed connection tableau with factorization for Example 5.1, $n = 3$.

torization, and regular connection atomic cut tableaux can linearly simulate (connection) tableaux with factorization.

PROOF. We perform the simulation proof for atomic cut tableaux. Given a closed tableau with factorization, each factorization step of a node N_1 with a node N_2, both labelled with a literal L, can be simulated as follows. First, perform a cut step with $\sim L$ and L at the ancestor N of N_2, producing new nodes N_4 and N_5; thereupon, move the tableau part formerly dominated by N below N_4; then, remove the tableau part underneath N_2 and attach it to N_5; finally, perform reduction steps at N_1 and N_2. The simulation is graphically shown in Figure 15. □

While it is an open problem whether clausal tableaux with factorization can polynomially simulate atomic cut tableaux, *connection* tableaux with factorization cannot even polynomially simulate pure clausal tableaux (without atomic cut), as will be discussed in the next subsection.

5.2. The Folding Up Rule

The so-called ***folding up rule*** is an inference rule which, for connection tableaux, is stronger than factorization concerning deductive power. Folding up generalizes the *c-reduction* rule introduced to the model elimination format in [Shostak 1976]. In contrast to factorization, for which the pessimistic and the optimistic application do not differ concerning their deductive power, the shortening of proofs achievable with folding up results from its pessimistic nature. The theoretical basis of the rule is the possibility of extracting ***bottom-up lemmata*** from solved parts of a tableau, which can be used on other parts of the tableau (as described in [Loveland 1968]

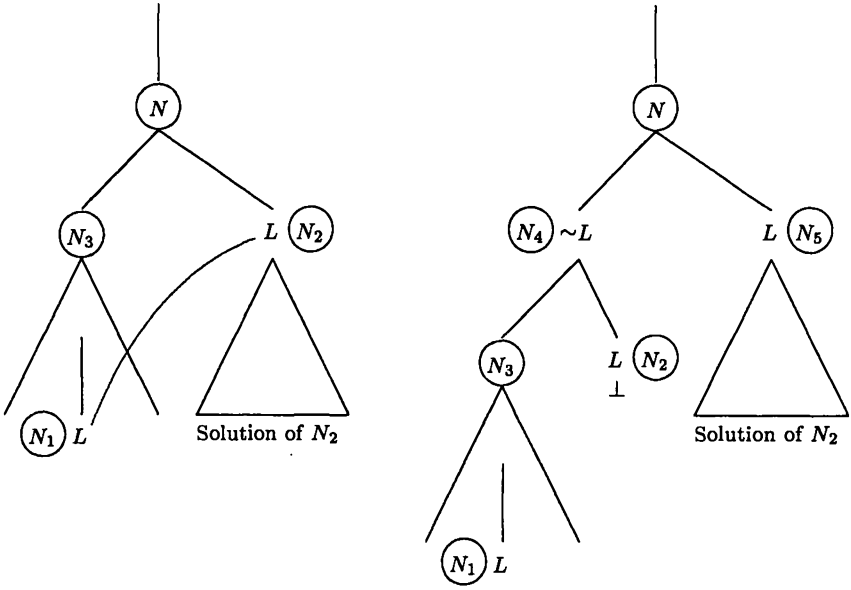


Figure 15: Simulation of factorization by cut rule applications.

and [Letz et al. 1992], or [Astrachan and Loveland 1991]). Folding up represents a particularly efficient realization of this idea.

We explain the rule with an example. Given the tableau displayed on the left of Figure 16, where the arrow points to the node at which the last inference step (a reduction step with the node 3 levels above) has been performed. With this step we have solved the dominating nodes labelled with the literals r and q. In the solutions of those nodes the predecessor labelled with p has been used for a reduction step. Obviously, this amounts to the derivation of two lemmata $\neg r \vee \neg p$ and $\neg q \vee \neg p$ from the underlying formula. The new lemma $\neg q \vee \neg p$ could be added to the underlying set and subsequently used for extension steps (this has already been described in [Letz et al. 1992]). The disadvantage of such an approach is that the new lemmata may be *non-unit* clauses, as in the example, so that extension steps into them would produce new subgoals, together with an unknown additional search space. The redundancy introduced this way can hardly be controlled.

With the folding up rule a different approach is pursued. Instead of adding lemmata of arbitrary lengths, so-called *context unit lemmata* are stored. In the discussed example, we may obtain two context unit lemmata:

$\neg r$, valid in the *(path)* context p, and
$\neg q$, valid in the context p.

Also, the memorization of the lemmata is not done by augmenting the input formula but *within* the tableau itself, namely, by "folding up" a solved node to the edge which dominates its solution context. More precisely, the folding up of a solved

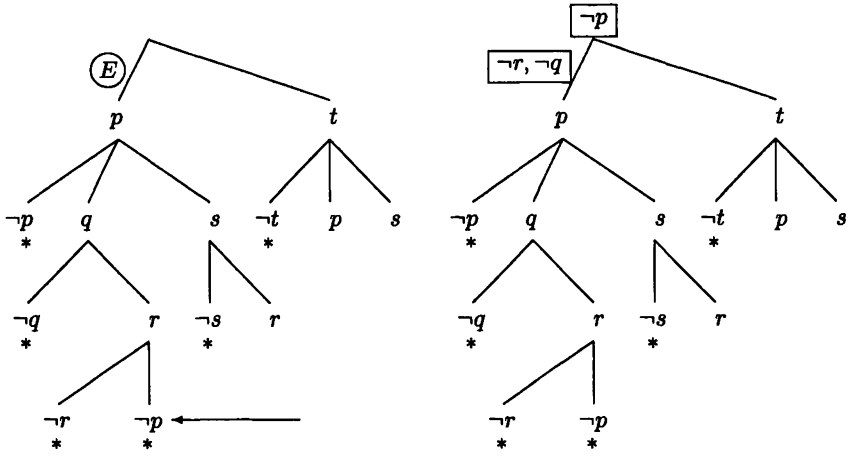


Figure 16: Connection tableau before and after folding up three times.

node N to an edge E means labelling E with the negation of the literal at N. Thus, in the example in Figure 16 the edge E above the p-node on the left-hand side of the tableau is successively labelled with the literals $\neg r$ and $\neg q$, as displayed on the right-hand side of Figure 16; lists of context-unit lemmata are shown as framed boxes. Subsequently, the literals in the boxes at the edges can be used for ordinary reduction steps. So, at the subgoal labelled with r a reduction step can be performed with the edge E, which was not possible before the folding up. After that, the subgoal s could also be folded up to the edge E, which we have not done in the figure, since after solving that subgoal the part below E is completely solved. But now the p-subgoal on the left is solved, and we can fold it up above the root of the tableau; since there is no edge above the root, we simply fold up *into* the root. This folding up step facilitates that the p-subgoal on the right can be solved by a reduction step.

The gist of the folding up rule is that only *unit* lemmata are added, so that the additionally imported indeterminism is not too large. Over and above that, the technique gives rise to a new form of pruning mechanism called *strong regularity*, which is discussed below. Furthermore, the folding up operation can be implemented very efficiently, since no renaming of variables is needed, as in a full lemma mechanism.

In order to be able to formally introduce the inference rule, we have to slightly generalize the notion of tableaux.

5.6. DEFINITION *((Marked) edge-labelled tableau).* A *(marked) edge-labelled tableau (E-tableau)* is just a tableau as introduced in the Definitions 2.1 and 2.5 with the only modifications that also the edges and the root node are labelled by the labelling function λ, namely, with lists of literals. Additionally, in every extension and reduction step, the closed branch is *marked* with the respectively used ances-

tor literal. The *path set* of a non-root node N in an E-tableau is the union of the sets of literals at the nodes dominating N and in the lists at the root and at the edges dominating the immediate predecessor of N.

5.7. DEFINITION *(E-tableau folding up).* Let T be an E-tableau, N a non-leaf node with literal L and the subtree below N be closed. The insertion position of the literal $\sim L$ is computed as follows. From the markings of all branches dominated by N select the set M of nodes which dominate N (M contains exactly the predecessor nodes on which the solution of N depends).

1. If M is empty or contains the root node only, then add the literal $\sim L$ to the list of literals at the root.
2. Otherwise, let N' be the deepest path node in M. Add the literal $\sim L$ to the list of literals at the edge immediately above N'.[8]

As an illustration, consider Figure 16, and recall the situation when the 'q'-node N on the left has been solved completely. The markings of the branches dominated by N are the 'r'-node below N and the 'p'-node above N. Consequently, $\neg q$ is added to the list at the edge E.

Additionally, the reduction rule has to be extended, as follows.

5.8. DEFINITION *(E-tableau reduction).* Given a marked E-tableau T, select a subgoal N with literal L, then,

1. either select a dominating node N' with literal $\sim K$ and a minimal unifier σ for L and K, and mark the branch with N',
2. or select a dominating edge or the root E with $\sim K$ in $\lambda(E)$ and a minimal unifier σ for L and K; then mark the branch with the node immediately below the edge or with the root, respectively.

Finally, apply the substitution σ to the literals in the tableau.

The *tableau* and the *connection tableau calculus with folding up* result from the ordinary versions by working with edge-labelled tableaux, adding the folding up rule, substituting the old reduction rule by the new one, starting with a root labelled with the empty list, and additionally labelling all newly generated edges with the empty list. Subsequently, we will drop the prefix 'E-' and simply speak of 'tableaux'.

The soundness of the folding up operation is expressed in the following proposition.

5.9. PROPOSITION (Soundness of folding up). *Let N be any subgoal with literal L in a marked tableau T, P the path set of N, and S a set of clauses. Suppose T' is any tableau deduced from T incorporating folding up steps and employing only clauses from S in the intermediate extension steps. Then, for the new path set P' of N in T': $P \cup S$ logically implies P'.*

[8]The position of the inserted literal exactly corresponds to the *C-point* in the terminology used in [Shostak 1976].

PROOF. The proof is by induction on the number n of folding up steps between T and T'. The base case for $n = 0$ is trivial, since $P' = P$. For the induction step, let $P' = P^n$ be the path set of N after the n-th folding up step inserting a literal, say L', into the path above N. This step was the consequence of solving a literal $\sim L'$ with clauses from S and path assumptions from P^{n-1}, i.e., the path set of N before the n-th folding up step. This means that $P^{n-1} \cup S \cup \{\sim L'\}$ is unsatisfiable. Now, by the induction assumption, $P \cup S \models P^{n-1}$. Consequently, $P \cup S \models P^{n-1} \cup \{L'\} = P'$. □

It is obvious that if a depth-first selection function is used and N is not yet solved in T', then all clauses in S are tableau clauses dominated by N or a brother node of N. This knowledge is used in the proof of Theorem 6.11 below.

In [Letz et al. 1994], it is proven that, for connection tableaux, the folding up rule is properly stronger concerning its deductive power than the factorization rule. In fact, the formula class used in this proof can also be used to show that connection tableaux with factorization cannot even polynomially simulate the pure clausal tableau calculus (without the atomic cut rule).

5.10. PROPOSITION. *Connection tableaux with factorization cannot polynomially simulate connection tableaux with folding up.*

PROOF. See [Letz et al. 1994]. □

Conversely, for a certain class of selection functions, the polynomial simulation in the other direction is possible.

5.11. PROPOSITION. *For arbitrary depth-first selection functions, (connection) tableaux with folding up linearly simulate (connection) tableaux with factorization.*

PROOF. Given any closed (connection) tableau T with factorization, let $\prec$ be its factorization dependency relation. By the strong node selection independency of factorization (Proposition 5.4), T can be constructed with any selection function. We consider a construction $C = (T_0, \ldots, T_m, T)$ of T with a depth-first selection function ϕ which respects the partial order of the factorization dependency relation $\prec$, i.e., for any two nodes N_1, N_2 in the tableau, $N_1 \prec N_2$ means that N_1 is selected before N_2; such a selection function exists since $\prec$ solely relates brother nodes. The deduction process C can directly be simulated by the (connection) tableau calculus with folding up, as follows. Using the same selection function ϕ, any expansion (extension) and reduction step in C is simulated by an expansion (extension) and reduction step. But, whenever a subgoal has been completely solved in the simulation, it is folded up. Since in the original deduction process, due to the pessimistic application of factorization, the factorization of a node N_2 with a node N_1 (with literal L) involves that N_1 has been solved before, in the simulation the literal L must have been folded up before. Now, *any* solved node can be folded up at least one step, namely, to the edge E above its predecessor (or into the root). Since E

(or the root) dominates N_1, the original factorization step can be simulated by a reduction step. The simulation of factorization by folding up is graphically shown in Figure 17. □

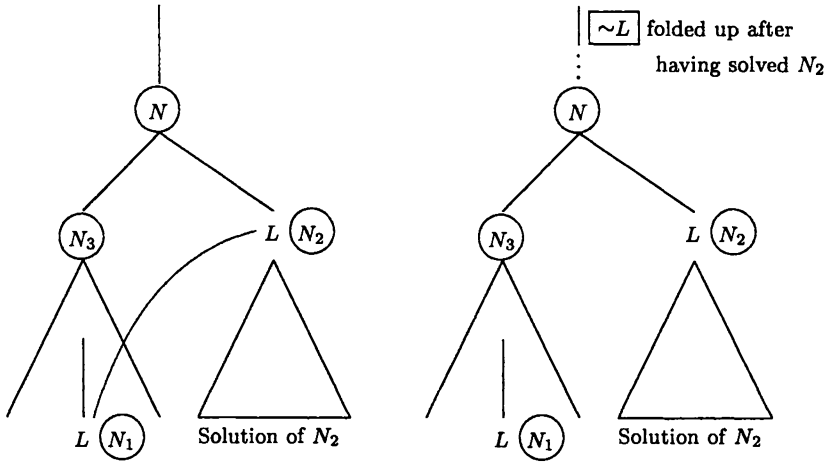


Figure 17: Simulation of factorization by folding up.

Finally, we show that the folding up rule, though properly more powerful than factorization, is still a hidden version of the cut rule.

5.12. PROPOSITION. *Atomic cut tableaux and atomic cut regular connection tableaux linearly simulate (connection) tableaux with folding up.*

PROOF. We perform the simulation proof for atomic cut tableaux. Given a tableau derivation with folding up, each folding up operation at a node N_0 adding the negation $\sim L$ of the literal L at a solved node to the label of an edge above a node N (or to the root), can be simulated as follows. Perform a cut step at the node N with the atom of L as the cut formula, producing two new nodes N_1 and N_2 labelled with L and $\sim L$, respectively; shift the solution of L from N_0 below the node N_1 and the part of the tableau previously dominated by N below its new successor node N_2; finally, perform a reduction step at the node N_0. It is obvious that the unmarked branches of both tableaux can be injectively mapped to each other such that all pairs of corresponding branches contain the same leaf literals and the same sets of path literals, respectively. The simulation is graphically shown in Figure 18. □

Connection tableaux with folding up can linearly simulate atomic cut tableaux. This is a straightforward corollary of a result proven in [Mayr 1993].

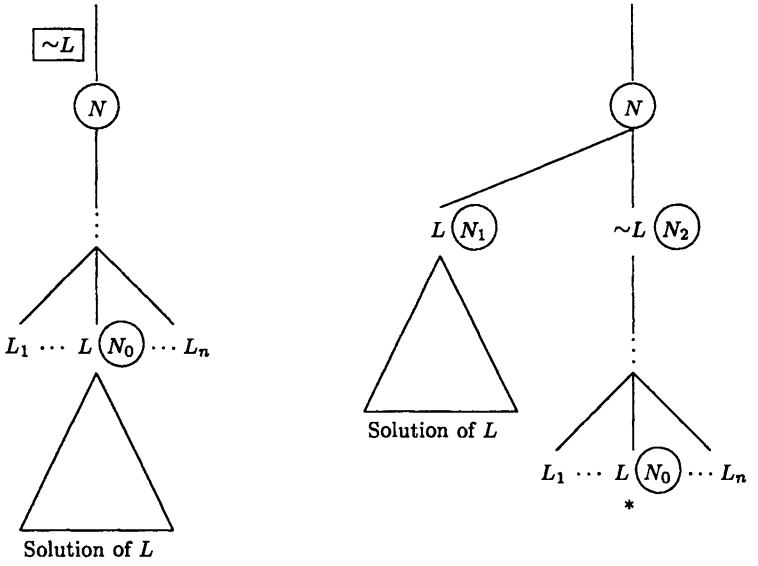


Figure 18: Simulation of folding up by the cut rule.

5.3. The Folding Down Rule

The simulation of factorization by folding up also shows how a restriction of the folding up rule can be defined which permits an *optimistic* labelling of edges. If a strict linear (dependency) ordering is defined on the successor nodes $N_1, \ldots, N_m$ of any node, then it is permitted to label the edge leading to any node N_i, $1 \leq i \leq m$, with the set of the negations of the literals at all nodes which are smaller than N_i in the ordering. We call this operation the *folding down* rule. The folding down operation can also be applied incrementally, as the ordering is completed to a linear one. The folding down rule is sound, since it can be simulated by the cut rule plus reduction steps, as illustrated in Figure 19. The rule is a very simple and efficient way of *implementing* factorization. Over and above that, if the literals on the edges are also considered as path literals in the regularity test, an additional search space reduction can be achieved this way which is discussed in the next section. It should be noted that it is very difficult to identify this refinement in the factorization framework. There, it is normally formulated in a restricted version, namely, as the condition that the set of literals at the *subgoals* of a tableau need to be consistent.

5.13. PROPOSITION. *(Regular) (connection) tableaux with folding down and (regular) (connection) tableaux with factorization linearly simulate each other.*

PROOF. See [Letz et al. 1994]. □

5.14. REMARK. Folding down is essentially *Prawitz reduction* [Prawitz 1960].

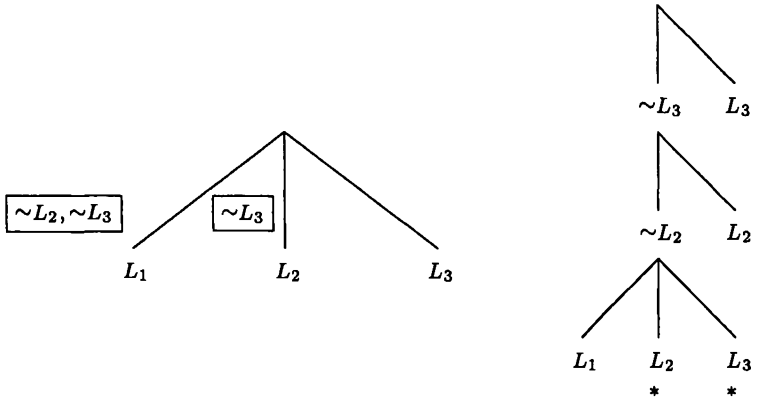


Figure 19: Simulation of folding down by cut.

Hence, by the above proposition, Prawitz reduction and factorization have the same deductive power when added to the connection tableau calculus, and Prawitz reduction is properly weaker than folding up and atomic cut.

5.4. Universal and Local Variables

The improvement of the tableau rules that we investigate in this part is of a quantificational nature. It deals with the problem that normally free variables in tableaux are considered as rigid, i.e., each as a place holder for an unknown ground term. Accordingly, every occurrence of a literal in a tableau can be used in one instance only. The simple reason is that tableau branches are disjunctively connected and the universal quantifier does not distribute over disjunctions. Under certain circumstances, however, such a distribution is possible and the respective variables may be read as universally quantified on the branch. The formulae containing such variables can then be used in different instances concerning the universal variables contained, which can permit a significant shortening of proofs. A very general definition of so-called *universal variables* is formulated in [Beckert and Hähnle 1992, Beckert 1998, Beckert and Hähnle 1998] as well as in [Hähnle 2001] (Chapter 3 of this Handbook). In its final reading, a variable x in a formula F on a branch of a general free-variable tableau T is *universal* if the formula $\forall x F$ may be added to the tableau branch and the formula of the resulting tableau is logically implied by the formula of T.

5.4.1. Local variables

Obviously, this definition is too general to be of any practical use, since this property of a variable is undecidable. One simple sufficient condition for being universal which applies to the case of clauses is that a variable occurs in only one literal of

a clause. For connection tableaux, a proof shortening effect can only be achieved this way if this occurs in non-unit clauses, since unit clauses can be freely used in new instances without producing new open branches. Unfortunately, such non-unit clauses do rarely occur in practice. Consequently, in [Letz 1998*a*, Letz 1999], the notion of local variables has been developed, which has more applications in connection tableaux.

5.15. DEFINITION *(Local variable).* Given an open tableau branch B of a clausal tableau, if a variable x occurs on B and on no other open branch of the tableau, then x is called *local (to B).*

Obviously, every local variable is universal. How can this property of a variable be utilized in detail? If a variable x in a formula F is universal, then the formula $\forall x F$ could be added to the branch without affecting the soundness of the calculus. As a matter of fact, this *generalization rule* is merely of a theoretical interest. Since we are only interested in calculi performing atomic branch closure, it is clear that the new formulae have to be decomposed by the γ-rule of free-variable tableaux, thus producing a variant of the formula F in which the variable x is renamed. And, since in connection tableaux instantiations are only performed in inference steps closing a branch, one can perform the generalization implicitly, exactly at that moment. This naturally leads to a generalized version of the unification rule.

5.16. DEFINITION *(Local unification rule).* Let K and L with universal variables $x_1, \ldots, x_n$ and $y_1, \ldots, y_n$ respectively be two literals to be unified in an inference step in a tableau T. Obtain K' by replacing all occurrences of $x_1, \ldots, x_n$ in K with distinct new variables not occurring in T; *afterwards* obtain L' by doing the same for L (i.e., perform no simultaneous replacement in K *and* L). If there is a (minimal) unifier for K' and L', apply σ to the tableau and close the respective branch.[9] If the set of universal variables is determined by the locality condition, we speak of the *local unification rule.*

The *local extension rule* and the *local reduction rule* are obtained from the standard rules by replacing standard unification with local unification.

The proof shortening effect of the local closure rule can be demonstrated by considering the set of the two clauses $R(x) \vee R(f(x))$ and $\neg R(x) \vee \neg R(f(f(x)))$, and the closed tableau shown in Figure 1. With the local unification rule one can obtain the significantly smaller closed tableau displayed in Figure 20. Assume the tableau construction is performed using a left-most branch selection function. The crucial difference occurs when the entire left subtree has been closed and the right-most subgoal $\neg R(f(f(f(x))))$ is extended using the first input clause with the first literal as entry literal. Since the variable x in the subgoal is local, it is renamed to, say y, which eventually leads to attachment of the tableau clause $R(f(f(f(y))))$

[9]Note that this rule is more powerful than the extension of unification defined in [Beckert and Hähnle 1998]. For example, a branch containing two formulae $P(x)$ and $\neg P(f(x))$ with universal variable x may be closed.

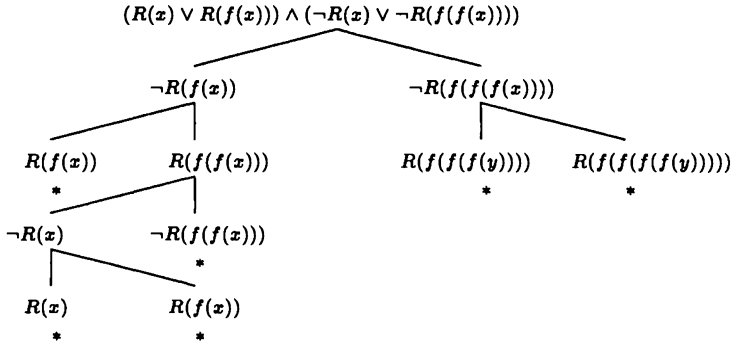


Figure 20: Closed clausal tableau with local unification rule.

and $R(f(f(f(f(y)))))$ and the closure of the left-most new branch. The remaining subgoal $R(f(f(f(f(y)))))$ can then be closed by a (local) reduction step.

Note that the tableau displayed in Figure 20 has no closed ground instance, in contrast to the tableau calculi developed so far that have the *ground projection property* [Baaz and Leitsch 1992], i.e., when substituting ground terms for the variables in a tableau proof for a set of clauses S, the resulting tableau remains closed and all tableau clauses are instances of clauses in S. The modified tableau system with the local unification rule permits to build refutations that are smaller than the so-called Herbrand complexity of the input, which normally is a lower bound to any standard tableau proof; the *Herbrand complexity* of an unsatisfiable set of clauses S is the complexity of the smallest unsatisfiable set of ground instances of clauses in S. It is evident, however, that tableau calculi containing the local unification rule are not independent of the branch selection function. If we would select the subgoals on the right before complete closure of the left subtree, x would not be local and hence the proof shortening effect would be blocked. Consequently, with the local unification rule, the order in which branches are selected can influence the size of minimal tableau proofs. Note that the same holds for the folding up rule discussed in Section 5.2.

6. Completeness of Connection Tableaux

In this section we will provide completeness proofs for the most important of the new calculi.

6.1. Structurally Refined Connection Tableaux

First, we consider the case of connection tableaux with the structural refinements of regularity, strong connectedness, and the use of relevance information, which were

mentioned in Section 3. Since the path connectedness condition is less restrictive, it suffices to consider the full connectedness condition. Unfortunately, we cannot proceed as in the case of free-variable tableaux or general clausal tableaux. Since the connection tableau calculus lacks proof confluence, the standard completeness proof using Hintikka sets cannot be applied. Instead, an entirely different approach for proving completeness will be used. The proof consists of two parts. In the first part, we demonstrate the completeness for the case of ground formulae—this is the interesting part of the proof. In the second part, this result is lifted to the first-order case by a standard proof technique. Beforehand, we need some additional terminology.

6.1. DEFINITION *(Strengthening).* The *strengthening* of a set of clauses S *by* a set of literals $P = \{L_1, \ldots, L_n\}$, written $P \rhd S$, is the set of clauses obtained by first removing all clauses from S containing literals from P and afterwards adding the n unit clauses $L_1, \ldots, L_n$.

6.2. EXAMPLE. For the set of propositional clauses $S = \{p \vee q, p \vee s, \neg p \vee q, \neg q\}$, the strengthening $\{p\} \rhd S$ is the set of clauses $\{p, \neg p \vee q, \neg q\}$.

Clearly, every strengthening of an unsatisfiable set of clauses is unsatisfiable. In the ground completeness proof, we will make use of the following further property.[10]

6.3. LEMMA (Strong Mate Lemma). *Let S be an unsatisfiable set of ground clauses. For any literal L contained in any relevant clause c in S there exists a clause c' in S such that*

(i) c' contains $\sim L$,

(ii) every literal in c' different from $\sim L$ does not occur complemented in c, and

(iii) c' is relevant in the strengthening $\{L\} \rhd S$.

PROOF. From the relevance of c follows that S has a minimally unsatisfiable subset S_0 containing c; every formula in S_0 is essential in S_0. Hence, there is an interpretation $\mathcal{I}$ for S_0 with $\mathcal{I}(S_0 \setminus \{c\}) = \text{true}$ and $\mathcal{I}(c) = \text{false}$, hence $\mathcal{I}(L) = \text{false}$. Define another interpretation $\mathcal{I}'$ by setting $\mathcal{I}'(L) = \text{true}$ and otherwise $\mathcal{I}' = \mathcal{I}$. Then $\mathcal{I}'(c) = \text{true}$. The unsatisfiability of S_0 guarantees the existence of a clause c' in S_0 with $\mathcal{I}'(c') = \text{false}$. We prove that c' meets the conditions (i) – (iii). First, the clause c' must contain the literal $\sim L$ and not the literal L, since otherwise $\mathcal{I}(c') = \text{false}$, which contradicts the selection of $\mathcal{I}$, hence (i). Additionally, for any literal L' in c' different from $\sim L$: $\mathcal{I}(L') = \mathcal{I}'(L') = \text{false}$. As a consequence, L' cannot occur complemented in c, since otherwise $\mathfrak{I}(c) = \text{true}$; this proves (ii). Finally, the essentiality of c' in S_0 entails that there exists an interpretation $\mathcal{I}''$ with $\mathcal{I}''(S_0 \setminus \{c'\}) = \text{true}$ and $\mathcal{I}''(c') = \text{false}$. Since $\sim L$ is in c', $\mathcal{I}''(L) = \text{true}$. Therefore, c' is essential in $S_0 \cup \{L\}$, and also in its unsatisfiable subset $\{L\} \rhd S_0$. From this conclusion and

[10] In terms of resolution, it expresses the fact that, for any literal L in a clause c that is relevant in a clause set S, there exists a non-tautological resolvent "over" L with another relevant clause in S.

from the fact that $\{L\} \rhd S_0$ is a subset of $\{L\} \rhd S$ follows that c' is relevant in $\{L\} \rhd S$. □

6.4. PROPOSITION (Ground completeness of regular strong connection tableaux). *For any finite unsatisfiable set S of ground clauses and for any clause c which is relevant in S, there exists a closed regular strong connection tableau for S with top clause c.*

PROOF. Let S be a finite unsatisfiable set of ground clauses and c any relevant clause in S. A closed regular strong connection tableau T for S with top clause c can be constructed from the root to its leaves via a sequence of intermediate tableaux as follows. Start with a tableau consisting simply of c as the top clause. Then iterate the following non-deterministic procedure, as long as the intermediate tableau is not yet closed.

> Choose an arbitrary open leaf node N in the current tableau with literal L. Let c be the tableau clause of N and let $P = \{L_1, \ldots, L_m, L\}$, $m \geq 0$, be the set of literals on the path from the root up to the node N. Then, select any clause c' which is relevant in $P \rhd S$, contains $\sim L$, is strongly connected to c, and does not contain literals from the path $\{L_1, \ldots, L_m, L\}$; perform an expansion step with c' at the node N.

First, note that, evidently, the procedure admits solely the construction of regular strong connection tableaux, since in any expansion step the attached clause contains the literal $\sim L$, no literals from the path to its parent node (regularity), nor does c' contain a literal different from $\sim L$ which occurs complemented in c. Due to regularity, there can be only branches of finite length. Consequently, the procedure must terminate, either because every leaf is closed, or because no clause c' exists for expansion which meets the conditions stated in the procedure. We prove that the second alternative does never occur, since for any *open* leaf node N with literal L there exists such a clause c'. This will be demonstrated by induction on the node depth. The induction base, $n = 1$, is evident, by the Strong Mate Lemma (6.3). For the step from n to $n+1$, with $n \geq 1$, let N be an open leaf node of tableau depth $n+1$ with literal L, tableau clause c, and with a path set $P \cup \{L\}$ such that c is relevant in $P \rhd S$, the induction assumption. Let S_0 be any minimally unsatisfiable subset of $P \rhd S$ containing c, which exists by the induction assumption. Then, by the Strong Mate Lemma, S_0 contains a clause c' which is strongly connected to c and contains $\sim L$. Since no literal in $P' = P \cup \{L\}$ is contained in a non-unit clause of $P' \rhd S$ and because N was assumed to be open, no literal in P' is contained in c' (regularity). Finally, since S_0 is minimally unsatisfiable, c' is essential in S_0; therefore, c' is relevant in $P' \rhd S$. □

The second half of the completeness proof uses a standard lifting argument.

6.5. LEMMA (Lifting Lemma). *Let S and S' be two sets of clauses such that every clause in S' is an instance of a clause in S. Let furthermore $T'_0, \ldots, T'_n$ be any sequence of successive (regular) (path) (connection) tableaux for S'. Then there*

exists a sequence $T_0, \ldots, T_n$ of successive (regular) (path) (connection) tableaux for the set S such that T'_n is an instance of T_n, i.e., $T'_n = T_n\sigma$, for some substitution σ, and, for every branch B' in T'_n, B' is closed if and only if its corresponding branch B in T_n is closed.

PROOF. The proof is straightforward by induction on the length of the construction sequence. The induction base, $n = 1$, is trivial. For the induction step, let T'_{n+1} be obtained from T'_n by applying on a branch B' with subgoal N' either (1) a closure step or (2) an expansion or (3) a (path) extension step using a clause from S' with matrix c'. Now let B with subgoal N be the branch in T_n corresponding to B'. In case (1), N' must have a complementary ancestor node. Select one, say N'_a, and let N_a be its corresponding node in T_n. Since, by the induction assumption, T'_n is an instance of T_n, there exists a (minimal) unifier τ for the literal at N and the complement of the literal at N_a. Obtain T_{n+1} by applying τ to T_n and closing B. In case (2), select a clause c from S such that $c\tau = c'$, for some substitution τ (such a clause exists by assumption) and obtain T_{n+1} by performing an expansion step using c at B. Case (3) is just a combination of (2) and (1), because a (path) extension step is an expansion step followed by a certain closure step. In all cases, the same branches are closed in both tableaux and T'_{n+1} is an instance of T_{n+1}. Therefore regularity is preserved, since any tableau which has a regular instance must also be regular. □

6.6. THEOREM. *For any unsatisfiable set S of clauses, there exists a closed regular connection tableau.*

PROOF. First, by Herbrand's completeness theorem, there exists a finite unsatisfiable set S' of ground instances of clauses in S. Let T' be a closed regular ground connection tableau for S' which exists by Proposition 6.4. The Lifting Lemma then guarantees the existence of a closed regular connection tableau T' for S. □

6.2. Enforced Folding and Strong Regularity

The folding up operation has been introduced as an ordinary inference rule which, according to its indeterministic nature, may or may not be applied. Alternatively, we could have defined versions of the (connection) tableau calculi with folding up in which any solved node *must* be folded up immediately after it has been solved. It is clear that whether folding up is performed freely, as an ordinary inference rule, or in an enforced manner, the resulting calculi are not different concerning their deductive power, since the folding up operation is a monotonic operation which does *not decrease* the inference possibilities. But the calculi differ with respect to their search spaces, since by treating the folding up rule just as an ordinary inference rule, which may or may not be applied, an additional and absolutely useless form of indeterminism is imported. Consequently, the folding up rule should not be introduced as an additional inference rule, but as a tableau operation to be

performed immediately after the solution of a subgoal. The resulting calculi will be called the *(connection) tableau calculi with enforced folding up.*

The superiority of the enforced folding up versions over the unforced ones also holds if the regularity restriction is added, according to which no two *nodes* on a branch can have the same literal as label. But the manner in which the folding up and the folding down rules have been introduced raises the question whether the regularity condition might be sharpened and extended to the literals in the labels of the edges as well. It is clear that such an extension of regularity is not compatible with folding up, since any folding up operation causes the respective closed branch to immediately violate the extended regularity condition. A straightforward remedy is to apply the extended condition to the *subgoal trees* of tableaux only.

6.7. DEFINITION *(Strong regularity).* A tableau T is called *strongly regular* if it is regular and no literal at a subgoal N of T is contained in the path set of N.

When the strong regularity condition is imposed on the connection tableau calculus with enforced folding up, then a completely new calculus is generated which is no extension of the regular connection tableau calculus, that is, not every proof in the regular connection tableau calculus can be directly simulated by the new calculus. This is because after the application of a folding up operation certain inference steps previously possible for other subgoals may then become impossible. A folding up step may even lead to an immediate failure of the extended regularity test, as demonstrated below. Since the new calculus is no extension of the regular connection tableau calculus and therefore the completeness result for regular connection tableaux cannot be applied, its completeness is not to be taken for granted. In fact, the new calculus is *incomplete* for some selection functions.

6.8. PROPOSITION. *There is an unsatisfiable set S of ground clauses and a selection function ϕ such that there is no refutation for S in the strongly regular connection tableau calculus with enforced folding up.*

6.9. EXAMPLE. The set S consisting of the clauses

$\neg p \vee \neg s \vee \neg r,$	$p \vee s \vee r,$	$\neg q \vee r,$	$q \vee \neg r,$
$\neg p \vee t \vee u,$	$p \vee \neg t \vee \neg u,$	$\neg q \vee s,$	$q \vee \neg s,$
		$\neg q \vee t,$	$q \vee \neg t,$
		$\neg q \vee u,$	$q \vee \neg u.$

PROOF. Let S be the set of clauses given in Example 6.9, which is minimally unsatisfiable. The non-existence of a refutation with the top clause $p \vee s \vee r$ for a certain unfortunate selection function ϕ is illustrated in Figure 21. There, the tableau extension steps are shown using black lines while the grey lines indicate the different alternatives for such extensions. If ϕ selects the s-node, then two alternatives exist for extension. For the one on the left-hand side, if ϕ shifts to the p-subgoal above and completely solves it in a depth-first manner, then the enforced folding up of the p-subgoal immediately violates the strong regularity, indicated

with a '↯' below the responsible $\neg p$-subgoal on the left. Therefore, only the second alternative on the right-hand side may lead to a successful refutation. Following the figure, it can easily be verified that for any refutation attempt there is a selection possibility which either leads to extension steps which immediately violate the old regularity condition or produce subgoals labelled with $\neg p$ or $\neg r$. In those cases, the selection function always shifts to the respective p- or r-subgoal in the top clause, solves it completely and folds it up afterwards, this way violating the strong regularity. Consequently, for such a selection function, there is no refutation with the given top clause. The same situation holds for any other top clause selected from the set. This can be verified in a straightforward though tedious manner. □

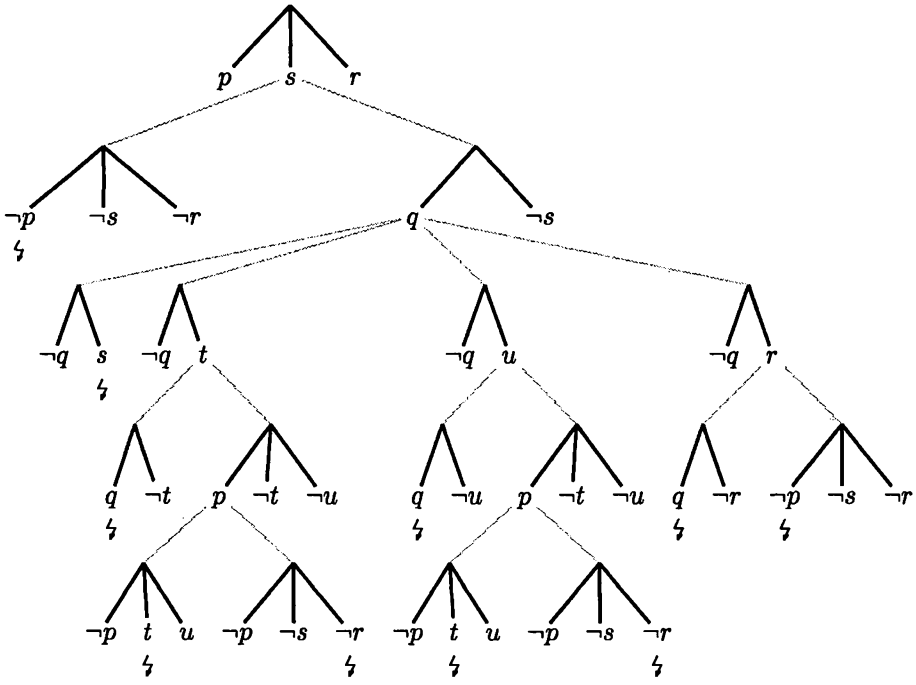


Figure 21: Incompleteness for free selection functions of the strongly regular connection tableau calculus with enforced folding up.

This result demonstrates that there is a trade-off between optimal selection functions and structural restrictions on tableaux. It would be interesting to investigate under which weakenings of the strong regularity the completeness for arbitrary selection functions might be obtained.

If we restrict ourselves to depth-first selection functions, however, the calculus is complete, as shown next.

We are now going to present completeness proofs for two calculi, for strongly regular connection tableaux with enforced folding up using depth-first selection

functions and for strongly regular connection tableaux with enforced folding down using arbitrary selection functions. The completeness proofs are based on the following non-deterministic procedure for generating connection tableaux which is similar to the one used in the proof of proposition 6.4 However, in the following procedure a mapping α is carried along as an additional control structure which, upon selection of a subgoal N, associates with N a specific subset $\alpha(N)$ of the input clauses.

6.10. PROCEDURE. Let S_0 be a finite unsatisfiable set of ground clauses, c_0 any clause which is relevant in S_0, and ϕ any subgoal selection function. First, perform a start step with the clause c_0 at the root N_0 of a one-node tableau, select a subset S of S_0 with c_0 being essential in S, and set $\alpha(N_0) = S$. Then, as long as applicable, iterate the following procedure.

Let N be the subgoal selected by ϕ, P the path set of N, L the literal and c the tableau clause at N, and $S = \alpha(N')$ where N' is the immediate predecessor node of N.

If $\sim L \in P$, perform a reduction step at N.

Otherwise, perform an extension step at N with a clause c' in S such that c' is relevant in $(P \cup \{L\}) \rhd S$, select a subset S' of S with c' being essential in the set $(P \cup \{L\}) \rhd S'$, and set $\alpha(N) = S'$.

Additionally, depending on the chosen extension of the calculus, enforced folding up or folding down operations need to be applied.

It suffices to perform the completeness proofs for the ground case, since the lifting to the first-order case is straightforward, using the Lifting Lemma (6.5).

6.11. THEOREM (Completeness for enforced folding up). *For any finite unsatisfiable set S_0 of ground clauses, any depth-first subgoal selection function, and any clause c_0 which is relevant in S_0, there exists a refutation of S_0 with top clause c_0 in the strongly regular connection tableau calculus with enforced folding up.*

PROOF. Let S_0 be a finite unsatisfiable set of ground clauses, c_0 any relevant clause in S_0, and ϕ any depth-first subgoal selection function. We demonstrate that *any* deterministic execution of Procedure 6.10 including enforced folding up operations leads to a refutation in which only strongly regular connection tableaux are constructed. We start with a tableau consisting simply of c_0 as the top clause, and let α map the root to any subset S of S_0 in which c_0 is essential. Then we prove by induction on the number of inference steps needed for deriving a tableau that

(i) any generated tableau T is strongly regular, and

(ii) an inference step can be performed at the subgoal $\phi(T)$ according to Procedure 6.10.

The induction base, $n=0$, is evident. For the induction step, let T be a tableau generated with $n > 0$ inference steps, $N = \phi(T)$ with literal L and path set P, c the tableau clause at N, N' the immediate predecessor of N, and $\alpha(N') = S$. Two cases have to be distinguished.

Case 1. Either, the last node selected before N was N'. In this case, by the induction assumption and the fact that Procedure 6.10 only permits extension (or start) steps with clauses not containing literals from the path set P, it is guaranteed that T is strongly regular, hence (i). By the induction assumption, c is essential in $P \rhd S$. Consequently, due to the Strong Mate Lemma, an inference step according to the procedure can be performed at N, therefore (ii).

Case 2. Or, the last inference step before the selection of N completely solved a brother node of N. In this case, after having entered the clause c, additional literals may have been inserted by intermediate folding up operations. We show that the resulting tableau is still strongly regular. For this purpose let N_i be an arbitrary subgoal in T, L_i the literal and c_i the clause at N_i, P_i its (extended) path set in T, and N_i' the immediate predecessor of N_i. With S_i we denote the clause set $\alpha(N_i')$. Furthermore, let $T^\star$ be the former tableau resulting from the extension step at N' into the clause c, and $P_i^\star$ the path set of N_i in $T^\star$. By the induction assumption, L_i is not contained in $P_i^\star$. According to Procedure 6.10, in the solutions of brother nodes of N_i only clauses from the set $S_i \setminus \{c_i\}$ are permitted for extension steps. Due to the depth-first selection function, the solution process of brother nodes of N is a subprocess of the solution process of brother nodes of N_i. Therefore, by the soundness of the folding up rule (Proposition 5.9), the set of literals K_i inserted into $P_i^\star$ during the derivation of T from $T^\star$ is logically implied by the satisfiable set $A_i = (P_i^\star \cup S_i) \setminus \{c_i\}$. Since, by the induction assumption, $A_i \cup \{c_i\}$ is unsatisfiable, $A_i \cup \{L_i\}$ is also unsatisfiable. Consequently, $L_i \notin K_i$, and hence $L_i \notin P_i$. Since this holds for all subgoals of T, T must be strongly regular, which proves (i). Furthermore, all c_i remain essential in the sets $P_i \cup S_i$. Therefore, by the Strong Mate Lemma, at the subgoal N in T an inference step according to Procedure 6.10 can be performed, hence (ii).

Now we have proven that the procedure produces only strongly regular connection tableaux and whenever the procedure terminates, it must terminate with a closed tableau. Finally, the termination of the procedure follows from the fact that, for any finite set of ground clauses, only strongly regular tableaux of finite depth exist. □

6.12. THEOREM (Completeness for enforced folding down). *For any finite unsatisfiable set S_0 of ground clauses, any subgoal selection function, and any clause c_0 which is relevant in S_0, there exists a refutation of S_0 with top clause c_0 in the strongly regular connection tableau calculus with enforced folding down.*

PROOF. The structure of the proof is the same as the one for folding up, viz., by induction on the number of inference steps it has to be shown that properties (i) and (ii) from the proof of Theorem 6.11 apply. Therefore, only the induction step is carried out. Suppose a subgoal N is selected with literal L, tableau clause c, path set P, and $\alpha(N') = S$ for the immediate predecessor N' of N. The enforced folding down operation inserts the negations of the literals at the unsolved brother nodes of N into the edge leading to N *before* the subgoal N is solved. First, we prove that such steps always preserve the strong regularity condition. Clearly, folding down

operations can only violate this condition for tautological tableau clauses. Since no tautological clause can be relevant in a set and Procedure 6.10 only permits the use of relevant clauses, no tautological clause can occur in a generated tableau, hence (i). It remains to be shown that any selected subgoal can be extended in accordance with Procedure 6.10. By the induction assumption, c is essential in $P \rhd S$. Hence, there is an interpretation $\mathcal{I}$ with $\mathcal{I}(c) = \text{false}$ and $\mathcal{I}((P \rhd S) \setminus \{c\}) = \text{true}$. We prove that any folding down operation preserves the essentiality of the clause c. Let $P' = \{\sim K_1, \ldots, \sim K_n\}$ be the set of literals inserted above N in a folding down operation on the literals $K_1, \ldots, K_n$ at the other subgoals in c. Clearly, $\mathcal{I}(\sim K_i) = \text{true}$, for all literals in P'. Therefore, c is essential in $P' \cup (P \rhd S)$ and hence also in its unsatisfiable subset $(P' \cup P) \rhd S$. Therefore (ii) also applies. □

7. Architectures of Model Elimination Implementations

All competitive implementations of model elimination are iterative-deepening search procedures using backtracking. When envisaging the implementation of such a procedure, one has the choice between fundamentally different architectures, for reasons we will now explain. As indicated at the end of Section 2.7.1, it is straightforward to recognize that SLD-resolution (the inference system underlying Prolog) can be considered as a refinement of model elimination obtained by simply omitting the reduction inference rule. Since highly efficient implementation techniques for Prolog have been developed, one can profit from these efforts and design a *Prolog Technology Theorem Prover (PTTP)*. The crucial characteristic of Prolog technology is that input clauses are *compiled* into procedures of a virtual or actual machine which permits a very efficient execution of the extension operation. There are even two different approaches taking advantage of Prolog technology. On the one hand, one can build on some of the efficient implementation techniques of Prolog and add the ingredients needed for a sound and complete model elimination proof procedure. On the other hand, one can use Prolog itself as implementation language with the hope that its proximity to model elimination permits a short and efficient implementation of a model elimination proof search procedure. The PTTP approaches, both of which will be described in this section, have dominated the implementations of model elimination in the last years. The use of Prolog technology, however, has a severe disadvantage, namely, that the framework is not flexible enough for an easy integration of new techniques and new inference rules. This inflexibility has almost blocked the implementations of certain important extensions of model elimination, in particular, the integration of inference mechanisms for an efficient equality handling. Therefore, we also present a more natural and modular implementation architecture for model elimination, which is better suited for various extensions of the calculus. Although this approach cannot compete with the PTTP approaches concerning the efficiency by which new instances of input clauses are generated, this drawback can be compensated for by an intelligent mechanism of reusing clause copies, so that about the same high rates of inferences per seconds can be achieved for the typical problems occurring in automated deduction.

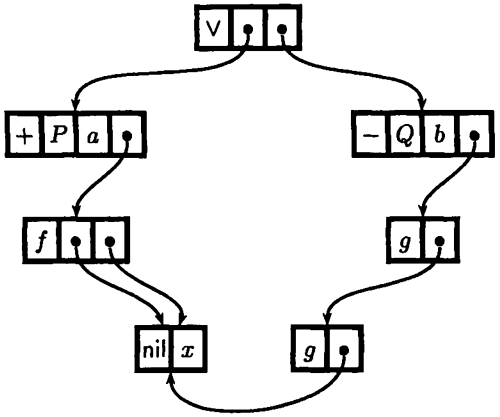


Figure 22: Internal representation of the clause $P(a, f(x, x)) \vee \neg Q(b, g(g(x)))$.

7.1. Basic Data Structures and Operations

When analyzing the actual implementations of model elimination, one can identify some data structures and operations that are more or less common to all successful approaches and hence form some kind of a standard framework. The first subject is the way formulae are represented internally in order to permit efficient operations on them. It has turned out that all terms, literals, and clauses may be represented in a natural tree manner except variables, which should be shared. In Figure 22, such a standard representation of a clause is illustrated. The treatment of variables needs some further explanation. Internally, variables are typically represented as structures consisting of their actual bindings and their print names with nil indicating that the variable is currently free. Variables are not identified and distinguished by their print names but by the addresses of their structures.

7.1.1. Unification

The next basic ingredient is the unification algorithm employed, which is specified generically in Table 2. In the displayed procedures, it is left open how variable bindings are performed and retracted. Unification is specified with two mutually recursive procedures, the first one for the unification of two lists of terms, the other one for the unification of two terms. The standard in model elimination implementations is that a binding is performed destructively by deleting nil from the variable cell and inserting a pointer to the term to be substituted for the variable. The resulting bound variable cell then does no more denote a variable but the respective term. The recursive function binding(term) returns term if term is not a bound variable cell, or otherwise binding(first(term)). On backtracking, variable bindings have to be retracted. This is done by simply reinserting nil in the first

```
procedure unify_lists( args1,args2 )
    if ( args1 = ∅ ) then
      true;
    /* check first arguments */
    elseif ( unify( binding( first( args1 ) ),binding( first( args2 ) ) ) ) then
      unify_lists( rest( args1 ),rest( args2 ) );
    /* undo variable bindings made in this procedure */
    else
      unbind;
      false;
    endif;

procedure unify( arg1,arg2 )
    if ( is_var( arg2 ) ) then
      if ( occurs( arg2,arg1 ) ) then
        false;
      else
        bind( arg2,arg1 );
        true;
      endif;
    elseif ( is_var( arg1 ) ) then
      if ( occurs( arg1,arg2 ) ) then
        false;
      else
        bind( arg1,arg2 );
        true;
      endif;
    elseif ( functor( arg1 ) == functor( arg2 ) ) then
      unify_lists( args( arg1 ),args( arg2 ) );
    else
      false;
    endif;
```

Table 2: The unification procedures.

element of the respective bound variable cells, so that the original state is restored. unbind causes the bindings of the whole unification attempt to be retracted.

7.1.2. Polynomial unification

It is straightforward to recognize that this unification procedure is linear in space but exponential in time in the worst case. Although this is not a critical weakness for the typical formulae in automated deduction, one may easily improve the given procedure to a polynomial time complexity by using methods described in [Corbin and Bidoit 1983]. The key idea of such methods is that one attaches an additional tag to any complex term. This tag is employed to avoid that the same pairs of complex terms are successfully unified more than once during a unification operation. Furthermore, this tag can be used to reduce the number of occurs checks to a polynomial (see also [Letz 1993, Letz 1999]).

7.1.3. Destructive unification using the trail

In order to know which bound variables have to be unbound, the *trail* is used as a typical data structure. The trail is a global list-like structure in the program which contains the pointers to the bound variables in the order in which they have been bound. Since all standard backtracking procedures retract bindings exactly in the reversed order of their generation, a simple one-dimensional list-like structure is sufficient for the trail. The *trailmarker* is a global variable which gives the current position of the trail. The number of bindings performed may differ from one inference step to another. In order to know how many bindings have to be retracted when an inference step is retracted, there are two techniques. One possibility is to locally store either the number of performed bindings or the trail position at which the bindings of the previous inference step start. Alternatively, one can use a special stop label on the trail which is written in a trail cell whenever an inference step ends; in this case, no local information is needed. Depending on which solution is selected, the unification procedure has to be modified accordingly.

Figure 23 documents the entire binding process and the trail modifications performed during proof search for the set of the four clauses $\neg P(x,y) \vee \neg P(y,x)$, $P(a,z)$, $P(b,v)$, and $Q(a,b)$. The description begins after the start step in which the first input clause has been attached (a). First an extension step using the second input clause is performed, which produces two bindings (b). Then an extension step with the Q-subgoal is attempted: y (and implicitly z) are bound to a, but the unification fails when a (the binding of x) is compared with b (c). Next the two inferences are retracted (d). After extension steps using the third clause (e) and the fourth clause (f), the proof attempt succeeds. This technique permits that backtracking can be done very efficiently.

7.1.4. The connection graph

The connection condition from Definition 2.3 is a prerequisite for any model elimination extension step. Therefore it is advantageous to be able to quickly find the complementary literals for a selected subgoal. The set of connections between the

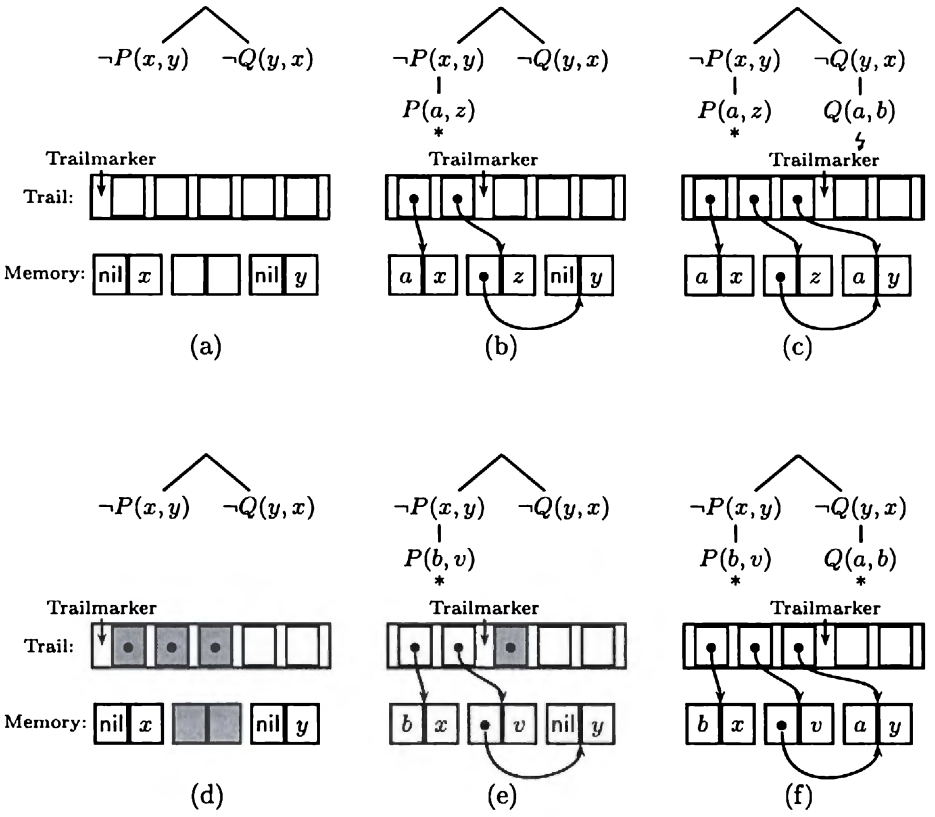


Figure 23: An example of the trail modifications during proof search.

literals of a clause set can be represented in an undirected graph, the so-called *connection graph*. When a subgoal is selected for an expansion step during the proof search, it is sufficient to consider the connections involving that subgoal.

7.1. EXAMPLE. To demonstrate the concept of the connection graph, we consider the clauses $\neg P(g(x)) \vee \neg Q(g(x)) \vee \neg Q(f(x))$, $Q(x) \vee \neg P(f(x))$, $Q(g(y)) \vee P(f(y))$ and $P(f(a))$. The connection graph of this set of clauses is shown in Figure 24. The connections are indicated by the solid lines between the literals. The faint dashed lines show the pairs of literals where, even though they have the same predicate symbol and complementary signs, the argument terms cannot be unified. These literals are not connected. Thus, when the literal $\neg P(g(x))$ has been chosen for an extension step, the clause $Q(g(y)) \vee P(f(y))$ need not be tried.

Since the variables in clauses are all implicitly universally quantified, the connections between literals are independent of any instantiations applied during the proof

search. Therefore, the computation of the literal connections can be done statically and used as a filter. If, generally speaking, there is a connection (P, Q) in a set of clauses, then the literal P is also said to have a *link* to literal Q and vice versa. Thus, each literal has a list of links. The link lists for the literals in Example 7.1 are:

$$
\begin{aligned}
P(g(a)): &\quad \{\neg P(g(x))\} \\
\neg P(g(x)): &\quad \{P(g(a))\} \\
\neg Q(g(x)): &\quad \{Q(g(y)), Q(x)\} \\
\neg Q(f(x)): &\quad \{Q(x)\} \\
Q(x): &\quad \{\neg Q(g(x)), \neg Q(f(x))\} \\
\neg P(f(x)): &\quad \{P(f(y))\} \\
P(f(y)): &\quad \{\neg P(f(x))\} \\
Q(g(y)): &\quad \{\neg Q(g(x))\}
\end{aligned}
$$

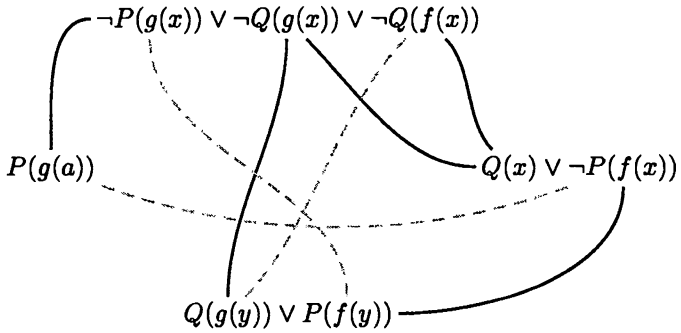


Figure 24: The connection graph for the clause set $\neg P(g(x)) \vee \neg Q(g(x)) \vee \neg Q(f(x))$, $Q(x) \vee \neg P(f(x))$, $Q(g(y)) \vee P(f(y))$ and $P(g(a))$.

7.1.5. The problem of generating clause variants

One of the main difficulties when implementing model elimination procedures is how to provide renamed variants of input clauses efficiently, because in every extension step a new variant of an input clause is needed. It is obvious that the generation of a new variant of an input clause by copying the clause and replacing its variables consistently with new ones is a time consuming operation, all the more since variables are shared and it is not a tree that has to be copied but a graph. The search for an efficient solution of this problem naturally leads to the use of Prolog technology.

7.2. Prolog Technology Theorem Proving

One reason for the high efficiency of current Prolog systems is the fact that many of the operations to be performed in SLD-resolution steps can be determined in advance depending on the respective clause and its entry literal. This information can be used for compiling every Prolog input clause A :- $A_1,\ldots,A_n$ (which corresponds to the clause $A \vee {\sim}A_1 \vee \cdots \vee {\sim}A_n$ with entry literal A) into procedures of some actual or virtual machine. Since SLD-resolution steps are nothing else but extension steps, this technique can also be applied to model elimination. The first one to use such a compilation method for model elimination was Mark Stickel [Stickel 1984] who called his system a PTTP, a Prolog Technology Theorem Prover.

In summary, the main deficiencies of Prolog as far as first-order automated reasoning is concerned are the following:

1. the incompleteness of SLD-resolution for non-Horn formulae,
2. the unsound unification algorithm, and
3. the unbounded depth-first search strategy.

To extend the reasoning capabilities of Prolog to full model elimination, it is necessary to extend SLD-resolution to the full extension rule and to add the start rule and the reduction rule.

7.2.1. Contrapositives

In order to implement the full extension rule and to further permit the compilation of input clauses into efficient machine procedures, one has to account for the fact that a clause may be entered at every literal. Accordingly, one has to consider all so-called *contrapositives* of a clause $L_1 \vee \cdots \vee L_n$, i.e., the n Prolog-style strings of the form L_i :- ${\sim}L_1,\ldots,{\sim}L_{i-1},{\sim}L_{i+1},\ldots,{\sim}L_n$. The start rule can also be captured efficiently, by adding a contrapositive of the form $\bot$:- ${\sim}L_1,\ldots,{\sim}L_n$ for every input clause $L_1 \vee \cdots \vee L_n$. Now, with the Prolog query ?- $\bot$ as the single start clause, all start steps can be simulated with extension steps. As a matter of fact, one can use relevance information here and construct such start contrapositives only for subsets of the input clause set containing a relevant start clause; by default, start contrapositives are generated for the set of all-negative input clauses.

7.2.2. Unification in Prolog

Prolog by default uses a unification algorithm that is designed for maximum efficiency but that can lead to incorrect results.[11] A Prolog program like

```
X < ( X + 1 ).
:- ( Y + 1 ) < Y.
```

can prove that there is a number whose successor is less than itself; the reason for the unsoundness is that no occurs check is performed in Prolog unification. Since the compilation of extension steps into machine procedures also concerns parts of the unification, this compilation process has to be adapted such that sound unification

[11] Some Prolog systems provide sound unification via compile time and/or runtime options or via libraries.

Contrapositive: $P(a, f(x,x))$:- $Q(b, g(g(x)))$

```
procedure P( arg1,arg2 )
    variable x,tq,arg21,arg22,trail_position;
    x := new_free_variable;
    arg1 = binding( arg1 );
    /* mark trail position */
    trail_position := trailmarker;
    /* unify clause head: check first arguments */
    if ( is_var( arg1 ) or ( is_const( arg1 ) and arg1 == a ) ) then
      if ( is_var( arg1 ) ) then bind( arg1,a ); endif;
      /* first arguments unifiable, check second arguments */
      arg2 = binding( arg2 );
      if ( is_var( arg2 ) ) then
        bind( arg2,make_complex_term( f,x,x ) );
        tq := make_complex_term( g, make_complex_term( g,x ) );
        add_subgoal( Q(b,tq) );
        next_subgoal;
      elseif ( is_complex_term( arg2 ) and functor( arg2 ) == f ) then
        arg21 := binding( get_arg( arg2,1 ) );
        arg22 := binding( get_arg( arg2,2 ) );
        if ( unify( arg21,arg22 ) ) then
          tq := make_complex_term( g, make_complex_term( g,arg21 ) );
          add_subgoal( Q(b,tq) );
          next_subgoal;
        endif;
      endif;
    endif;
    /* undo variable bindings made in this procedure */
    unbind( trail_position );
```

Table 3: Compilation of a contrapositive into a procedure.

is performed. In special cases, however, efficiency can be preserved, for example, if the respective entry literal L is *linear*, i.e., if every variable occurs only once in L. It is straightforward to recognize that in this case, no occurs check is needed in extension steps and the highly efficient Prolog unification can be used. For the general case, an ideal method takes advantage of this optimization by distinguishing the first occurrence of a variable in a literal from all subsequent ones. For every first occurrence, the occurs check may be omitted. In Table 3, a procedure is shown which performs an extension step including the generation of a new clause variant in a very efficient manner.

7.2.3. Path information and other extensions

Unfortunately, for the implementation of the reduction inference rule, one definitely has to provide additional data structures. While in SLD-resolution the ancestor literals of a subgoal are not needed, for model elimination the tableau paths have to be stored and every subgoal must have access to its path. This additional effort cannot be avoided. On the other hand, access to the ancestors of a subgoal is necessary for the implementation of basic refinements like regularity, which is also very effective in the pure Horn case.

Finally, the unbounded depth-first search strategy of Prolog has to be extended to incorporate completeness bounds like the inference bound, the depth bound or other bounds discussed in Section 2.5. It turned out in general that this can be achieved with the following additional data structures. In order to capture bounds which allocate remaining resources directly to subgoals such as the depth bound, every subgoal has to be additionally labelled with its current depth. When inference bounds are involved, a global counter is needed.

7.3. Extended Warren Abstract Machine Technology

As described in [Stickel 1984], PTTP implementations perform their tasks in two distinct phases. First the input formula is translated to Prolog code which in the second phase is compiled into (real or virtual) machine code. The main problem of such a two-step compilation, first to some real programming language and then to native code, is that the second compilation process takes too much time for typical applications, which require short response times. In order to avoid the second compilation phase, an interpreter for the code generated in the first compilation phase has to be used. Since the full expressive power of an actual programming language is not needed, this has caused the development of a very restricted abstract language tailored specifically to the processing of Prolog or model elimination, respectively. We begin with a description of the basics of such a machine for Prolog (see [Warren 1983] and [Schumann 1991] for a more detailed description).

7.3.1. The Warren Abstract Machine

D.H.D. Warren developed a virtual machine for the execution of Prolog programs [Warren 1983] which is called the Warren Abstract Machine (WAM). It combines high efficiency, good portability, and the possibility for compiling Prolog programs. The WAM is widely used and has become a kernel for commercial Prolog systems, implemented as software emulation or even micro-coded on dedicated hardware [Taki, Yokota, Yamamoto, Nishikawa, Uchida, Nakashima and Mitsuishi 1984, Benker, Beacco, Bescos, Dorochevsky, Jeffré, Pöhlmann, Noyé, Poterie, Sexton, Syre, Thibault and Watzlawik 1989]. The WAM is structured as a register-based multi-memory machine as shown in Figure 25. Its *memory* holds the program (as a sequence of WAM instructions) and data. The *register file* keeps a certain set of often used data and control information. The WAM instruction, located in the memory at the place where the *program counter* register points to,

is fetched and executed by the *control unit.*

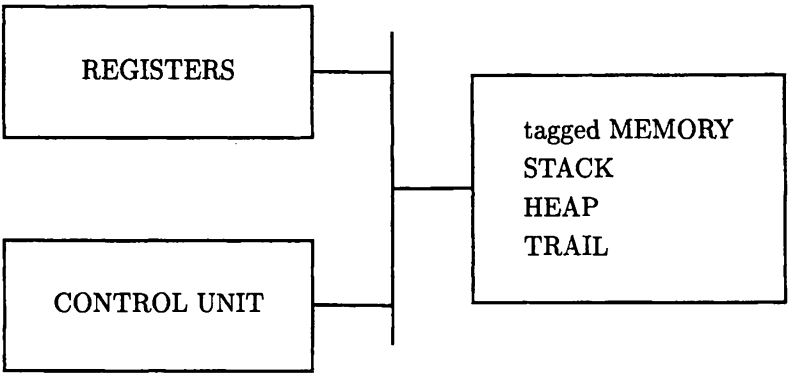


Figure 25: The Warren Abstract Machine

Next we will describe how a Prolog program is *compiled* into machine instructions of the WAM. We begin with the special case of a *deterministic* program which corresponds to a situation in which there is only one possibility for extending the subgoals of the clause. In this case no backtracking inside the clause is needed. The respective tableau is generated using a depth-first left-to-right selection function. Then the program can be executed in the same manner as in a *procedural* programming language, that is, the head of a clause is considered as the *head* of a procedure and the subgoals as the *procedure calls* of other procedures (the parameter passing, however, is quite different). Accordingly, this can be implemented on a machine level exactly in the way it is done in functional or procedural languages, using a *stack* with *environment control blocks* which hold the control information (return address, dynamic link) and the local variables. A detailed explanation of these concepts can be found e.g. in [Aho and Ullman 1977]. The local variables are addressed using a register E pointing to the beginning of the current environment. A Horn clause `H :- G1,...,Gn.` is executed using the following instructions[12].

```
H:                          % entry point for clause H.
        allocate            % generate new environment (on stack) with space for locals
        ...                 % pass parameters (discussed below)
        ...                 % set parameters for G1 (discussed below)
        call        G1      % call first subgoal, remember return address A
A: ...
        ...                 % set parameters for Gn (discussed below)
        call        Gn      % call last subgoal, remember return address
        deallocate          % deallocate control block and return
```

Each environment contains a pointer to the previous environment (*dynamic link*). The entire list, in effect, represents the *path* from the root to the current node in the

[12]Actually, the WAM provides a number of different instructions for the sake of optimization, e.g., for tail recursion elimination. Here, only the basic instructions are described.

tableau, the return addresses in the environments point to the code of the subgoals. The program terminates when the last call in the query returns.

The parameters of the head and the subgoals of the clauses are *terms* in a logical sense consisting of *constants*, *logical variables*, *lists*[13], and *structures* (complex terms). A Prolog term is represented by a *word* of the memory, containing a *value* and a *tag*. The tag distinguishes the type of the term, namely *reference*, *structure*, *list*, and *constant*. The tag type "reference" is used to represent the (logical) variables. Structures are represented in a non structure-sharing manner, i.e. they are copied explicitly with their functors.

For the purposes of parameter passing, the WAM uses two sets of registers, the registers $A_1, \ldots, A_n$ for keeping parameters and temporary registers $T_1, \ldots, T_n$. When a subgoal is to be called, its parameters are provided in the registers A_i by using `put` instructions. There exists one `put` instruction for each data type. In the head of a clause, the parameters in the A registers are fetched and compared with the respective parameter of the head, using a `get` or `unify` instruction. Here again, a separate instruction for each data type is provided. The matching algorithm has to check if constants and functors are equal. If a variable has to be bound to a constant, the value of the constant and the tag "constant" is written into the memory location where the variable resides; if the variable is bound to a structure, a pointer to that structure is written into the variable cell, together with the tag "reference". Structures themselves are created in a separate part of memory, the *heap*, to ensure their permanent storage.

An example illustrates the usage of the instructions to pass the parameters. Let us assume that a subgoal $P(a, z)$ calls a head of a clause $P(a, f(x, y))$:- .... The variables reside in the environment control block and are accessed via an offset from the register E pointing to the current environment. In [Warren 1983] they are noted as $Y_1, \ldots, Y_n$.

```
    put_constant      a,A1     % put first parameter (constant a) into register A1
    put_variable      Y4,A2    % put variable z (in variable cell #4) into A2
    call              P        % call the "P-clause"

P:
    allocate          2        % allocate space for 2 variables
    get_const         a,A1     % try to unify 1st parameter with constant a
    get_structure     f/2,A2   % get second argument: must be a structure or
                               % a variable to be bound to a new structure
    unify_variable    Y1       % unify with first arg (in local cell #1)
    unify_variable    Y2       % unify with second arg (in local cell #2)
    ...                        % body of clause comes here
```

This example also shows that the `get` and `unify` instructions must operate in two modes ("read","write") according to the type of parameter they receive. If the variable z in the subgoal has been bound to some function symbol prior to this call, for example, to $f(a, b)$, then the list is taken apart by the `get_list` instruction and x and y in the head of P are bound to a respectively to b (read mode). If, however, z

[13] A list is considered as a data type of its own for reasons of efficiency. A list could also be represented as a binary structure: *list*(*Head*, *Tail*) comparable to a Lisp *cons*-structure.

in the subgoal has not yet been bound to a function symbol, a *new* binary function symbol with two variables as arguments is created as a structure on the heap by the instructions `get_structure` and `unify_variable` (write mode). Note that the creation on the heap is necessary, since the newly created structure has to stay in existence even after the execution of the clause P.

Finally, let us consider the full case of nondeterministic programs, in which a subgoal of a clause unifies with more than one (complemented) head of a clause, in which case backtracking is needed. Backtracking is implemented by means of so-called *choice points*, control blocks which hold all the information for undoing an inference step. These choice points are pushed onto the stack. The basic information of a choice point is a link to its predecessor, a code address to the entry point of the next clause to be attempted, and the information that is needed to undo all extension steps executed since that choice point was created. This involves a copy of all registers of the WAM as well as the variables which have been bound since the generation of the choice point. For the latter purpose a *trail* is used, in the same manner as described in Section 7.1. Whenever a backtracking action has to be performed, all registers from the current choice point are loaded into the WAM, all stack modifications are undone, and the respective variables are unbound. Then the next clause is attempted. The WAM contains a last alternative optimization, according to which the choice point can be discarded if the last extension clause is tried. The list of different possibilities is coded by the instructions `try_me_else`, and `trust_me_else_fail`, the latter representing the last alternative. Assuming that there are three clauses c1, c2, c3 for extension, the compiled code is shown below.

```
    ...
    call    P               % call the P-clauses
P:                          % generate a choice-point.
c123:
    try_me_else C2a         % try c1; if this fails, try c2
c1:
                            % code of clause c1
c2a:
    try_me_else C3a         % try c2; if this fails, try c3
c2:
                            % code of clause c2
c3a:
    trust_me_else_fail      % there is only one alternative left
c3:
                            % code of clause c3
```

The WAM has some additional instructions for optimization which we do not consider here. First, a *dynamic* preselection on the data type of the first parameter is done (`switch_on_term`). Its arguments give entry points of lists of clauses which have to be tried according to the type of the first parameter of the current subgoal (variable, constant, list, structure). Also hash tables are used for selection of a clause head, which is useful when there is a large number of head literals with constants as first arguments.

7.3.2. The SETHEO abstract machine

Inspired by the architecture of the WAM, the SETHEO model elimination prover [Letz et al. 1992] has been implemented. The central part of SETHEO is the SETHEO Abstract Machine (SAM), which is an extension of the WAM. The concepts introduced there had to be extended and enhanced for attaining a complete and sound proof procedure for the model elimination calculus, and for facilitating the use of advanced control structures and heuristics. A detailed description of all the instructions and registers of the original version is available as a manual [Letz, Schumann and Bayerl 1989]. The layout of the abstract machine is basically the same as in Figure 25, except that additional space is reserved for the *proof tree* and the *constraints*, which are discussed in Section 8. The *proof tree* stores the current state of the generated tableau, which can be displayed graphically to illustrate the structure of the proof. Additionally, there are global counters, e.g. for the number of inferences performed.

In the following we will point out and explain some of the most important differences between the SAM and the WAM.

The Reduction Step. In order to successfully handle non-Horn clauses in model elimination, extension steps and *reduction steps* are necessary. A subgoal in the tableau can be closed by a reduction step if there exists a complementary unifiable literal in the path from the root to the current node. The resulting substitution σ is then applied to the entire tableau. How can this reduction step be implemented within the concepts of an Abstract Machine? As described above, the tableau is implicitly represented in the stack of the machine, using a linked list of *environment control blocks*. This linked list just represents the path from the root of the tableau to the current node[14]. Thus, the instruction executing the reduction step searches through this list, starting from the current node, to find a complementary literal which is unifiable with the current subgoal. The respective unification is carried out in the standard way. This procedure, however, requires that additional information must be stored in each environment, namely, the *predicate symbol* of the head literal of a contrapositive, its *sign*, and a pointer to the *parameters* of that literal. The detailed structure of an environment of the SAM is displayed in Figure 26, the *base pointer* points to the current environment in the *stack*.

The reduction inference rule itself is nondeterministic in the sense that a subgoal may have more than one connected predecessor literal in the path. Hence, we have to store an additional *pointer* in every choice point, pointing to the environment which corresponds to the node which will be tried next for a reduction step.

Efficiency Considerations. To increase the efficiency of the SETHEO machine, a tagged memory is used. The basic types of variables, terms, constants and reference cells, which are used in the Warren Abstract Machine, are divided into further subtypes in order to gain a better performance (compare also [Vlahavas and Halatsis 1987]). Thus, for instance, the type 'variable' has the subtypes: 'free

[14]For this reason no *tail recursion* optimization is allowed as it is done in the WAM. This optimization tries to throw away environments as soon as possible, e.g., before executing the last subgoal of a clause.

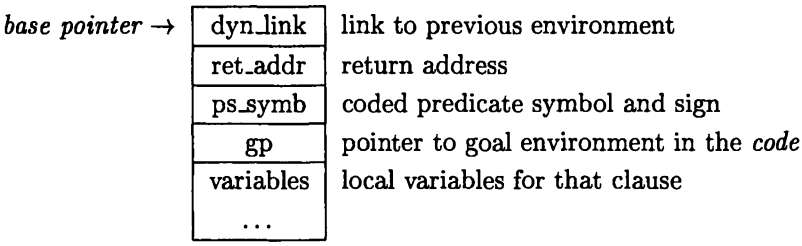


Figure 26: The SAM environment

variable' (T_FVAR), 'temporary variable' (T_TVAR), and 'bound variable', i.e. a reference cell (T_BVAR). Additionally, complex terms are tagged differently depending on whether they contain variables or not. The additional information contained in these tags can be used for optimizing the unification operation.

Parameter Transfer. In the original WAM, parameter transfer from a subgoal to a head of a clause is done via the A_i registers. As the number of registers had to be minimized and the ability to deal with a variable number of parameters was required, this solution was not suitable for the SAM. Instead, the parameters are transferred via an argument vector. This approach originates from [Vlahavas and Halatsis 1987], but had to be adapted. The number of parameters of a subgoal and their types are fixed. The only exception are variables, which may be unbound or bound to an arbitrary object. Consequently, an argument vector is generated in the code area during compilation which contains the values and data types of the parameters. For variables an offset into the current environment is given. After dereferencing this address, the object to which the variable is bound can be accessed. The only information directly passed during the execution of a `call` instruction is the address of the beginning of this argument vector. It is put into the register *gp* (goal pointer). After the selection of a head of a clause, the unification between the parameters of the goal and those of the head starts. For each parameter in the head, a separate unify instruction is used which tries the unification with the parameter *gp* points to, and, in the case of success, increments *gp*. The following example shows the construction of the argument vector. Consider a subgoal $P(a, x, f(x))$. It will generate something like the following argument vector consisting of three words.

```
gp:     T_CONST   16    % 1st argument: constant a as index into symbol table
        T_VAR1    1     % 2nd: variable x with offset 1 (w.r.t.\ environment)
        T_CREF    term1 % 3rd: pointer to term f(x)
        ...
term1:  T_NGTERM  17    % functor f with index 17
        T_VAR2    1     % variable x (second occurrence)
        T_EOSTR   0     % end of the structure
```

7.4. Using Prolog as an Implementation Language

The preceding section has shown that implementing model elimination by extending Prolog technology requires considerable effort. Since SLD-resolution is very similar to model elimination, many newer implementations of model elimination are done directly in Prolog. We will now consider the potential of using Prolog as an implementation language. It is easy to see that a basic implementation of model elimination can be obtained in Prolog with a few straightforward additions. For this purpose, we need to explicitly create all contrapositives and a mechanism for performing reduction steps has to be provided. Both can be done in a simple and methodical way, as will be demonstrated with the following formula proposed by J. Pelletier in [Pelletier and Rudnicki 1986]. We have written the problem in Prolog-like notation, i.e., with variables in capital letters and function and predicate symbols in lower case letters. A semi-colon is used when more than one positive literal is in a clause.

```
< - p(a,b).
< - q(c,d).
p(X,Z) < - p(X,Y), p(Y,Z).
q(X,Z) < - q(X,Y), q(Y,Z).
p(X,Y) < - p(Y,X).
p(X,Y) ; q(X,Y) < -.
```

The transformation starts by forming the Horn contrapositives for the input clauses, as shown in Section 7.2.1. To simulate the negation sign, predicate symbols are preceded with labels, `p_` for positive literals and `n_` for negative literals. Additionally, start clauses are added as Prolog queries.

Furthermore, to overcome the incompleteness of Prolog for non-Horn formulae, we need to simulate the reduction operation. This is done as follows. First, the paths are added as additional arguments to the logical arguments of the respective literals. For optimization purposes, we use two path lists, one for the positive and one for the negative literals in the respective path. In each extension step, the respective path list is extended by the respective literal. Finally, for actually enabling the performance of reduction steps, an additional clause is added for each predicate symbol and sign that tries all unifiable literals in the path list. The output then looks as follows.

```
% Start clauses
false :- p_p(a,b, [ ],[ ]).

false :- p_q(c,d, [ ],[ ]).

% Contrapositives
n_p(a,b, P, N).

n_q(c,d, P, N).

p_p(X,Z, P,N) :- N1 = [ p(X,Z) | N ], p_p(X,Y, P,N1), p_p(Y,Z, P,N1).
n_p(X,Y, P,N) :- P1 = [ p(X,Y) | P ], n_p(X,Z, P1,N), p_p(Y,Z, P1,N).
```

```
n_p(Y,Z, P,N) :- P1 = [ p(Y,Z) | P ], n_p(X,Z, P1,N), p_p(X,Y, P1,N).

p_q(X,Z, P,N) :- N1 = [ q(X,Z) | N ], p_q(X,Y, P,N1), p_q(Y,Z, P,N1).
n_q(X,Y, P,N) :- P1 = [ q(X,Y) | P ], n_q(X,Z, P1,N), p_q(Y,Z, P1,N).
n_q(Y,Z, P,N) :- P1 = [ q(Y,Z) | P ], n_q(X,Z, P1,N), p_q(X,Y, P1,N).

p_p(X,Y, P,N) :- N1 = [ p(X,Y) | N ], p_p(Y,X, P,N1).
n_p(Y,X, P,N) :- P1 = [ p(Y,X) | P ], n_p(X,Y, P1,N).

p_p(X,Y, P,N) :- N1 = [ p(X,Y) | N ], n_q(X,Y, P,N1).
p_q(X,Y, P,N) :- N1 = [ q(X,Y) | N ], n_p(X,Y, P,N1).

% Clauses for performing reduction steps
n_p(X,Y, P,N) :- member(p(X,Y), N).
p_p(X,Y, P,N) :- member(p(X,Y), P).
n_q(X,Y, P,N) :- member(q(X,Y), N).
p_q(X,Y, P,N) :- member(q(X,Y), P).

member(X,[ X | R ]).
member(X,[ Y | R ]) :- member(X,R).
```

What is missing in order to perform a complete proof search, is the implementation of a completeness bound and the iterative deepening. We consider the case of the tableau depth bound (Section 2.5.2), which can be implemented by adding the remaining depth resource `D` as an additional argument to the literals in the contrapositives and start clauses. After having entered a contrapositive, it is checked whether the current depth resource is > 0, in which case it is decremented by 1 and the new resource is passed to the subgoals of the clause. For start clauses, the depth may be passed unchanged to the subgoals.

```
% for contrapositives
P(...,D) :- D > 0, D1 is D-1, P1(...,D1), ..., Pn(...,D1).

% for start clauses
false(D):- P1(...,D), ..., Pn(...,D).
```

When posing the query, say, `false(5)`, the Prolog backtracking mechanism will automatically ensure that all connection tableaux up to tableau depth 5 are examined. Finally, the iterative deepening is handled by simply adding the following clause to the end of the program.

```
false(D) :- D1 is D+1, false(D1).
```

After having loaded such a program into Prolog (in some Prolog systems the clauses have to be ordered such that all predicates occur consecutively), one can start the proof search by typing in the query: `?- false(1).`

For the discussed example, the Prolog unification (which in general is unsound) poses no problem, since no function symbol of arity > 0 occurs. In the general case, however, one has to use sound unification. Some Prolog systems offer sound unification, often in various ways. Either the system has a sound unification predicate in its library or sound unification can be switched on by setting a global flag. While

the latter is more comfortable, it may lead to unnecessary run-time inefficiencies, since the occurs check is always performed even if it would not be needed according to the optimizations discussed in the previous sections. Such an optimization may also be achieved in a Prolog implementation by linearization of the clause heads (for which Prolog unification may be used then) and a subsequent sound unification of the remaining critical terms (see, for example, [Plaisted 1984]).

In summary, this illustrates how surprisingly simple it is to implement a pure model elimination proof search procedure in Prolog. Furthermore, such an implementation also yields a very high performance in terms of inference steps performed per second. The approach of using Prolog, however, becomes more and more problematic when trying to implement model elimination proof procedures with more advanced search pruning mechanisms such as the ones discussed in Section 3 and Section 4.

7.5. A Data Oriented Architecture

The prover architectures described so far all rely on the approach of compiling the input clauses and some parts of the inference system into procedures, the ones creating Prolog source code and the ones generating native or abstract machine instructions. The inference rules and important subtasks such as the unification algorithm, the backtracking mechanism, or the subgoal processing are deeply intertwined and standardized in order to achieve high efficiency. Such an approach is suitable when a certain kind of optimized proof procedure has evolved for which no obvious improvements are known. In automated theorem proving, however, this is not the case. New techniques are constantly developed which may lead to significant improvements. Against this background, the most important shortcoming of Prolog technology based provers is their inflexibility. Changing the unification such as to add sorts, for example, or adding new inference rules, e.g., for equality handling, or generalizing the backtracking procedure becomes extremely cumbersome if not impossible in such an architecture.

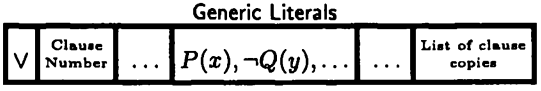
Accordingly, as the last of the architectures, we discuss a more natural or straightforward implementation of the model elimination procedure, in the sense that the components of the program are modularized and can be identified more naturally with their mathematical definitions. Since the most important difference to Prolog technology style provers is that clauses are represented as data structures and do not become part of the prover program, such an approach will be called a data oriented proof procedure, as opposed to the clause compilation procedures. Unlike the WAM-based architectures, which heavily rely on the implicit encoding of the proof in the program execution scheme, the proof object here is the clausal tableau, which is completely stored in memory. Although this leads to a larger memory consumption, it causes no problems in practice, as today's computers have enough main storage space to contain the proof trees for practically all *feasible* proof problems. Only very large proofs, that means proofs with more than, say, 100,000 inferences become unfeasible with the data oriented concept.

An implementation of this data oriented approach, the SETHEO-based prover system Scheme-SETHEO, has been partially completed.

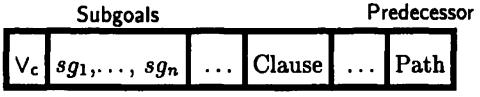
7.5.1. The basic data structures

The data objects used in this approach can be categorized into *formula data objects* and *proof data objects*. The basic formula data objects are the (input) formula, the clauses, the clause copies and the subgoals. From these objects, the formula is implicitly represented by the set of its clauses (as there is only one input formula). Reasonable data structures for the other objects are given here.

Clauses. The most important elements of the clause structure are the original or *generic* literals and the list of clause copies used in the proof. Since clauses may be entered at any subgoal, it is not necessary to compute contrapositives.



Clause copies. In every extension step, a renamed copy of the original clause has to be made and added to the tableau.



Subgoals. Subgoal objects contain the information about the literal they represent, i.e. the sign, predicate symbol, the argument terms, etc.

As a matter of fact, additional control information can be included in these data objects, which is omitted here for the sake of clarity. Further important data structures utilized in the proof process are the variable trail (which was described in Section 7.1.3) and the list of subgoals. The variable trail is one of the few concepts adopted from the Prolog technology architecture, since a device for manual bookkeeping of the variable instantiations is required.

Global subgoal list. This is the central global data object. It consists of the sequence of subgoals of all clause copies hitherto introduced to the current tableau. Figure 27 illustrates how the subgoals of the clause copies constitute the global subgoal list. The dotted lines refer to the underlying tree structure, the dashed arrows indicate the linking between the elements of the global subgoal list. In any inference step, the literal at which an extension or reduction step is performed, is marked as selected, as illustrated in Figure 27 by a grey shading of the subgoals.

The global access to the list of subgoals relieves us from the need to conform to some sort of depth-first search. Instead, a subgoal selection function can be em-

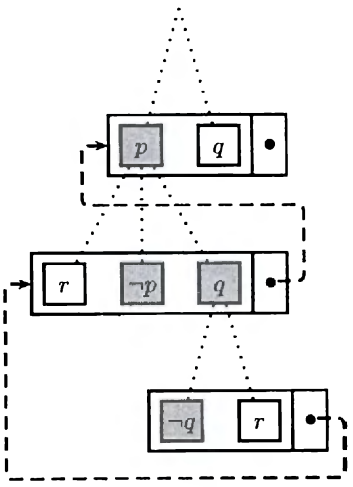


Figure 27: The global subgoal list and the implied tableau structure.

ployed that chooses an arbitrary subgoal for the next inference step. This way, new heuristics become feasible that operate globally on the proof object. For example, free subgoal reordering can be performed easily.

In Figure 28, a detailed snapshot of the subgoal and clause copy data structures of a certain proof state is shown. The tableau structure is only given implicitly. In fact, all information needed to move through the tableau, as, for instance, during a folding up operation, is provided by extensive cross-referencing among the different data objects. The figure shows the connections between data objects of the various kinds (data objects for original clauses are not contained in the tableau). Again, selected subgoals are highlighted by grey shading. The subgoal p has been chosen for an extension step with the clause $C = \{r, \neg p, q\}$. A copy C' of C is linked to p via the extension pointer. The subgoals of C' are accessible via the subgoal vector pointed to by C'. This subgoal vector is appended to the global subgoal list. To allow upward movement through the tableau, the copy is linked to the extended subgoal, while the new subgoals are linked to the clause copy. The subgoals p and $\neg p$ are immediately marked as selected, the subgoal q becomes selected in the next extension step. It should be noted that, since we rely on clauses instead of contrapositives, the connected literal need not be the first literal in the clause, as is the case here.

7.5.2. The proof procedure

Table 4 shows a simplified data oriented proof procedure (not featuring the reduction rule or start clause selection). Based on the connection graph of the input formula, the list of associated links is attached to each subgoal. The procedure

```
procedure solve( sg, links, resource )
    if ( links ≠ ∅ ) then
      extension( sg, first( links ), resource );
      /* try next alternative */
      solve( sg, rest( links ), resource );
    endif;

procedure extension( sg, link, resource )
    dec_resource := decrement_resource( resource );
    if ( dec_resource > 0 ) then
      clause := new_clause_copy( link );
      head := head( clause, link );
      trail_pos := trailmarker;
      if ( unify_literals( ~sg, head ) ) then
        old_subgoals := subgoals;
        make_new_subgoals( clause, sg, head );
        new_sg := select_subgoal;
        if ( new_sg )
          new_links := links( new_sg );
          new_resource := resource( new_sg );
          solve( new_sg, new_links, new_resource );
        else
          proof_found;
          abort;
        endif;
        /* backtracking */
        unbind( trailpos );
        subgoals := old_subgoals;
      endif;
    endif;
```

Table 4: A rudimentary model elimination proof procedure.

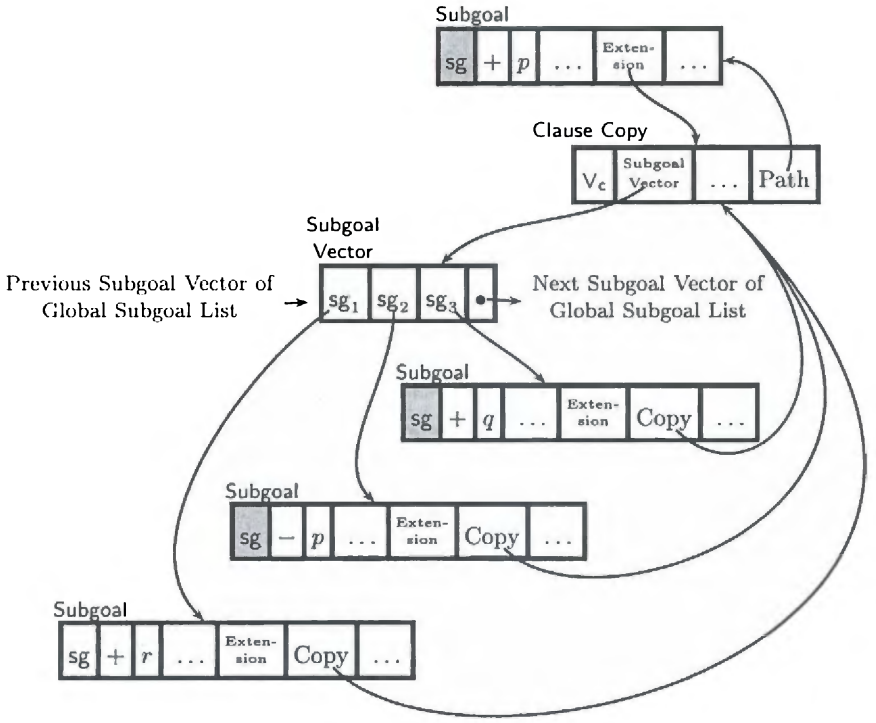


Figure 28: Cross-referencing between clause copy and subgoal data structures.

solve explores the search space by successively applying the extension rule using the elements in the link list of its subgoal argument *sg*. The reduction rule can be incorporated easily as an additional inferential alternative. The procedure **extension** checks the resource bound, adds the linked clause to the proof tree, and modifies the global subgoal list. When a literal can be selected, **solve** is called again with the new subgoals, otherwise a proof has been found and the procedure aborts.

7.5.3. Reuse of clause instances

How can high performance be achieved with such an architecture? It turns out that the most time consuming procedure in this approach is the generation of a new instance of an input clause, which has to be performed in every extension step. One of the main reasons for the high performance of the PTTP based model elimination procedures is that this operation is implemented very efficiently. But the question is, whether it is really necessary to generate a new clause instance in every extension step. Typically, proof search procedures based on model elimination process relatively small tableaux, but a large number of them. That is, in model elimination

theorem proving, the degree of backtracking is extremely high if compared with typical Prolog applications. Many Prolog executions require deep deduction trees including optimizations like tail recursion. For those applications, a new generation of clause instances is indispensable. This striking difference between the deduction trees considered in Prolog and in theorem proving shows that central ingredients of Prolog technology will rarely be needed in theorem proving.

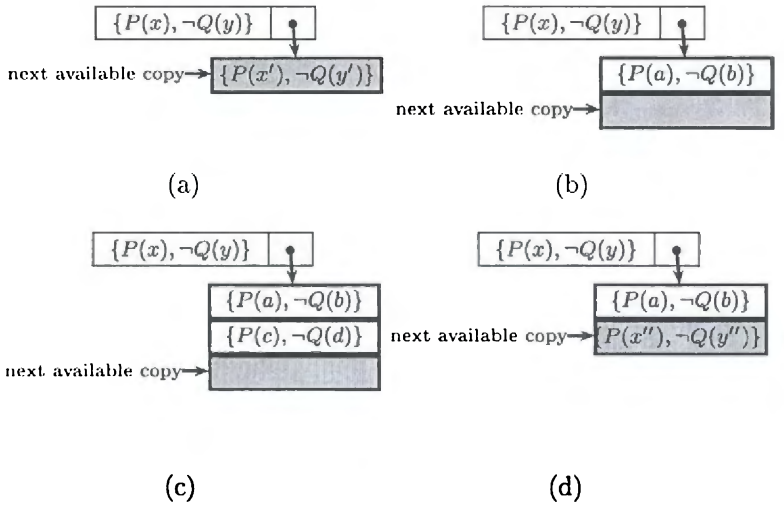


Figure 29: Clause instance creation and availability during backtracking

The key idea for achieving high performance when clause copying is time consuming is the *reuse* of clause instances. The clause copies created once are not discarded when backtracking but are kept in a list of available copies for later reuse, as illustrated with the example in Figure 29. At startup (subfigure (a)), one uninstantiated copy is provided for each clause. This copy is used in an extension step and instantiated, as shown in subfigure (b). Now no other copy is available. When the clause is selected for an extension step again, a new copy has to be created. This situation is shown in subfigure (c). When backtracking occurs during the search process and the extension step that initiated the creation of the copy in subfigure (c) is undone, the copy remains in the list of clause copies and only the pointer to the next available copy is moved backward. This situation is displayed in subfigure (d). This way, over the duration of the proof, a monotonically growing list of clause copies is built and in most cases clause copies can be reused instead of having to be created. As a matter of fact, this requires that all variable bindings are retracted. However, when using destructive unification and the trail concept this can be done very efficiently. Experimental results have shown that with such an architecture inference rates may be obtained that are comparable to the ones achieved with PTTP implementations.

7.6. Existing Model Elimination Implementations

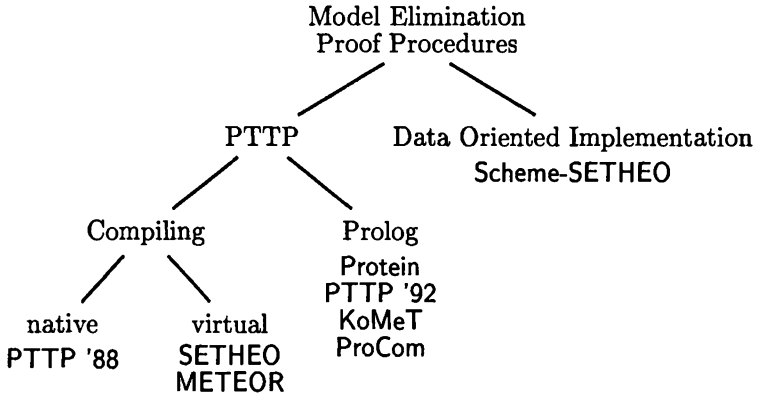


Figure 30: An overview of the architectures of model elimination systems.

In Figure 30, existing implementations of model elimination are classified according to the descriptions given in this section. The references for the listed systems are: PTTP '88 [Stickel 1988], SETHEO [Letz et al. 1992], METEOR [Astrachan and Loveland 1991], Protein [Baumgartner and Furbach 1994], PTTP '92 [Stickel 1992], KoMeT [Bibel, Bruening, Egly and Rath 1994], ProCom [Neugebauer and Petermann 1995, Neugebauer 1995], Scheme-SETHEO (see Section 7.5).

8. Implementation of Refinements by Constraints

When considering the presented tableau refinements such as regularity, tautology, or subsumption-freeness, the question may be raised whether it is possible with reasonable effort to check these conditions after each inference step. Note that a unification operation in one part of a tableau can produce instantiations which may lead to an irregularity, tautology, or subsumed clause in another distant part of the tableau. The structure violation can even concern a closed part of the tableau. Fortunately, there exists a *uniform* and *highly efficient* technique for implementing many of the search pruning mechanisms presented in the previous sections: *Syntactic disequation constraints.*

8.1. Reformulation of Refinements as Constraints

8.1.1. Tautology elimination

Let us demonstrate the technique first using an example of dynamic tautology elimination. Recall that certain input clauses may have tautological instances, which can be avoided as tableau clauses. When considering the transitivity clause

$\neg P(x,y) \vee \neg P(y,z) \vee P(x,z)$, there are two classes of instantiations which may render the formula tautological. Either x and y are instantiated to the same term, or y and z. Obviously, the generation of a tautological instance can be avoided if the unification operation is constrained by forbidding that the respective variables are instantiated to the same terms. In general, this leads to the formulation of *disequation constraints* of the form $s_1, \ldots, s_n \neq t_1, \ldots, t_n$ where the s_i and t_i are terms. Alternatively, one could formulate this instantiation prohibition as a disjunction $s_1 \neq t_1 \vee \cdots \vee s_n \neq t_n$. A disequation constraint is violated if *every* pair $\langle s_i, t_i \rangle$ in the constraint is instantiated to the same term. For the transitivity clause, the two disequation constraints $x \neq y$ and $y \neq z$ can be generated and added to the transitivity formula. The non-tautology constraints for the formulae of a given input set can be generated in a preprocessing phase *before* starting the actual proof process. Afterwards, the tableau construction is performed with *constrained clauses*. Whenever a constrained clause is to be used for tableau expansion, the formula and its constraints are consistently renamed, the tableau expansion is performed with the clause part, and the constraints are added. If the constraints are violated, then a tautological tableau clause has been generated, in which case one can immediately perform backtracking.

8.1.2. Regularity

Regularity can also be captured using disequation constraints. In contrast to non-tautology constraints, however, regularity constraints have to be generated dynamically during the proof search. Whenever a new renamed variant c of a (constrained) clause is attached to a branch in an extension step, then, for every literal L with argument sequence $s_1, \ldots, s_n$ in the clause c and for every branch literal with the same sign and predicate symbol with arguments $t_1, \ldots, t_n$, a disequation constraint $s_1, \ldots, s_n \neq t_1, \ldots, t_n$ must be generated.

8.1.3. Tableau clause subsumption

Tableau clause subsumption is essentially treated in the same manner as tautology elimination. Recall the example from Section 3.3 where in addition to the transitivity clause a unit clause $P(a,b)$ is assumed to be in the input set. Then, the disequation constraint $x, z \neq a, b$ may be generated and added to the transitivity clause. Like non-tautology constraints, non-subsumption constraints can be computed and added to the formulae in the input set before the actual proof process is started.[15] Please note that this mechanism does not capture certain cases of tableau clause subsumption, as demonstrated with the following example. Assume that the transitivity clause and a unit clause $P(f(v), g(v))$ are contained in the input set. In analogy to the other example, a disequation constraint $x, z \neq f(v), g(v)$ could be added to the transitivity formula. But now the constraint contains the variable v, which does not occur in the transitivity clause. Since clauses (and their con-

[15] Note, however, that due to the NP-completeness of subsumption, it might be advisable not to generate *all* possible non-subsumption constraints, since this could involve an exponentially increasing preprocessing time.

straints) are always renamed before being integrated into a tableau, the renaming of the variable v will occur in the constraint only and nowhere else in the tableau. Consequently, this variable can never be instantiated by tableau inference steps, so that the constraint can never be violated and is therefore absolutely useless for search pruning. Clearly, the case of full subsumption cannot be captured in this manner. The constraint mechanism should prevent x and z from being instantiated to any terms which have the *structures* $f(t)$ and $g(t)$, respectively, regardless what t is. This can be conveniently achieved by using *universal variables* in addition to the *rigid variables*. The respective disequation constraint then reads $\forall v\ x, z \neq f(v), g(v)$, which is violated exactly when x and z are instantiated to any terms of the structures $f(s)$ and $g(t)$ with $s = t$.

More general disunification problems are discussed in [Comon and Lescanne 1989].

8.2. Disequation Constraints

After this explanation of the potential of constraints, we will now more rigorously present the framework of disequation constraints, with respect to their use in pruning the proof search in model elimination.

8.1. DEFINITION *(Disequation constraint).* A *disequation constraint* C is either true or of the form $\forall u_1 \cdots \forall u_m\ l \neq r$ $(m \geq 0)$ with l and r being sequences of terms $s_1, \ldots, s_n$ and $t_1, \ldots, t_n$ $(n \geq 0)$, respectively. For any disequation constraint C of the latter form, $l \neq r$ is called the *kernel* of C, n its *length*, $u_1, \ldots, u_m$ its *universal variables*, and the disequation constraints $s_i \neq t_i$ are termed the *subconstraints* of C. Occasionally, we will use the *disjunctive form* of a disequation constraint kernel, which is $s_1 \neq t_1 \vee \cdots \vee s_n \neq t_n$.

An example of a disequation constraint of length $n = 1$ and with one universal variable is

$$\forall z\ f(g(x, a, f(y)), v) \neq f(g(z, z, f(z)), v).$$

Since all considered constraints will be disequation constraints, we will simply speak of constraints in the sequel. Next, we will discuss the meaning of constraint violation.

8.2. DEFINITION *(Constraint violation, equivalence).* No substitution *violates* the constraint true. A substitution σ *violates* a constraint of the form $\forall u_1 \cdots \forall u_m\ l \neq r$ if there is a substitution τ with domain $u_1, \ldots, u_m$ such that $l\tau\sigma = r\tau\sigma$. When a violating substitution exists for a certain constraint, we say that the constraint *can be violated*; a constraint *is violated*, if all substitutions violate it. Two constraints are *equivalent* if they have the same set of violating substitutions.

For example, the substitution $\sigma = \{x/f(a)\}$ violates the constraint $\forall y\ x \neq f(y)$, since $x\tau\sigma = f(y)\tau\sigma$ for $\tau = \{y/a\}$.

8.2.1. Constraint normalization

How may the violation of a constraint be detected in an efficient manner? The basic characteristic of constraints permitting an efficient constraint handling is that these constraints can often be simplified. For example, the complex constraint given after Definition 8.1 is equivalent to the simpler constraint $x, y \neq a, a$, which obviously can be handled more efficiently. Constraints can always be expressed in a specific form.

8.3. DEFINITION *(Constraint in solved form).* A constraint is in *solved form* if it is either true or if its kernel has the form $x_1, \ldots, x_n \neq t_1, \ldots, t_n$ where all variables on the left-hand side are pairwise distinct and non-universal (i.e., do not occur in the quantifier prefix of the constraint), and no variable x_i occurs in terms of the right-hand side.

For example, the constraint $x, y \neq a, a$ is in solved form whereas the equivalent constraint $x, y \neq y, a$ is not. Every constraint can be rewritten into solved form by using the following nondeterministic algorithm.

8.4. DEFINITION *(Constraint normalization).* Let C be any disequation constraint as input. If the constraint is true or if the two sides l and r of its kernel are not unifiable, then the constraint true is a *normal form* of C. Otherwise, let σ be any minimal unifier for l and r that contains no binding of the form x/u where u is a universal variable of C and x is not. Let $\{x_1/t_1, \ldots, x_n/t_n\}$ be the set of all bindings in σ with the x_i being non-universal in C, and let $u_1, \ldots, u_m$ be the universal variables in C that occur in some of the terms t_i. Then the constraint $\forall u_1 \cdots \forall u_m\ x_1, \ldots, x_n \neq t_1, \ldots, t_n$ is a *normal form* of C.

Note that, for preserving constraint equivalence and for achieving a solved form, the use of a minimal unifier is needed in the procedure, employing merely most general unifiers will not always work. Consider, for example, the constraint $f(y) \neq x$ and the most general unifier $\{x/f(x), y/x\}$. This would yield the constraint $x, y \neq f(x), x$ as a normal form, which is not equivalent to $f(y) \neq x$ and which cannot even be violated.

Let us demonstrate the effect of the normalization procedure by applying it to the complex constraint $\forall z\ f(g(x, a, f(y)), v) \neq f(g(z, z, f(z)), v)$ mentioned above. First, we obtain the minimal unifier $\{x/a, z/a, y/a\}$. Next, the binding z/a is deleted, since z is universal, which eventually yields the normal form constraint $x, y \neq a, a$. As shown with this example, some constraint variables may vanish in the normalization process. On the other hand, the length of a constraint may also increase during normalization.

8.5. PROPOSITION. *Any normal form of a constraint C is in solved form and equivalent to C.*

PROOF. If C is true or if the two sides of its kernel are not unifiable, then the normal form of C is true, which is in solved form and equivalent to C. It remains

to consider the case of a constraint $C = \forall u_1 \cdots u_m \; l \neq r$ with unifiable l and r. When normalizing C according to the procedure in Definition 8.4, first, a minimal unifier σ for l and r is computed which does not bind non-universal variables to universal ones. The kernel $l' \neq r'$ of the corresponding normal form C' of C contains exactly the subconstraints $x_i \neq t_i$ for every binding $x_i/t_i \in \sigma$ with non-universal x_i. Since minimal unifiers are idempotent, no variable in the domain of σ occurs in terms of its range. Therefore, C' is in solved form. For considering the equivalence of C and C', first note the following. Since σ is a minimal unifier for l and r, it is idempotent and more general than any unifier for l and r. Therefore, a substitution ρ unifies l and r if and only if $v\rho = s\rho$ for every binding $v/s \in \sigma$. Let now θ be any substitution.

Case 1. If θ violates C, then there exists a substitution τ with domain $\{u_1, \ldots, u_m\}$ and $l\tau\theta = r\tau\theta$. Therefore, for any binding $v/s \in \sigma$, $v\tau\theta = s\tau\theta$, i.e., $\tau\theta$ is a unifier for l' and r'. Let τ' be the set of bindings in τ with domain variables occurring in C'. Then $\tau'\theta$ unifies l' and r'. Consequently, θ violates C'.

Case 2. If θ violates C', then there exists a substitution τ with its domain being the universal variables in C' and $l'\tau\theta = r'\tau\theta$. Let σ' be the set of bindings in σ which bind universal variables. Then, for any binding $v/s \in \sigma$, $v\sigma'\tau\theta = s\sigma'\tau\theta$, and hence $\sigma'\tau\theta$ unifies l and r. Since $\sigma'\tau$ is a substitution with domain $\{u_1, \ldots, u_m\}$, θ violates C. □

8.3. Implementing Disequation Constraints

We will discuss now how the constraint handling can be efficiently integrated into a model elimination proof search procedure. First, we consider the problem of generating constraints in normal form.

8.3.1. Efficient constraint generation

Unification is a basic ingredient of the normalization procedure mentioned above. In the successful implementations of model elimination, a destructive variant of the unification procedure specified in Table 2 is used. If slightly extended, this procedure can also be used for an efficient constraint generation. First, universal variables must be distinguished from non-universal ones. The best way to do this is to extend the internal data structure for variables with an additional cell where it is noted whether the variable is universal or not. The advantage of this approach is that the type of a variable may change during the proof process which nicely goes together with the feature of local variables mentioned in Section 5.4.1. Then, the mentioned unification operation must be modified in order to prevent the binding of a non-universal variable to a universal one. After these modifications, the generation and normalization of a constraint can be implemented efficiently by simply using the new unification procedure, as follows.

8.6. DEFINITION *(Constraint generation).* Given any two sequences l and r of terms that must not become equal by instantiation.

1. Destructively unify l and r and push the substituted variables onto the trail.
2. Collect the respective bindings of the non-universal variables only.
3. Finally undo the unification.

After these operations the term sequences l and r are in their original form, and the collected bindings represent the desired disequation constraint in normal form.

8.3.2. Efficient constraint propagation

During proof search with disequation constraints, every tableau is accompanied by a set of constraints. When an inference step is performed, it produces a substitution which is applied to the tableau. In order to achieve an optimal pruning of the search space, it should be checked after each inference step whether the computed substitution violates one of the constraints of the tableau. If so, the respective inference step can be retracted and we call this a *constraint failure*. If not, the substitution σ has to be propagated to the constraints, i.e., every constraint C has to be replaced by $C\sigma$ before the next inference step is being executed. As a matter of fact, if some of the new constraints can no longer be violated, they should be ignored for the further proof attempt. This is important for reducing the search effort, since normally, a wealth of constraints will be generated during proof search.

If the constraints are always kept in normal form, then the mentioned operations can be performed quite efficiently. Assume, for example, that a substitution $\sigma = \{x/a\}$ is applied to the current tableau. Then it is obvious that all constraints in which x does not occur on the left-hand side may be ignored. In case no constraint of the current tableau is violated by the substitution σ, a new constraint $C\sigma$ has to be created and afterwards normalized for every constraint C containing x on the left-hand side, which is still a considerable effort. In order to do this efficiently, new constraints should not be generated explicitly, but the old constraints should be reused and modified appropriately. For this purpose, it is more comfortable to keep the constraints in disjunctive form. Then, for any such constraint C, only the respective subconstraint $(x \neq t)\sigma$ needs to be normalized to, say C', and the former subconstraint $x \neq t$ in C can be replaced with the subconstraints of C'. This operation may also change the *actual length* of the former constraint. In summary, this results in the following procedure for constraint propagation.

8.7. DEFINITION *(Constraint propagation).* All constraints are assumed to be normalized and in disjunctive form. Suppose a substitution $\sigma = \{x_1/s_1, \ldots, x_n/s_n\}$ is performed during the tableau construction. Then, for every subconstraint $C_i = x_i/t_i$ (i.e., with x_i in the domain of σ) of every constraint C, successively compute the normal form C_i' of $s_i \neq t_i\sigma$ with length, say k, and perform the following operations:

1. If $C_i' = \mathsf{true}$, ignore C for the rest of the proof attempt (it cannot be violated),
2. If $k = 0$, decrement the actual length of C by 1; if the actual length 0 is reached, perform backtracking (the constraint is violated),
3. Otherwise replace C_i with C_i' and modify the actual length of C by adding $k - 1$.

In order to guarantee efficiency, all modifications performed on the constraints have to be stored intermediately and undone on backtracking.

8.3.3. Internal representation of constraints

Obviously, a prerequisite for the efficiency of the constraint handling is a suitable internal representation of the constraints. When analyzing the described constraint handling algorithms, such a representation has to meet the following requirements.

1. After the instantiation of any variable x, a quick access to all subconstraints of the form $x \neq t$ is needed.
2. If a subconstraint C_i of a constraint C is violated, it must be easy to check whether C is violated without considering the other subconstraints of C.
3. Whenever a subconstraint C_i of a constraint C normalizes to true, then it must be easy to deactivate C and all other subconstraints of C.

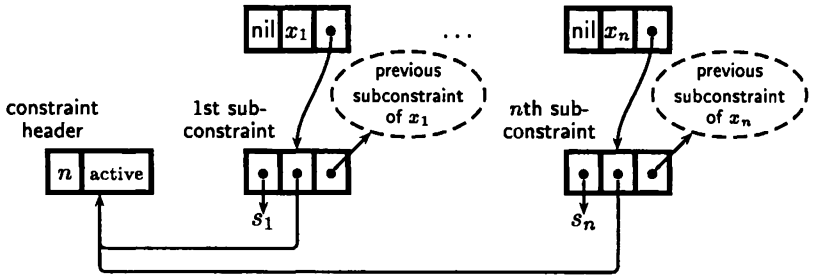


Figure 31: Internal representation of a constraint $x_1, \ldots, x_n \neq s_1, \ldots, s_n$.

This can be achieved by using a data structure as displayed in Figure 31. In order to have immediate access from a variable x to all subconstraints of the form $x \neq t$, it is reasonable to maintain a list of the subconstraints corresponding to each variable. The best solution is to extend the data structure of a variable by a pointer to the last element in its subconstraint list. From this subconstraint the previous subconstraint of the variable x can be accessed, and so forth. (The aforementioned tag which expresses whether a variable is universal or not is omitted in the figure.)

A constraint itself is separated into a *constraint header* and its subconstraints. The header contains the actual length of the constraint and a tag whether the constraint is already true or whether it can still be violated (active). From each subconstraint there is a pointer to the respective constraint header. Now if a subconstraint is violated, then the length counter in the header is decremented by 1. If, on the other hand, a subconstraint normalizes to true, then the tag in the header is set to true. Because of the shared data structure, both modifications are immediately visible and can be used from all other subconstraints of the constraint. Please note that an explicit access from a constraint header to its subconstraints is not needed.

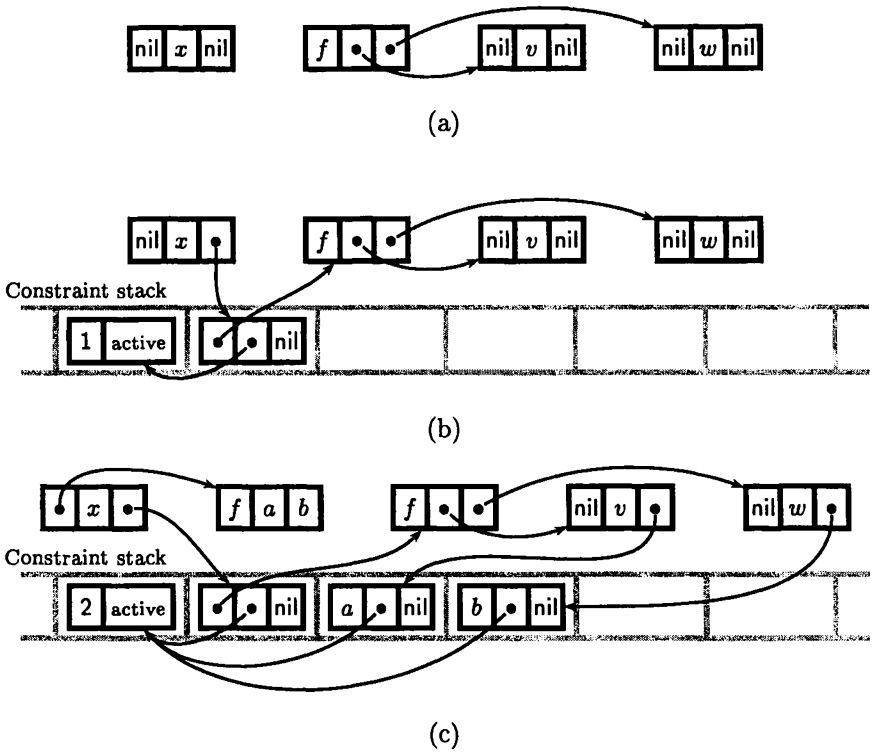


Figure 32: The constraint stack.

It is comfortable to reserve a special part of the memory for the representation of constraints, which we call the *constraint stack*. In order to understand the modifications of the constraint stack during the proof process for the case of a more complex normalization operation, consult Figure 32. Assume we are given a tableau with subgoals $P(x)$ and $\neg Q(x)$ and a predecessor literal $P(f(v,w))$. Assume that no constraints for the variables x, v and w exist (a). Now, a regularity constraint $x \neq f(v,w)$ may be generated, which requires that a constraint header and a subconstraint are pushed onto the constraint stack (b). Assume that afterwards an extension step is performed at the subgoal $\neg Q(x)$ with an entry literal $Q(f(a,b))$. The unifier $\sigma = \{x/f(a,b)\}$ has to be propagated to the constraints. This is done by pushing the two new subconstraints $v \neq a$ and $w \neq b$ onto the constraint stack, which were obtained after normalization. Furthermore, the subconstraint lists of v and w have to be extended. Finally, the counter in the constraint header has to be incremented by 1. Note that nothing has to be done about the old subconstraint $x \neq f(v,w)$. Since the variable x has been bound, the old subconstraint will simply be ignored by all subsequent constraint checks.

8.3.4. Constraint backtracking

The entire mechanism of constraint generation and propagation has to be embedded into the backtracking driven proof search procedure of model elimination. Accordingly, also all modifications performed on the constraint stack and in the subconstraint lists of the variables have to be properly undone when an inference step is retracted. For this purpose, after each inference step and the corresponding modifications in the constraint area, one has to remember the following data.

1. The old length values in the affected constraint headers,
2. the old values (active or true) in the second cells of the affected constraint headers, and
3. the old pointers to the previous subconstraints in the affected variables and subconstraints.

This is exactly the information that has to be stored for backtracking. A comfortable method for doing this would be the use of a *constraint trail* similar to the variable trail, except that here also the old values need to be stored while the variable trail only has to contain the list of bound variables. Additionally, in order to permit the reuse of the constraint stack, one has to remember the top of the constraint stack before each sequence of constraint modifications.

8.3.5. Disequation constraints in Prolog

Some Prolog implementations offer the possibility of formulating disequation constraints. As an example, we consider the Prolog system Eclipse [Wallace and Veron 1993]. Here, using the infix predicate ~= one can formulate syntactic disequation constraints. This permits that constraints resulting from structural tableau conditions can be easily implemented. We describe the method for regularity constraints on the first contrapositive of the transitivity clause

```
p_p(X,Z, P,N) :- N1 = [ p(X,Z) | N ], p_p(X,Y, P,N1), p_p(Y,Z, P,N1).
```

taken from the Prolog example in Section 7.4. We show how regularity can be integrated by modifying the clause as follows.

```
p_p(X,Z, P,N) :- N1 = [ p(X,Z) | N ],
                 not_member(p(X,Z), P),
                 not_member(p(X,Y), N1),
                 not_member(p(Y,Z), N1),
                 p_p(X,Y, P,N1), p_p(Y,Z, P,N1).
```

where not_member is defined as:

```
not_member(_,[ ]).
not_member(E,[F|R]) :- E ~= F, not_member(E,R).
```

With similar methods an easy integration of tautology and subsumption constraints can be achieved. However, when it comes to the integration of more sophisticated constraints such as the ones considered next, it turns out that an efficient Prolog implementation is very difficult to obtain.

8.4. Constraints for Global Pruning Methods

8.4.1. Improving the matings pruning with constraints

In Section 4.1, a method was described which can guarantee that certain permutations of matings are not generated more than once. The idea was to impose an ordering on the literals in the input formula, which is inherited to the tableau nodes. Now, a reduction step from a subgoal N to an ancestor node N' may be avoided if the entry node N'' immediately below N' is smaller than N in the ordering. In fact, this method can also be implemented and even improved by using disequation constraints, as follows. The prohibition to perform a reduction step on N using N' may be reexpressed as a disequation constraint $l \not\equiv r$ where l and r are the argument sequences of the literals at N and N', respectively. Such a constraint does prune not only the respective reduction step, but all tableaux in which the literals at N and N'' become equal by instantiation. In [Letz 1998*b*] it is proven that this extension of the matings pruning preserves completeness, the main reason being that the matings pruning is compatible with regularity.

8.4.2. Failure caching using constraints

The failure caching mechanism described in Section 4.3 can also be implemented using disequation constraints. Briefly, the method requires that when a subgoal N is solved with a solution substitution σ on the variables of the respective path and the remaining subgoals cannot be solved with this substitution, then σ is turned into a failure substitution as defined in Section 4.3 and, for any alternative solution substitution τ for N, σ must not be more general than τ.

8.8. DEFINITION *(Constraint of a failure substitution).* Let $\sigma = \{x_1/t_1, \ldots, x_n/t_n\}$ be a failure substitution generated at a subgoal N and V the set of variables on the path with leaf N in the last tableau in which the subgoal N was selected for an inference step. The *constraint of the failure substitution* σ is the normal form of the constraint $\forall u_1 \cdots \forall u_m \; x_1, \ldots, x_n \not\equiv t_1, \ldots, t_n$ where $u_1, \ldots, u_m$ are the variables occurring in terms of σ that are not in V.

It is straightforward to recognize that a failure substitution σ of a tableau node N is more general than a solution substitution τ of N if and only if the constraint of the failure substitution σ is violated by τ. Consequently, the constraint handling mechanism can be used to implement failure caching. In order to adequately implement failure caching by using constraints, the use of universal variables is also necessary, such as for the case of tableau clause subsumption (Section 8.1.3). This can be seen by considering, for example, a subgoal N with failure substitution $\sigma = \{x/f(z,z)\}$

where z is a variable not occurring in the set V. The constraint of σ is $\forall z\, x \neq f(z,z)$. When N can be solved with a solution substitution $\tau = \{x/f(a,a)\}$, then σ is more general than τ and, in fact, the constraint $\forall z\, x \neq f(z,z)$ is violated by τ. Obviously, it is impossible to capture such a case without universal variables.

8.4.3. Centralized management of constraints

It is apparent that structural constraints resulting from different sources, tautology, regularity, subsumption, or matings, need not be distinguished in the tableau construction. Furthermore, in general, the constraints need not even be tied to the respective tableau clauses, but the constraint information can be kept separate in a special constraint storage space. This also fits in with the method of forgetting closed parts of a tableau and working with subgoal trees instead, because all relevant structure information of the solved part of the tableau is contained in the constraints. However, when structural constraints are used in combination with constraints resulting from failure substitutions, constraints have to be deactivated in certain states of the proof process, as shown in Section 4.3. In this case, it is necessary to take the tableau positions into account at which the respective constraints were generated.

9. Experimental Results

In the previous sections we introduced numerous refinements of model elimination. But, as we are also discussing implementation techniques in this text, knowing about the soundness and completeness of these refinements is not sufficient, it is also of vital importance to know how all these refinements behave when implemented and applied in practice. This section presents a number of experimental results to give an idea of the actual performance of model elimination refinements.

9.1. Test Problem Set and Experimental Environment

All our experiments were conducted in a uniform manner on a subset of the TPTP problem library [Sutcliffe et al. 1994] (version 2.2.1). Starting with the entire set of 4004 TPTP problems, we eliminated all non-clausal problems, all satisfiable problems and all unit equality and pure equality problems. Of the remaining problem set, we selected those problems which at least one of the model elimination search strategies in use with our current e-SETHEO system could solve within 300 seconds. This selection process left us with a test set of 1057 problems. One experiment has been restricted to the test problems containing equality predicates, 431 of which were in our test set.

All experiments were conducted on a Sun Ultra-60 workstation. Each strategy was run on each problem with a time resource of 300 seconds.

Finally, it should be noted that the success of all possible enhancements of model elimination procedures depends on a diligent implementation of the same. The

additional effort induced by a bad implementation can more than nullify the gains of a potentially successful technique.

We have tested the most important refinements of model elimination, the results of these tests can be found in the following sections.

9.2. Regularity

The concept of regularity, as introduced in Section 3.1, is one of the most successful single improvement techniques, as can readily be seen in Figure 33. We compared two strategies using simple iterative deepening without further refinements, one with regularity constraints and one without.

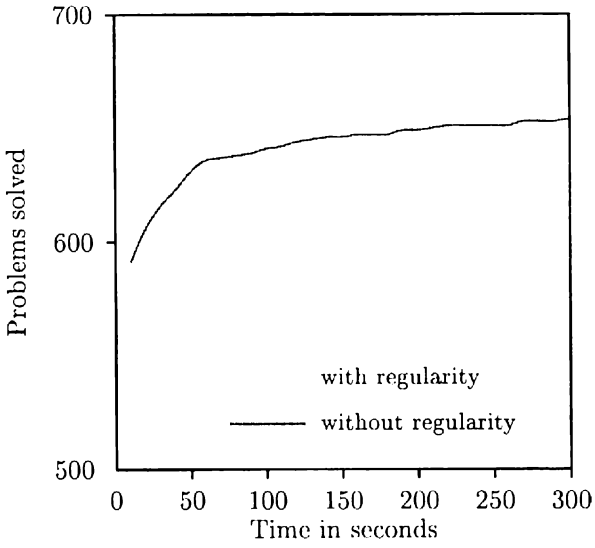


Figure 33: Performance of the depth bound without and with regularity.

9.3. Completeness Bounds

As detailed in Section 2.5, different completeness bounds may be used by the model elimination procedure. We have compared the depth bound, the weighted depth bound and the inference bound. The results of our experiments, however, were ambiguous. In Figure 34, we can see the results on the entire test set.

It is apparent from that figure that the depth bound is the most successful strategy if no knowledge about the problem is available. Yet, distinguishing problems according to some very basic criteria can yield a different picture.

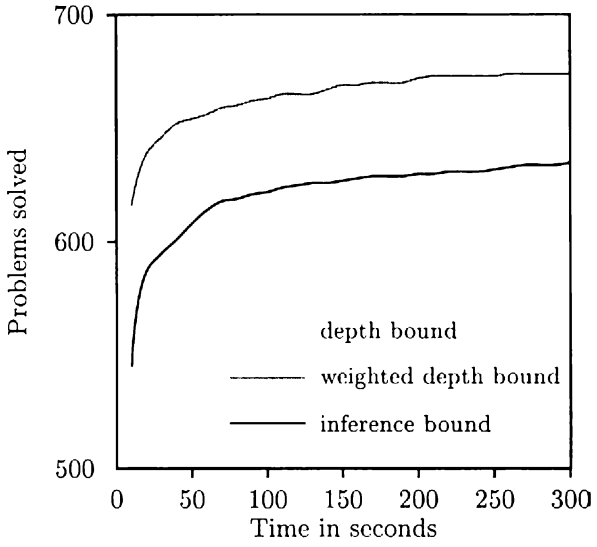


Figure 34: Entire problem set: performance of various completeness bounds with regularity.

Figure 35 depicts the test results for Horn problems only. Here, the weighted depth bound is more successful than the depth bound. A similar result is obtained when restricting the experiments to problems containing the equality predicate, as shown in Figure 36, where we omitted the generally least successful inference bound.

9.4. Relevance Information

As explained in Section 3.5, the use of relevance information can significantly reduce the search space by limiting the number of possible start clauses. The TPTP library [Sutcliffe et al. 1994] provides such relevance information by introducing input clause types. When relevance information is used, only conjecture type clauses are selected as start clauses, whereas in the standard case all negative clauses are potential start clauses.

The result of a comparison between proving with and without relevance information can be seen in Figure 37. As expected, better results are obtained with the use of relevance information.

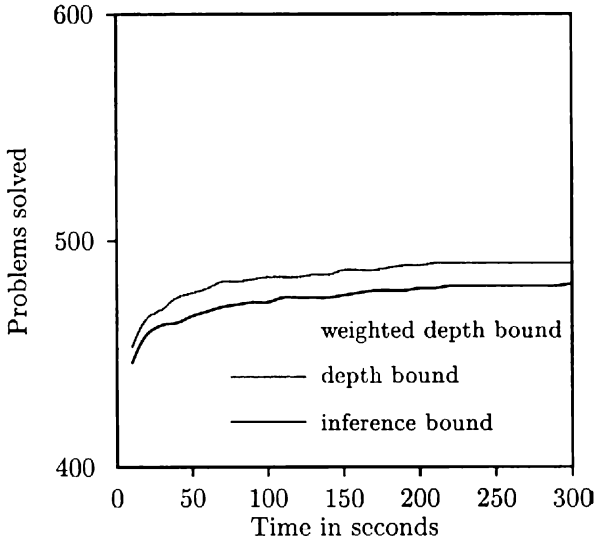


Figure 35: Horn problems: performance of various completeness bounds with regularity.

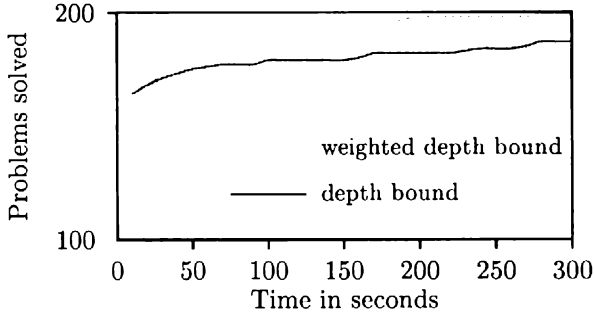


Figure 36: Equality problems: performance of various depth bounds with regularity.

9.5. Failure Caching

The technique of failure caching as described in Section 4.3 has been implemented in SETHEO with the use of constraints, as described in Section 8.4.2. As becomes apparent from Figure 38, the use of this technique has a considerable influence on the test results.

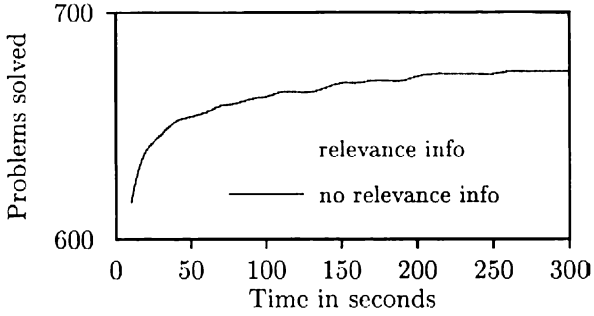


Figure 37: The effect of the use of relevance information (with weighted depth bound and regularity).

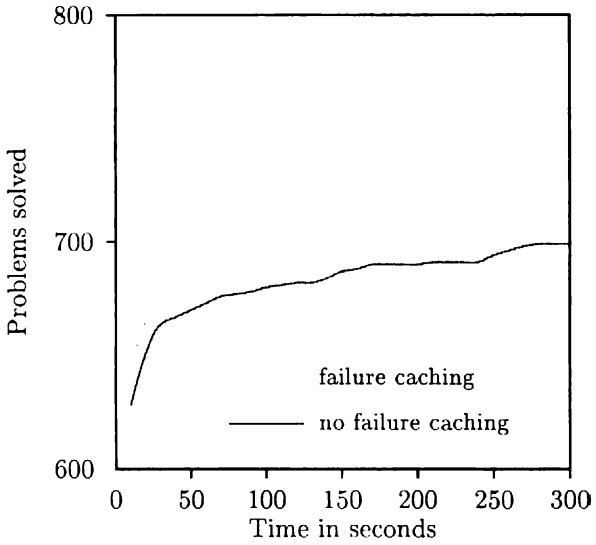


Figure 38: The effect of failure caching (with weighted depth bound and regularity).

9.6. Folding Up

Introduced in Section 5.2, the folding up technique has also been implemented as a model elimination refinement in SETHEO and has turned out to be very successful. Figure 39 shows three curves: The lowest curve indicates the proofs found with the use of regularity only, while the middle curve gives the number of proofs using regularity and folding up. Finally, the curve with the highest number of proofs shows the result of combining folding up with the full use of constraints (enabling both regularity and failure caching).

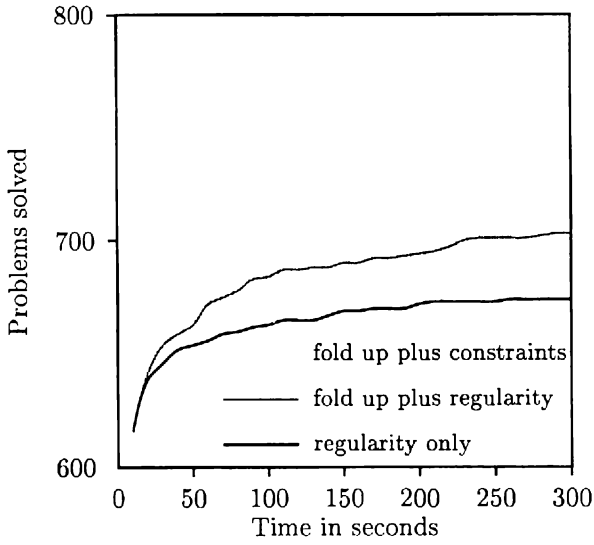


Figure 39: Enhancements through the use of the folding up technique and failure caching (with weighted depth bound).

9.7. Dynamic Subgoal Reordering

The concept of subgoal reordering was introduced in Section 2.6.1. As can be seen in Figure 40, even a local (this means only within the selected clause) subgoal selection according to the fewest solution principle can have a positive effect.

9.8. Summary

The experimental results presented in this section show that the refinements described in this chapter are applicable in automated theorem proving and that they may also be combined in several ways to accumulate the gains of the individual refinements. Among all these refinements, regularity and failure caching stand out in particular as being the most important and successful ones.

10. Outlook

The model elimination or connection tableau approach is an efficient and successful paradigm in automated deduction. There are, however, a number of deficiencies, which have to be addressed in the future. One of the fundamental weaknesses of connection tableaux is the handling of equality. The naïve approach to simply add the congruence axioms of equality, suffers from the weakness that equality specific

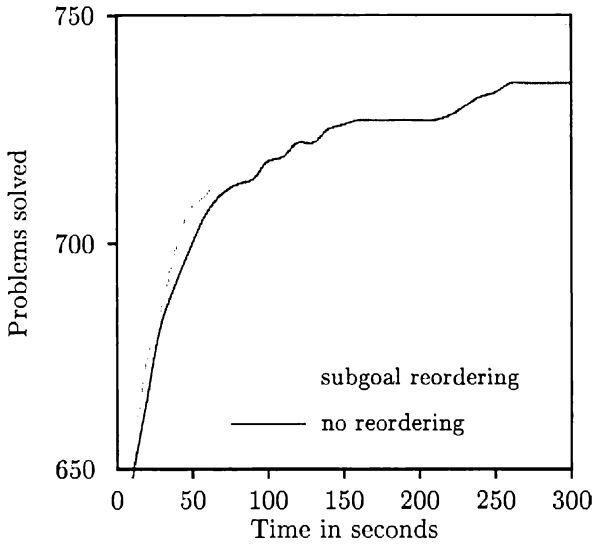


Figure 40: Performance of the weighted depth bound with constraints, with and without dynamic subgoal reordering.

redundancy elimination techniques are ignored. The most successful paradigm for treating equality in saturation-based theorem proving, *ordered paramodulation*, is not compatible with connection tableaux. There have been attempts to integrate *lazy paramodulation*, a variant of paramodulation without orderings which is compatible with model elimination. This method is typically implemented by means of a transformation (like Brand's modification method [Brand 1975]), which eliminates the equality axioms and compiles certain equality inferences into the formula. A certain search space pruning might be obtained by using limited ordering conditions [Bachmair, Ganzinger and Voronkov 1998], preferably implemented as ordering constraints. This would fit well with the constraint technology applicable in connection tableaux.

Another, more general weakness of the search procedure is that it typically performs poorly on formulae with relatively long proofs. On the one hand, this has directly to do with the methodology of iterative-deepening search. On the other hand, when proofs are becoming longer, the goal-orientedness loses its reductive power. To prove difficult formulae in one big leap by reasoning backwards from the conjecture is very difficult. An interesting perspective here is the use of *lemmata*, intermediate results typically deduced in a forward manner from the axioms. Some progress has been made in this direction by the development of powerful filtering techniques.

A further interesting line of research could be the use of pruning methods based on semantic information. One could, for example, use small models of the axioms

in order to detect the unsolvability of certain subgoals. Finally, the consideration of a confluent and possibly even nondestructive integration of connection conditions into the tableau framework definitely deserves attention.

Acknowledgments

We would like to thank Peter Baumgartner and Uwe Petermann for their valuable comments on an earlier version of this chapter. Furthermore we want to thank Herbert Stenz for his exceptionally thorough proofreading and Rajeev Goré for his linguistic advice. This work was partially funded by the *Deutsche Forschungsgemeinschaft (DFG)* as part of the *Schwerpunktprogramm Deduktion* and the *Sonderforschungsbereich (SFB) 342.*

Bibliography

AHO A. V., SETHI R. AND ULLMAN J. D. [1986], *Compilers – Principles, Techniques, and Tools*, Addison-Wesley, Reading, MA.

AHO A. V. AND ULLMAN J. D. [1977], *Principles of Compiler Design*, Addison-Wesley, Reading, MA. See also the widely expanded subsequent book [Aho, Sethi and Ullman 1986].

ANDREWS P. B. [1981], 'Theorem proving through general matings', *Journal of the ACM* **28**, 193–214.

ASTRACHAN O. L. AND LOVELAND D. W. [1991], METEORs: High performance theorem provers using model elimination, Technical Report DUKE–TR–1991–08, Department of Computer Science, Duke University.

ASTRACHAN O. AND STICKEL M. [1992], Caching and Lemmaizing in Model Elimination Theorem Provers, *in* D. Kapur, ed., 'Proceedings, 11th International Conference on Automated Deduction (CADE-11), Saratoga Springs, NY, USA', Vol. 607 of *LNAI*, Springer, Berlin, pp. 224 – 238.

BAAZ M. AND LEITSCH A. [1992], 'Complexity of resolution proofs and function introduction', *Annals of Pure and Applied Logic* **57**(3), 181–215.

BACHMAIR L. AND GANZINGER H. [2001], Resolution theorem proving, *in* A. Robinson and A. Voronkov, eds, 'Handbook of Automated Reasoning', Vol. I, Elsevier Science, chapter 2, pp. 19–99.

BACHMAIR L., GANZINGER H. AND VORONKOV A. [1998], Elimination of Equality via Transformation with Ordering Constraints, *in* C. Kirchner and H. Kirchner, eds, 'Proceedings, 15th International Conference on Automated Deduction (CADE-15), Lindau, Germany', Vol. 1421 of *LNAI*, Springer, Berlin, pp. 175–190.

BAUMGARTNER P. [1998], Hyper Tableau — The Next Generation, *in* H. de Swart, ed., 'Proceedings of the International Conference on Automated Reasoning with Analytic Tableaux and Related Methods (TABLEAUX-98), Oisterwijk, The Netherlands', Vol. 1397 of *LNAI*, pp. 60–76.

BAUMGARTNER P. AND BRÜNING S. [1997], 'A Disjunctive Positive Refinement of Model Elimination and its Application to Subsumption Deletion', *Journal of Automated Reasoning* **19**(2), 205–262.

BAUMGARTNER P., EISINGER N. AND FURBACH U. [1999], A Confluent Connection Calculus, *in* H. Ganzinger, ed., 'Proceedings of the 16th International Conference on Automated Deduction (CADE-16)', Vol. 1632 of *LNAI*, Springer, Berlin, pp. 329–343.

BAUMGARTNER P. AND FURBACH U. [1994], PROTEIN: A PROver with a Theory Extension INterface, *in* A. Bundy, ed., 'Proceedings of the 12th International Conference on Automated Deduction (CADE-12)', Vol. 814 of *LNAI*, Springer, Berlin, pp. 769–773.

BECKERT B. [1998], Integrating and Unifying Methods of Tableau-based Theorem Proving, PhD thesis, University of Karlsruhe, Department of Computer Science.

BECKERT B. AND HÄHNLE R. [1992], An Improved Method for Adding Equality to Free Variable Semantic Tableaux, *in* D. Kapur, ed., 'Proceedings, 11th International Conference on Automated Deduction (CADE-11), Saratoga Springs, NY, USA', LNCS 607, Springer, Berlin, pp. 507–521.

BECKERT B. AND HÄHNLE R. [1998], Analytic Tableaux, *in* W. Bibel and P. H. Schmitt, eds, 'Automated Deduction — A Basis for Applications', Vol. I: Foundations, Kluwer, Dordrecht, pp. 11–41.

BENKER H., BEACCO J. M., BESCOS S., DOROCHEVSKY M., JEFFRÉ T., PÖHLMANN A., NOYÉ J., POTERIE B., SEXTON A., SYRE J. C., THIBAULT O. AND WATZLAWIK G. [1989], KCM: A Knowledge Crunching Machine, *in* M. Yoeli and G. Silberman, eds, 'Proceedings of the 16th Annual International Symposium on Computer Architecture, Jerusalem, Israel', IEEE Computer Society Press, New York, pp. 186–194.

BIBEL W. [1987], *Automated Theorem Proving*, second revised edn, Vieweg, Braunschweig.

BIBEL W., BRUENING S., EGLY U. AND RATH T. [1994], KoMeT, *in* 'Proceedings, 12th International Conference on Automated Deduction (CADE-12), Nancy, France', Vol. 814 of *LNAI*, Springer, Berlin, pp. 783–787.

BILLON J.-P. [1996], The disconnection method: a confluent integration of unification in the analytic framework, *in* P. Migliolo, U. Moscato, D. Mundici and M. Ornaghi, eds, 'Proceedings of the 5th International Workshop on Theorem Proving with Analytic Tableaux and Related Methods (TABLEAUX)', Vol. 1071 of *LNAI*, Springer, Berlin, pp. 110–126.

BOY DE LA TOUR T. [1990], Minimizing the Number of Clauses by Renaming, *in* M. E. Stickel, ed., '10th International Conference on Automated Deduction (CADE-10), Kaiserslautern, Germany', LNCS, Springer, Berlin, pp. 558–572.

BRAND D. [1975], 'Proving Theorems with the Modification Method', *SIAM Journal on Computing* **4**(4), 412–430.

COMON H. AND LESCANNE P. [1989], 'Equational Problems and Disunification', *Journal of Symbolic Computation* **7**(3–4), 371–425.

CORBIN J. AND BIDOIT M. [1983], A Rehabilitation of Robinson's Unification Algorithm, *in* 'Information Processing', North Holland, Amsterdam, pp. 909–914.

EDER E. [1984], An Implementation of a Theorem Prover Based on the Connection Method, *in* W. Bibel and B. Petkoff, eds, 'Proceedings of the International Conference on Artificial Intelligence: Methodology, Systems and Applications (AIMSA), Varna, Bulgaria', North Holland, Amsterdam, pp. 121–128.

FITTING M. C. [1990], *First-Order Logic and Automated Theorem Proving*, Springer, Berlin.

FITTING M. C. [1996], *First-Order Logic and Automated Theorem Proving*, second revised edn, Springer, Berlin.

GENTZEN G. [1935], 'Untersuchungen über das logische Schließen', *Mathematische Zeitschrift* **39**, 176–210, 405–431. English translation in M. E. Szabo, editor, *The Collected Papers of Gerhard Gentzen*, pages 68–131. North Holland, Amsterdam, 1969.

GOLLER C., LETZ R., MAYR K. AND SCHUMANN J. M. P. [1994], SETHEO V3.2: Recent developments, *in* A. Bundy, ed., 'Proceedings of the 12th International Conference on Automated Deduction (CADE-12)', Vol. 814 of *LNAI*, Springer, Berlin, pp. 778–782.

HÄHNLE R. [2001], Tableaux and related methods, *in* A. Robinson and A. Voronkov, eds, 'Handbook of Automated Reasoning', Vol. I, Elsevier Science, chapter 3, pp. 100–178.

HARRISON J. [1996], Optimizing proof search in model elimination, *in* M. A. McRobbie and J. K. Slaney, eds, 'Proceedings of the 13th International Conference on Automated Deduction (CADE-13)', Vol. 1104 of *LNAI*, Springer, Berlin, pp. 313–327.

KLINGENBECK S. AND HÄHNLE R. [1994], Semantic Tableaux with Ordering Restrictions, *in* A. Bundy, ed., 'Proceedings, 12th International Conference on Automated Deduction (CADE-12), Nancy, France', LNCS 814, Springer, Berlin, pp. 708–722.

KORF R. E. [1985], Iterative-Deepening-A: An Optimal Admissible Tree Search, *in* A. Joshi, ed., 'Proceedings of the 9th International Joint Conference on Artificial Intelligence', Morgan Kaufmann, Los Angeles, CA, pp. 1034–1036.

KOWALSKI R. A. AND KUEHNER D. [1970], Linear resolution with selection function, Technical report, Metamathematics Unit, Edinburgh University, Edinburgh, Scotland.

KOWALSKI R. AND KUEHNER D. [1971], 'Linear Resolution with Selection Function', *Artificial Intelligence* **2**, 227–260.

LETZ R. [1993], First-order calculi and proof procedures for automated deduction, PhD thesis, Technische Hochschule Darmstadt, Darmstadt, Germany.

LETZ R. [1998*a*], Clausal Tableaux, *in* W. Bibel and P. H. Schmitt, eds, 'Automated Deduction — A Basis for Applications', Vol. I: Foundations, Kluwer, Dordrecht, pp. 43–72.

LETZ R. [1998*b*], Using Matings for Pruning Connection Tableaux, *in* C. Kirchner and H. Kirchner, eds, 'Proceedings, 15th International Conference on Automated Deduction (CADE-15), Lindau, Germany', Vol. 1421 of *LNAI*, Springer, Berlin, pp. 381–396.

LETZ R. [1999], First-Order Tableaux Methods, *in* M. D'Agostino, D. Gabbay, R. Hähnle and J. Posegga, eds, 'Handbook of Tableau Methods', Kluwer, Dordrecht, pp. 125–196.

LETZ R., MAYR K. AND GOLLER C. [1994], 'Controlled Integration of the Cut Rule into Connection Tableau Calculi', *Journal of Automated Reasoning* **13**(3), 297–338.

LETZ R., SCHUMANN J. AND BAYERL S. [1989], SETHEO: A SEquentiell THEOrem Prover for first order logic, Technical Report FKI-97-89, Technische Universität München, Munich, Germany.

LETZ R., SCHUMANN J., BAYERL S. AND BIBEL W. [1992], 'SETHEO: A High-Performance Theorem Prover', *Journal of Automated Reasoning* **8**(2), 183–212.

LOVELAND D. W. [1968], 'Mechanical Theorem Proving by Model Elimination', *Journal of the ACM* **15**(2), 236–251. Reprinted in: [Siekmann and Wrightson 1983].

LOVELAND D. W. [1969], 'A Simplified Format for the Model Elimination Theorem-Proving Procedure', *Journal of the ACM* **16**(3), 349–363.

LOVELAND D. W. [1972], 'A Unifying View of Some Linear Herbrand Procedures', *Journal of the ACM* **19**(2), 366–384.

LOVELAND D. W. [1978], *Automated theorem proving: A logical basis*, North Holland, Amsterdam.

MAYR K. [1993], Refinements and Extensions of Model Elimination, *in* A. Voronkov, ed., 'Proceedings of the 4th International Conference on Logic Programming and Automated Reasoning (LPAR'93), St. Petersburg, Russia', Vol. 698 of *LNAI*, Springer, Berlin, pp. 217–228.

MOSER M., IBENS O., LETZ R., STEINBACH J., GOLLER C., SCHUMANN J. AND MAYR K. [1997], 'SETHEO and E-SETHEO—The CADE-13 Systems', *Journal of Automated Reasoning* **18**(2), 237–246.

NEITZ W. [1995], Untersuchungen zum selektiven Backtracking in zielorientierten Kalkülen des automatischen Theorembeweisens, PhD thesis, University of Leipzig.

NEUGEBAUER G. [1995], ProCom/CaPrI *and the Shell* ProTop. *User's Guide*, FB IMN, HTWK Leipzig. ftp://www.koralle.imn.htwk-leipzig.de/pub/ProCom/procom.html.

NEUGEBAUER G. AND PETERMANN U. [1995], Specifications of Inference Rules and their Automatic Translation, *in* P. Baumgartner, R. Hähnle and J. Posegga, eds, 'Proceedings of the 4th International Workshop on Theorem Proving with Analytic Tableaux and Related Methods (TABLEAUX)', Vol. 918 of *LNAI*, Springer, Berlin, pp. 185–200.

PELLETIER F. J. AND RUDNICKI P. [1986], 'Non-obviousness', *AAR Newsletter* (6), 4–5.

PLAISTED D. A. [1984], The Occur-check Problem in Prolog, *in* '1984 International Symposium on Logic Programming', IEEE Computer Society Press, New York.

PLAISTED D. A. AND GREENBAUM S. [1986], 'A Structure Preserving Clause Form Translation', *Journal of Symbolic Computation* **2**(3), 293–304.

PRAWITZ D. [1960], 'An improved proof procedure', *Theoria* **26**, 102–139. Reprinted in [Siekmann and Wrightson 1983].

REEVES S. V. [1987], 'Adding Equality to Semantic Tableaux', *Journal of Automated Reasoning* **3**, 225–246.

SCHUMANN J. [1991], Efficient Theorem Provers based on an Abstract Machine, PhD thesis, Technische Universität München.

SCHUMANN J. AND LETZ R. [1990], PARTHEO: A High-Performance Parallel Theorem Prover, *in* M. E. Stickel, ed., 'Proceedings, 10th International Conference on Automated Deduction (CADE-10), Saratoga Springs, NY, USA', LNAI 449, Springer, Berlin, pp. 40–56.

SHOSTAK R. E. [1976], 'Refutation Graphs', *Artificial Intelligence* **7**, 51–64.

SIEKMANN, J. AND WRIGHTSON, G., EDS [1983], *Automation of Reasoning*, Springer, Berlin. Two volumes.

SMULLYAN R. [1968], *First-Order Logic*, Springer, Berlin.

STICKEL M. E. [1984], A Prolog Technology Theorem Prover, *in* '1984 International Symposium on Logic Programming', IEEE Computer Society Press, New York.

STICKEL M. E. [1988], A Prolog Technology Theorem Prover, *in* E. Lusk and R. Overbeek, eds, '9th International Conference on Automated Deduction (CADE-9), Argonne, Ill', LNCS, Springer, Berlin, pp. 752–753.

STICKEL M. E. [1992], 'A Prolog technology theorem prover: a new exposition and implementation in Prolog', *Theoretical Computer Science* **104**, 109–128.

SUTCLIFFE G. AND SUTTNER C. B. [1998], The CADE-14 ATP System Competition, Technical Report JCU-CS-98/01, Department of Computer Science, James Cook University.
URL: *http://www.cs.jcu.edu.au/ftp/pub/techreports/98-01.ps.gz*

SUTCLIFFE G., SUTTNER C. AND YEMENIS T. [1994], The TPTP problem library, *in* A. Bundy, ed., 'Proceedings, 12th International Conference on Automated Deduction (CADE-12), Nancy, France', LNCS 814, Springer, Berlin, pp. 708–722. Current version available on the *World Wide Web* at the URL `http://www.cs.jcu.edu.au/ftp/users/GSutcliffe/TPTP.HTML`.

TAKI K., YOKOTA M., YAMAMOTO A., NISHIKAWA H., UCHIDA S., NAKASHIMA H. AND MITSUISHI A. [1984], Hardware Design and Implementation of the Personal Sequential Inference Machine (PSI), *in* 'Proceedings of the International Conference on Fifth Generation Computer Systems', ICOT, Tokyo, Japan, pp. 398–409.

TSEITIN G. [1970], 'On the Complexity of Proofs in Propositional Logics', *Seminars in Mathematics* **8**.

VLAHAVAS I. AND HALATSIS C. [1987], A New Abstract Prolog Instruction Set, *in* 'Expert systems and their applications (Proceedings)', Avignon, pp. 1025–1050.

WALLACE M. AND VERON A. [1993], Two problems – two solutions: One system – ECLiPSe, *in* 'Proceedings IEE Colloquium on Advanced Software Technologies for Scheduling', London.

WARREN D. H. D. [1983], An Abstract PROLOG Instruction Set, Technical Report 309, Artificial Intelligence Center, Computer Science and Technology Division, SRI International, Menlo Park, CA.

WEIDENBACH C. [2001], Combining superposition, sorts and splitting, *in* A. Robinson and A. Voronkov, eds, 'Handbook of Automated Reasoning', Vol. II, Elsevier Science, chapter 27, pp. 1965–2013.

Index

S

T

U

V