

# TERMINATOR

Grigorios Antoniou, Hans Jirgen Ohlbach

Institut fur Informatik I, University of Karlsruhe

## ABSTRACT

The Markgraf Karl Refutation Procedure (MKR-Procedure) is an automated theorem prover for sorted logic, based on an extended clause graph calculus, currently under development at the University of Karlsruhe. This paper describes the **TERMINATOR** module, a component of the MKR-Procedure, which is essentially a very fast algorithm for the search for unit refutations. The **TERMINATOR** is used as a fast pre-theorem prover as well as an integral component of the system, and is called for different tasks during the search for a proof.

## Introduction

There are presently two main groups of theorem proving calculi: In those of the first kind the theorem prover starts with a given set of logical formulas and creates new formulas by the application of certain deduction rules as for example resolution, paramodulation, natural deduction rules etc, until the theorem or a refutation has been derived. Recently new calculi like Andrews' mating calculus [1] or Bibel's matrix calculus [2] have been developed which initially do not deduce any new formulas, but only test certain path conditions ensuring satisfiability or unsatisfiability of the initial formula set.

Kowalski's connection graph proof procedure [8] in its original formulation is of the first kind: A connection graph consists of the nodes labeled by clauses in conjunctive normalform and links (connections) between complementary unifiable literals representing possible resolution steps. A deduction is performed by the selection of a link, creating the corresponding resolvent, inserting it into the graph and deleting the selected link, potentially causing further deletions of links and clauses. This proof procedure can be transformed into a calculus of the second kind: If the clause set is unit refutable, i.e. the empty clause can be derived by successive resolution steps with one-literal clauses, the clause graph has to contain a special subtree (refutation tree or terminator situation) which just represents this chain of unit resolutions.

## Examples:

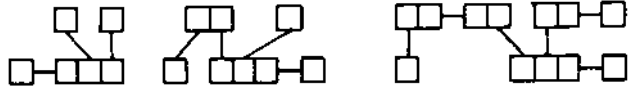


Fig. 1 a) b) c)

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

In general a connection graph is a refutation tree if:

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

A unit refutable clause set  $S$  is unsatisfiable, iff there exists a refutation tree for the factored set  $S$  [6]!

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

## The Algorithm

The first attempt to implement an n-level terminator algorithm used a recursive technique: A non-unit clause  $C$  was selected and examined for a one-level situation. If this first test failed, the algorithm was called recursively for other non-unit clauses connected to  $C$ , trying to find a chain of unit resolutions which resolves away all literals except for that one connected to  $C$ , and the one-level test for  $C$  was performed once again taking

advantage of this new information. Unfortunately the algorithm was rather inefficient because no results of former TERMINATOR calls had been stored, and therefore the same part of the clause graph had to be examined again and again. Hence only the  $n = 1$  case was within practical limits.

Whereas the old algorithm worked from inside the refutation tree to the leaf nodes (unitclauses), our new algorithm works just in the opposite direction and stores all information it has generated for later use. We shall present the working of the algorithm using a few examples, and a full description is given in [10].

The input clauses are divided into the set of unitclauses (UNITS) and the set of non-unitclauses (NON-UNITS). NON-UNITS are sorted according to increasing literal number. For each element  $C$  of NON-UNITS, of which every literal except at most one is connected to at least one element of UNITS a compatibility test is performed. An example will illustrate this test:

Suppose  $C$  has four literals:  $C = \langle L1 \ L2 \ L3 \ L4 \rangle$ . Each literal  $L_i$  may be connected to a set of unitclauses complementary to  $L_i$  and let  $U_i$  be the set of unifiers associated with these connections.

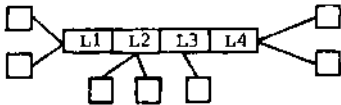


fig. 2

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

$$U_{12} = U_1 * U_2 = \{\sigma \mid \sigma = \sigma_1 * \sigma_2, \sigma_1 \in U_1, \sigma_2 \in U_2\}$$

where  $\sigma_1 * \sigma_2$  is a most general merge substitution (resp. unifier, see below) of  $\sigma_1$  and  $\sigma_2$ . Similarly we calculate  $U_{123} = U_{12} * U_3$  and  $U_{1234} = U_{123} * U_4$ . If  $U_{1234} \neq \emptyset$ , then every element of  $U_{1234}$  allows us to resolve away the whole clause, i.e. a proof is found. If  $U_{1234} = \emptyset$  but  $U_{123} \neq \emptyset$  we apply every substitution of  $U_{123}$  to literal  $L4$  thus generating new unitclauses and insert them into the UNITS-list.

With this step we deduce in fact new formulas, but we only use them as a compact representation of the merged unifiers with three additional advantages:

1. A very simple and fast subsumption test [10] between the unitclauses allows us to detect whether the result of a unit resolution chain is an instance of the result of another chain.
2. The treatment of the input unitclauses is just the same as the treatment of the deduced ones which simplifies the data- and control structures.
3. Heuristic valuation of the unitclauses allows for further pruning the search space.

If  $U_{1234} \neq \emptyset$ , we create new unitclauses in the same way applying  $U_{12} * U_4$  to  $L3$ ,  $U_1 * U_{34}$  to  $L2$  and  $U_{234}$  to  $L1$ . The links of the initial graph now pro-

vide the information to be able to attach all the new unitclauses to complementary literals in other clauses, (or respectively other literals in the same clause, if it is selfresolving).

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

The TERMINATOR is called from the MKR-Procedure at the beginning of the search for a proof and, if it does not find an immediate proof it is called again each time a certain number of steps has been performed by other components of the system, hence the overall behaviour amounts to a tightly controlled and sophisticated bidirectional search. In particular the other link selection mechanisms in the system prefer those resolution and paramodulation steps which are expected to transform a non-unit refutable clause graph into a unit refutable one. A second application of the TERMINATOR is the generation of unitclauses, for instance the rewrite component of the system can decide that a specific unit equation might be useful as a rewrite rule and charges the TERMINATOR to deduce, if possible, the desired equation from a conditional equation.

### The Compatibility Test

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

A unifier for two substitutions  $\sigma$  and  $\tau$  is a substitution  $\lambda$  such that

$$\lambda \circ \sigma = \lambda \circ \tau$$

$\sigma * \lambda := \lambda \circ \sigma = \lambda \circ \tau$  is called the merge substitution of  $\sigma$  and  $\tau$ .

There are fast unification algorithms known for the unification of terms, the most recent are even linear [7]. The unification of substitutions was first investigated in [11]. But for  $|A| = n$  and  $|B| = m$  you still have to perform  $n \cdot m$  such operations and in many applications this number may be greater than 105 for this reason the known methods are out of the question. Hence we have developed a very fast merging algorithm for large sets of substitutions which exploits special conditions present in the terminator problem. The goal is to find a representation for the substitutions which allows the direct access from each substitution in set  $A$  to those in set  $B$  which are compatible with  $\sigma$ .

The working of the algorithm will be illustrated with the following examples; a full description is given in [10]

**Example 1**

```

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

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

Since only links between unitclauses and non-unitclauses are being considered, we know that the single literal of the unitclause is not present in the resolvent after resolution upon such a link. Therefore only the substitution components for variables of the non-unitclause are relevant for the resolvent. Hence it is admissible to discard all other substitution components. Because of this it is possible to represent the remaining components in a table using a fixed ordering of the variables occurring in the non-unitclause. This representation will be called a T-representation.

For example 1 we obtain:

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

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

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

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

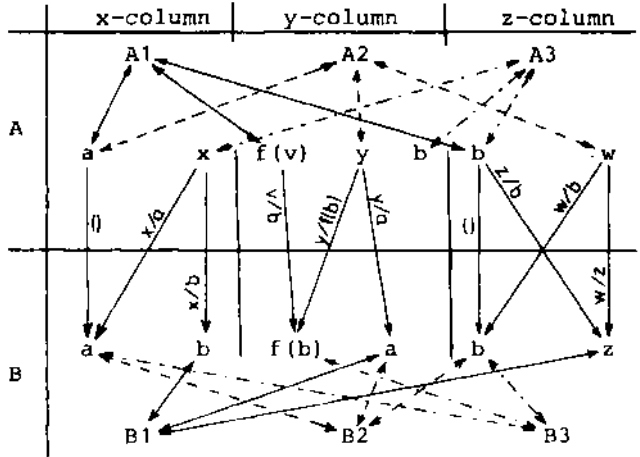
**Example 2: T-representation:**

A	x	y	z	B	x	y	z
A1	a	f(v)	b	B1	b	a	z
A2	a	y	w	B2	a	a	b
A3	x	b	b	B3	a	f(b)	b

n = m = 3      v,w,x,y,z are variables .

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

**P-representation of example 2:**



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

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

For example 2 we obtain:

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

All those substitutions in B with exactly p compatible single-term-unifiers are compatible with a, where p is the length of the T-representation (= number of variables in the non-unit clause). (In example 2:p=3.) For example 2 we obtain three compatible pairs of substitutions:

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

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

## Summary

With the terminator algorithm three main goals have been achieved:

1. In reality nothing has to be done twice, because all relevant intermediate results (merged substitutions) can be stored without requiring too much space.
2. The compatibility test is enormously reduced in complexity, allowing the fast unification of millions of pairs of substitutions.
3. Arbitrary rules for pruning the search space by deleting unitclauses can easily be incorporated into the algorithm.

These are the main advantages compared to similar methods for the extraction of refutation trees respectively graphs, like S. Sickel's graph unrolling [12] or Chang and Slagle's usage of rewrite rules [4]. Although their calculi are complete for non-unit refutable clause sets as well, their practical usefulness is questionable, unless their efficiency problems are solved.

The implementation of the terminator algorithm has considerably increased the performance of the MKR-Procedure. For instance we have solved SAM's Lemma, a famous problem in automatic theorem proving, which was first proved by an interactive theorem prover [5], and later by the automatic theorem prover at Argonne National Lab. It had to deduce 22000 clauses before finding a proof [3]. The TERMINATOR generated only 190 unitclauses before the proof was found [9].

## References

- [1] P.B. Andrews  
Theorem proving via general matings.  
J. ACM 28:2 (1981) 193-214
- [2] W. Bibel  
On Matrices with Connections.  
J. ACM 28:4 (1981) 633-645.
- [3] McCharen, Overbeck, Wos  
Problems and Experiments for and with Automatic Theorem Proving Programs.  
IEEE Transactions on Computers C-25:8 (1976) 773-782.
- [4] C.L. Chang, J.R. Slagle  
Using Rewrite Rules for Connection Graphs to Prove Theorems.  
Artificial Intelligence 12 (1979) 159-180.
- [5] Guard, Oglesby, Bennet, Settle  
Semi-Automated Mathematics  
J. ACM 16:1 (1969) .
- [6] M.C. Harrison and N. Rubin  
Another generalization of resolution.  
J. ACM 25:3 (1978) 341-351.
- [7] Kapur, Krishnamoorthy, Narendra  
A linear Algorithm for Unification.  
General Electric Rep. no 82CRD-100  
New York (1982).
- [8] R. Kowalski  
A Proof Procedure Using Connection Graphs.  
J. ACM 22:4 (1975).
- [9] H.J. Ohlbach  
The Markgraf Karl Refutation Procedure.  
The Logic Engine. Interner Bericht 24/82,  
University of Karlsruhe (1982).
- [10] H.J. Ohlbach  
The Markgraf Karl Refutation Procedure.  
The TERMINATOR Module. Interner Bericht,  
University of Karlsruhe (1983).
- [11] J. van Vaalen  
An Extension of Unification to Substitutions with an application to ATP.  
Proc. IJCAI-75, Tbilisi, USSR (1975).
- [12] S. Sickel  
A Search Technique for Clause Interconnectivity Graphs.  
IEEE Transactions on Computers C-25:8 (1976).