Centrum voor Wiskunde en Informatica
Centre for Mathematics and Computer Science

A. Eliëns

Parallel inference based on delta processing

Computer Science/Department of Software Technology Report CS-R8846 November

Bibliotheek
Centrum voor Wiskunde en Informatica
Amsterdam

The Centre for Mathematics and Computer Science is a research institute of the Stichting Mathematisch Centrum, which was founded on February 11, 1946, as a nonprofit institution aiming at the promotion of mathematics, computer science, and their applications. It is sponsored by the Dutch Government through the Netherlands Organization for the Advancement of Pure Research (Z.W.O.).

69C24, 69K11

Copyright © Stichting Mathematisch Centrum, Amsterdam

Parallel Inference based on Delta Processing

A. Eliëns

Centre for Mathematics and Computer Science
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

Expert system reasoning can be considered a special form of resolution. The disadvantage of general resolution, involving full forward and backward subsumption is that it is rather costly. A technique is presented that effectively reduces the computational cost of resolution-based inference in expert system reasoning. The idea is to encode the effects of a resolution step as instructions to modify a set of clauses. The technique, moreover, allows efficient parallelism, both by simultaneously processing generated information (AND-parallelism), and by exploring different search paths when splitting on a positive disjunction (OR-parallelism).

1980 Mathematics Subject Classification: 68G15.
1982 CR Categories: C.2.4, I.2.1.
Key Words & Phrases: expert systems, resolution, parallelism.
Note: The work in this document was conducted as part of the PRISMA project, a joint effort with Philips Research Eindhoven, partially supported by the Dutch "Stimulerings-projectteam Informatica-onderzoek" (SPIN).

Report CS-R8846
Centre for Mathematics and Computer Science
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

1. Introduction

From various perspectives it seems worthwhile to apply resolution based theorem proving techniques to expert system reasoning. In [1] it is noted that the operational characterization of the semantics of the knowledge representation formalism of MYCIN-like expert systems tends to obscure the meaning of a knowledge base. In [5] it is argumented that the limited reasoning capabilities of traditional expert systems like MYCIN, DENDRAL and PROSPECTOR will not suffice for intrinsically difficult problems, that require as they call it 'unfocussed search'. Both complaints can be accomodated by embedding an expert system in a general purpose resolution based theorem prover, such as LMA/ITP [5]. The disadvantage of general resolution, involving full forward and backward subsumption, however, is that it is rather costly.

Based on an analysis of the specific requirements imposed by expert system reasoning on a resolution based theorem prover, this paper presents an implementation technique that effectively reduces the cost of resolution based inference for this specific instance of automated reasoning. The idea is to encode the effects of a resolution step as instructions to modify a set of clauses. The technique, moreover, allows efficient parallelism on a multi-node distributed memory machine, both by simultaneously processing generated information (AND-parallelism), and by exploring different search paths when splitting on a positive disjunction (OR-parallelism).

2. Expert system reasoning as resolution

Expert system reasoning distinguishes itself from other applications of theorem proving by a clear distinction between data and rules [6]. It has been observed in [1] and [5] that, in order to apply resolution based inference, a model construction technique might be used. The generated model consists of the collection of facts that are positively true, such that all the rules in the knowledge base are satisfied, that is that for no rule both the conditions are fulfilled and the conclusions are false with respect to the interpretation represented by the collection of facts. To the extent that resolution based inference techniques apply, the algorithms for model generation presented here allow an efficient parallel implementation of expert system reasoning, assuming that the majority of rules is ground.

The conversion of production rules in if-then format to clauses is rather straightforward. As an example, the rule *if* $p(X)$ *and* $q(f(X))$ *then* $r(X,f(X))$ reads in clause form $-p(X) \mid -q(f(X)) \mid r(X,f(X))$. The conditions of a production rule correspond with the negative literals in the clause, and the conclusion corresponds with the positive literal. When multiple conclusions occur in a rule these have to be distributed over several clauses with an identical negative part. Production rules generally translate to Horn clauses, containing only one positive literal. Negative conditions in production rules, however, must translate to positive literals. [1] Clauses offer greater expressiveness than production rules since the positive part may contain more than one (positive) literal. However, positive (parts of) clauses represent a choice between alternative solutions which may be expensive to compute. Unit-clauses, that contain only one literal, play an important role in expert system reasoning since they represent the data. A set of clauses $C_1, \ldots, C_n$ will be written as $C_1 \ \& \ \cdots \ \& \ C_n$.

Resolution. Binary resolution is the rule that states that, for example, from clauses $-a \mid b$ and $-b \mid c$ one can derive the clause $-a \mid c$. The atom b that occurs positive in the first clause and negative in the second clause is called the *clashing* atom. The resulting clause is formed by joining the two clauses after deleting the literal that corresponds to the clashing atom. When variables are involved binary resolution requires that a substitution is found such that the atoms that potentially clash become identical. For instance, the clauses $a(1)$ and $-a(X) \mid b(X)$ result in the clause $b(1)$ with the unifying substitution that binds X to 1.
A disadvantage of binary resolution is that it generates all possible resolvents. Therefore, a number of inference rules have been proposed [5] that effectively reduce the number of generated clauses.

Each rule reflects a choice of strategy by imposing restrictions on what is allowed as input to the inference operation and what is allowed as output. In *unit resolution* resolution is restricted in such a way that at least one of the two input clauses is a unit clause. The objective of *unit-resulting resolution* is to produce a unit clause from a set of clauses one of which is a non-unit clause; the remaining must be unit clauses. For example, from the clause set $-p$ & q & $-q \mid p \mid r$ the unit r can be derived by applying unit-resulting resolution. The application of unit-resulting resolution can be viewed as a sequence of unit resolutions. Unit-resulting resolution is not refutation-complete for non-Horn clauses. The restrictions obeyed by *hyper-resolution*, that produces a positive clause from a set of clauses, one of which is negative or mixed while the remaining ones are positive, do not affect the completeness for arbitrary clauses.

Subsumption. Another way of limiting the amount of clauses generated is by applying a check for subsumption. Clause C_1 subsumes clause C_2 if there is a substitution θ such that $C_1\theta$ is contained in C_2. We speak of *forward subsumption* when a newly generated clause is discarded because a previously retained clause subsumes it; and of *backward subsumption* when a newly generated clause is used to discard previously retained clauses.

Set of support. To further restrict the number of clauses generated, as an additional strategy, a *set of support* may be maintained. The clauses are divided over two lists, an axiom list and a *set of support.* The *set of support* strategy [5] prohibits application of an inference rule to a set of clauses unless at least one clause has support. For a given rule of inference, a typical inference cycle, involving the *set of support*, can be characterized as follows.

while *set of support* $\neq \varnothing$ **do**
1. Select a clause from the *set of support.*
2. Compute all the resolvents of the selected clause and the other clauses.
3. Check for subsumption and add the remaining resolvents to the *set of support.*

od

The clause that is selected in step 1 is removed from the *set of support* and added to the axiom list. When an empty clause is generated in step 2 then an inconsistency must be reported. In step 3, when a resolvent clause is subsumed by an already retained clause it need not be added to the *set of support*; when a retained clause is subsumed by the generated clause it can be deleted.
The *set of support* strategy combined with binary resolution (and factoring) is complete, provided that the complement of the *set of support* is satisfiable. The use of the *set of support*, however, might inflict upon the completeness of a strategy, as for instance is the case with hyper-resolution when the *set of support* contains only negative literals.

For a typical expert system application the axiom-list (which in the sequel will be called the *theory*) is initialized to the set of rules, and the *set of support* to the data. Initially then, the *set of support* consists of unit clauses only. When unit-resultion resolution is applied, all generated clauses will be unit clauses. In this case the distinction between rules and data is preserved: resolving with the selected clause adds new data and testing for subsumption modifies the rules. Unfortunately, unit-resulting resolution is not complete for general (non-Horn) clauses. Other resolution strategies are, but these do not allow an interpretation of rules and data as operators on each other, unless the notion of data is extended to allow general clauses as input data, which clearly defeats the intent of the distinction.

Splitting. When positive clauses containing more than one literal occur, a model may be found by arbitrarily choosing an atom from the positive clause and trying to find a model for the augmented set of axioms, that now includes the chosen literal as a unit clause. For example, the clause set c & $-a \mid b$ & $-c \mid a \mid b$ has models $\{a,b,c\}$ and $\{b,c\}$, which may be found by respectively choosing for a and b when $a \mid b$ is generated. This example also shows that, dependent on the choice, it is not necessarily a minimal model that is found. This technique of model-generation is known as *splitting*: from a set of clauses T containing a positive clause in which an atom a occurs we can form the sets

$T_1 \equiv T \ \& \ a$ and $T_2 \equiv T \ \& \ -a$.
Splitting on clauses containing variables might be unsound, as illustrated by the example where $T \equiv a(X) | b(X) \ \& \ -a(1) \ \& \ -b(2)$. This set is satisfiable. Naive splitting however gives, after applying resolution and subsumption, $T_1 \equiv a(X) \ \& \ -a(1) \ \& \ -b(2)$ and $T_2 \equiv b(X) \ \& \ -a(X) \ \& \ -b(2)$ that are both refutable. It was observed in [4] that when the clauses satisfy the condition of range-restriction, which is the case if no variables occur in the positive literals of a clause that do not occur in a negative literal, then the resulting positive clauses will be variable-free.

Constraints. Clearly, splitting brings about an exponential growth in the cost of computation. In some cases, however, literals of a positive clause can be deleted by clashing them with negative unit clauses. Although negative units are not informative with respect to the model itself, except for detecting a contradiction, this use stresses the importance of negative (unit) clauses as *constraints* on generated clauses.

Expert system reasoning can be characterized by a clear distinction between data and rules. The *set of support* strategy, together with a suitable combination of hyper-resolution, unit-resulting resolution and case splitting allows to maintain this distinction during the inference, in the sense that either positive units are generated, or positive clauses that can be used for splitting, or negative units that may function as constraints. Other resolution strategies, like unit resolution or binary resolution, do not behave satisfactorily, in this respect. Why it is profitable to maintain this distinction can be clarified by the following observation concerning subsumption. When all the parents of a derived clause, except for one, are unit clauses and the non-unit parent clause is ground, then the derived clause subsumes the non-unit parent clause. For example the clause $c | d$ derived from $a \ \& \ b \ \& \ -a | -b | c | d$ subsumes the non-unit parent clause. As counter examples consider $a(1) \ \& \ -a(X) | -b(X)$ (containing variables) and $a | b \ \& \ -a | c$ (non-unit parents). The last example, by the way, can be handled by splitting on $a | b$. Assuming that the knowledge base is ground, this observation allows to apply unit resolution with the input data, and to effect subsumption simply by deleting the clashed literal from the rule.
The algorithms presented in the next sections will be based on the assumption that the majority of rules in the knowledge base is ground. How to deal with clauses containing variables will only be treated briefly.

3. The basic inference algorithm

As input to the procedure, we are given a theory T that is a set of clauses representing the rules and a set S of units that represent the data. The intent is to create a model M consisting of positive units. To allow checking for constraints, also a constraint set C is maintained, containing the negative unit clauses. An inference state, that is any point in the computation is fully characterized by the tuple $I = (T,S,M,C)$, consisting of a theory T, a *set of support* S, a model set M and a constraint set C.

Algorithm 1.
Input: T_0, S_0
Output: model M or inconsistent
Initialize M and C to empty, and invoke the procedure *infer*.
proc *infer* $(T,S,M,C) =$
repeat
  while $S \neq \emptyset$ **do**
    1. Take a unit l from S. If l is positive and not in M then add l to M, otherwise if l is negative and not in C then add l to C. Set S to $S-l$.
    2. If C and M contain complementary units then signal an inconsistency, else
    3. locate all clauses that contain a literal l' that clashes with l.
       Remove the clashed literals from the respective clauses, and if a unit remains add this to the *set of support* S.
  od
  If T does not contain any positive clauses then report that T_0 & S_0 is satisfiable and has model M, else select a positive clause from T, and invoke the procedure *infer* recursively both for $(T,\{l\},M,C)$ and $(T,\{-l\},M,C)$.
forever

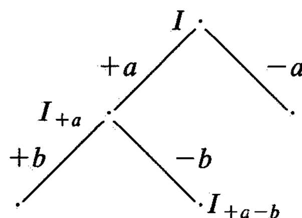
The procedure given above can be characterized as unit resolution. Since T is ground and one of the parents of each generated clause is a unit, subsumption can be handled in place, by simply deleting the clashed literal. As an additional test for subsumption it might be checked whether the selected unit corresponds with a literal of similar sign in a clause. In that case the entire clause can be deleted.

When there are non-ground clauses in T the clause resulting from the clash with the selected unit must be added to T, unless the resolvent is a unit clause. Assuming that only a small part of the rules will contain variables, a test for (either forward or backward) subsumption is not necessary, since subsumption merely serves to prune the search. As an optimization it might be considered to postpone resolving with the non-ground clause until either the *set of support* is exhausted or a sufficient number of units is available to apply hyper-resolution or unit-resulting resolution.

Implementation aspects. The *set of support* is a stream that presents units l of the form $+a$ or $-a$ to the theory T with the request to deliver back every unit that can be generated by processing this information. The theory-component must maintain an index to locate all occurrences of literals that clash with l, and further keep track of the contents of the clauses in order to check whether units can be send back to S. Processing the units delivered by T involves merely a check whether a unit of opposite sign is not contained in either the model set M or the constraint set C.

4. Delta processing

Case-splitting is intrinsically expensive, since it involves copying a full inference state, the theory T as well as the model set M and constraint set C. The resulting situation can be depicted by a tree of which the branches are labelled with the choices. For each node on the tree the inference state can be identified by the subscript formed from the labels on the path from the root to that node. For instance I_{+a-b} is the result of choosing respectively $+a$ and $-b$ as illustrated by



The amount of space needed to store the inference states grows exponentially. Even when use is made of backtracking the amount of space needed is proportional to the number of choices made. Moreover in the case of parallel execution the communication involved in creating new copies of inference states is prohibitive.

An alternative characterization of inference states allows to compute a model for a given theory and data more efficiently. Inference states will not be characterized as consisting of a theory T_i, a *set of support* S_i, a model M_i and a constraint set C_i, for some index i, but rather as a sequence of changes, named delta's, relative to an original inference-state $I = (T,S,M,C)$.

When discussing case splitting it was observed that the alternative inference states corresponding to the nodes of the tree could be uniquely identified by the sequence of labels occurring on the branches, representing the choice for a particular unit. This mode of indexing can be generalized by introducing nodes with one successor branch labelled by the unit selected from the *set of support*. In its most simple form, the sequence of delta's, that allows to (re)construct a particular inference state from a given inference state, is just of the form $<l_i>_i$, where each l_i is of the form $+a$ or $-a$ for an atom a. Clearly, when a choice is involved, the delta information is structured as a tree. The set of delta-sequences represented by the tree are all the subsequences of the sequences that are formed by going from the root to the leaves.

Merely recording the sequence of units processed by the *set of support* gives relatively little gain, except that storage of the modified theories for backtracking is no longer necessary. One can wonder if re-processing a sequence of changes isn't more expensive than just recording the state. A more refined notion of delta's, however, seems to induce a minimum of computational overhead in reconstructing an inference state from the delta-information, compared to copying full inference states, and moreover also seems to effectively reduce the communication-costs in the parallel processing of alternative states due to case splitting.

When the basic inference algorithm is more closely analysed it can be observed that the following kind of information is involved:

1. add a unit either to the model M or the constraint set C,
2. add a unit to T, to compute the resolvents,
3. delete a literal from a clause, due to a clash,
4. remove a subsumed clause from T,
5. add a unit to the *set of support*,

and in the presence of variables

6. add a clause to T as the result of a resolution step.

The case of adding a unit to the model M or the constraint set C involves a check for the consistency, which need only be performed once for each added unit. Adding a unit to T to compute the resolvents involves the major part of the work done during the inference process. For each given unit the clauses that contain a clashing literal must be located, the clauses must be accordingly simplified, a check must be made whether unit clauses should be added to the *set of support*, and whether the clashing-information must be updated. The result of this processing is a sequence of deletions of

literals from clauses and (possibly) a sequence of additions to the *set of support*. Processing a unit might also give rise to the removal of a clause due to subsumption. The delta's that have to be recorded to avoid recomputing the clash must contain the information that tells which occurrences of literals to delete, which clauses must be removed, and what has to be added to the *set of support*. All units that are added to the *set of support* will eventually also be put either on the model set or on the constraint set.

The following delta's are needed

a^+	add atom a to the model M
a^-	add atom a to the constraint set C
$\langle c,l \rangle$	delete literal l from clause c
$-c$	remove clause c from T

It is assumed that each clause has a unique clause number. The c in $\langle c,l \rangle$ and $-c$ refer to this number. When non-ground rules occur, as additional delta we need

(c,C)	add clause C with clause number c to T

Whenever a clause is added to T it must first be assigned a unique clause number, to allow the other delta's to refer to it.

As an example, let the theory T consist of clauses $C_1 \equiv -a \mid b$, $C_2 \equiv -b \mid c \mid d$ and $C_3 \equiv -c \mid e$, and let $S = \{a\}$. The sequence $<a^+,\langle 1,-a \rangle,b^+,\langle 2,-b \rangle>$ results initially, and will after splitting on $c \mid d$ (in the modified clause C_2) be continued with either $<c^+,\langle 3,-c \rangle,e^+>$ or $<c^-,\langle 2,+c \rangle,d^+>$.

As another example, involving variables, let T consist of $C_1 \equiv -a(X) \mid -b(X) \mid c(X)$ and $C_2 \equiv -a(X) \mid -c(X)$. With $S = \{a(1)\}$ this would result in $<a(1)^+,(3,C_3),c(1)^-,b(1)^->$, where $C_3 \equiv -b(1) \mid c(1)$. Note that C_3 is added because C_1 contains variables; C_2 does not give rise to an extra clause since clashing with $a(1)$ delivers the unit clause $-c(1)$, which is directly put on the *set of support*.

Using delta's allows to avoid copying full inference states since any state can be recomputed from a given state by processing the difference of the delta-sequence of the given state and the delta-sequence of the state that is to be reconstructed. Especially in the ground case, this can be done very efficiently. This setup also allows for efficient backtracking since, nice indeed, the delta's are reversible.

Treating equality. One extension that has to be dealt with, for the application in mind, is the possibility that a function term acquires a value during the inference, because of the occurrence of a positive equality unit. Operationally, a positive equality unit $eq(t_1,t_2)$ allows to replace a term t by t_2 provided that t is identical to t_1. If t_1 contains variables then a matching substitution θ must be found such that $t_1\theta = t$; in this case t is replaced by $t_2\theta$. To perform all simplifications due to an equality unit $eq(t_1,t_2)$ all terms must be located that potentially match with t_1. The position of a (sub)term in a term can be indicated by an integer sequence denoting the path to follow in a term. The empty sequence is the term itself and each subsequent number indicates the choice for a particular argument position. For instance, the term $t(X)$ is located on position $\langle 1,2 \rangle$ in $p(f(a,t(X)))$. The delta that results from a simplification must indicate, apart from the clause number and the literal number, the position in the atom of the literal and the value that is substituted. Also, in order to garantuee that delta's are reversible, the replaced term must be included. So, the delta that corresponds to replacing $t(X)$ by b in $p(f(a,t(X)))$ will be like $[c,l,\langle 1,2 \rangle,(t(X),b)]$, where c and l respectively are a clause and a literal number. When equality units are required to be of the form $eq(t,v)$, where v is some (irreducible) value, and moreover function terms are not allowed to occur nested, the term resulting from a simplification does not have to be further simplified, neither is it necessary to check whether due to this simplification other simplifications become possible.

5. Parallelism

The fact that delta's allow to recompute any inference-state from an initial state very efficiently is advantageous when employing a parallel search by case-splitting, since the communication overhead of copying the state in which the choice is made can be avoided.

5.1. Splitting

The parallel algorithm can be stated as follows. It is assumed that each process has the theory T built in when starting the computation. The model set M and constraint set C are local to a process and initialized to empty. The input to the procedure *infer* is a sequence of changes δ and a *set of support* S. Initially δ is empty and S contains the data provided by the user of the system. For convenience, delta notation $a^{\pm}$ and literal notation $\pm a$ will be used interchangably.

Algorithm 2:
proc *infer* (δ, S) =
Recompute the inference state by processing the sequence δ
repeat
 while $S \neq \emptyset$ **do**
 1. Take a unit from S, say a^*, and update the model or constraint set dependent on whether a^* is $+a$ or $-a$. If C and M contain complementary units then signal an inconsistency.
 2. Add a^* to δ, and send a^* to the theory component that computes the resolvents.
 3. Collect the sequence of changes δ' that results from processing a^*. Process δ', that is add all new items of the form a^* to the *set of support* and all other items to δ.

 od
 Request a choice atom a to use in case splitting,
 If no such choice atom can be provided, report that model M has been found, else put $+a$ on the *set of support* and invoke an inference process with a set of support $-a$ and delta-sequence δ.
forever

All intrinsically expensive computing is performed only once, in the resolution process, activated in step 2. The resulting state can efficiently be recomputed by a process waiting for the assignment of an inference-task, when it is invoked with a sequence of changes δ and a *set of support* S. An arbitrary number of processes can be initialized to the theory T and activated whenever necessary to perform the search. A possible optimization is to keep the waiting processes up to date by informing them about the changes.

A distinction has to be made between the top-level inference process and the processes at a lower level. Failure or inconsistency for low level processes merely indicates that the selected branch gives no result, whereas when the top-level process fails this directly indicates an inconsistency. When by splitting on an atom a precisely one branch succeeds, say $+a$, then $+a$ can be considered derivable; otherwise, in the case that both branches succeed neither $+a$ nor $-a$ is derivable, and in the case that both branches fail an inconsistency must be noted, with respect to the choices made thus far.

5.2. Clustering

The top level process as described in the previous section represents the linear part of the inference process. This part can take a proportional size. However, using delta's allows to simultaneously process items on the *set of support*. Linear parts of the inference can be computed by a cluster of processes according to the following algorithm.

Assume the cluster consists of resolution processes p_i, $i = 1,...,n$. Each such process is initialized to the theory T, and has the capability to perform resolution, that is to look for clashes and to effect simplifications. Initially, there is one master process with a cluster of slaves. A master-process can invoke another master process when splitting on a choice atom. Each master has a list of its resolution slaves as well as a free list of the slaves that are waiting for a task. A task is of the form a^* where $*$ is either $+$ or $-$.

```
Algorithm 3:
proc infer (δ,S) =
free: list of processes waiting for a task
slaves: list of all slave resolution processes
Initialize M and C by processing δ, and broadcast δ
to all slaves.
repeat
  while S ≠ ∅ do
    1. Take a unit a* from S, update M and C and check
       for consistency.
    2. Select a process p_i from the free list, and send it a*.
    3. Check for responses from slave processes, if slave p_i
       responds with delta sequence δ' broadcast δ'
       to all slaves except p_i, and process δ',
       that is add all new items of the form a* to the
       set of support and all other items to δ.
  od
  Request a choice atom a for case splitting.
  If no such choice atom can be found report model M,
  otherwise put +a on the set of support and invoke another
  master process with sequence δ and set of support −a.
forever
```

The clusters invoked can be of arbitrary size. Since delta processing can be applied the resolution-processes of these clusters can be initialized to T at startup time and remain waiting for a task. Also it would be possible for a cluster to grow, for instance when the *set of support* exceeds a certain size, by invoking new resolution processes. These processes would then have to adapt their state to the current delta sequence of the master process before accepting any task. Note that, when the deltas are reversible, as is the case for the delta's used in algorithm 3, the theory in processes that have failed can be re-used simply by applying the delta's in reversed order.

5.3. Implementation

Delta's can be implemented very efficiently, at least for the ground case. When the rules contain no variables, and when we also know what units will be given as initial data, then all the terms and atoms that may occur in the inference can be determined in advance. Moreover, since in this case, resolution with units from the *set of support* will not result in new clauses, the set of clauses will not change in size. This static behavior, due to a ground knowledge base, allows for an efficient scheme of indexing, along the lines indicated in [3]. This scheme can be generalized to deal with the dynamic behavior due to the addition of new clauses and terms in the presence of variables.

Indexing. The most straightforward implementation of a ground theory is as an array of clauses, where each clause is an array of literals. Each clause contains an indication of the number of negative literals and the number of positive literals. A delta for literal deletion may have the form $\langle c,l \rangle$ where c is the clause number and l the literal number.
To find efficiently all clashing literals for an incoming unit a clash table is needed, containing for each atom its occurences in the clauses, in the form of a clause number, a literal number and possibly whether it occurs positively or negatively. Performing the clash consists, for each clause, in which a complementary literal occurs, of marking the clashed literal as deleted, and decrementing the number

of positive or negative literals. When the number of literals decreases to one, a unit will be generated and the clause may be marked as deleted. If, after literal deletion, the clause contains only positive literals it is a candidate for splitting. All clauses in which the unit occurs as a literal can be deleted rightaway, due to subsumption. For performing the simplifications efficiently the clash table must also contain for each term the positions in which it occurs.

Variables. When the set of clauses grows dynamically the scheme presented above is too rigid, unless the number of added clauses is known in advance. Clauses can still be implemented as arrays since unit resolution does not increase the length of the clauses. Hyper-resoltution, however, may result in clauses that contain more literals than the parent clauses. To profit from the fact that in the majority of cases ground resolution suffices, the best solution seems to distinguish between a static clause list (containing the rules of the knowledge base) and a dynamic clause list (containing the clauses generated by a resolution step).
The process that generates a new (non unit) clause must request a (globally unique) clause number, insert the clause in the dynamic clause list and generate a delta containing the clause number and the clause. Since the size of the static part is known in advance, a process can decide by inspecting the clause number if a clause belongs to the dynamic part or the static part.
The presence of variables precludes a direct identification of clashing literals by means of a table of ground atoms. For non ground literals only a potential clash can be indicated. Whether the incoming unit clashes with a non-ground literal depends on whether a unifying substitution can be found. Similarly, for simplifications only possibly matching terms can be located.

Distribution. At the start of the computation an arbitrary number of slave resolution processes can be initialized with the theory T, that is the static part of the clause set, and a clash table containing information of the locations of terms and atoms in the clauses. The most general and space efficient solution for the clash table is to have for each predicate name and function name a list of occurrences of the form $\langle c,l,p \rangle$, where for atoms the position p is the empty sequence. Initially all resolution processes will have copies of T and the clash table, that will be updated by the delta's generated during the inference. The clash table must be updated only when new clauses are generated, which can be done by the resolution processes on the basis of the received delta's.

To reduce the cost of communication, a further simplification of the delta's might be achieved by maintaining a global table of all (ground) terms that occur, whereby we regard atoms as terms. Each of the master processes as well as the resolution slave processes must have a copy of this table at startup time and keep it updated by delta's of the form (t,T) which indicates that t is the number of T. This allows that both the model set and the constraint set consists of term numbers. Also the delta's corresponding to the addition of units to the *set of support* will be shorter; however if a new unit is generated, say $a(1)^+$ then two delta's may be needed, one to include $a(1)$ in the term table and one to add it to the *set of support*, as in $<(t,a(1)),t^+>$, where t is a new term number. The slave process that generates the new term must ask for a unique (term) number, similarly so when a new clause is generated.

To allow the resolution processes some liberty with respect to the inference strategy they employ, each such process must also have a copy of the model set M and constraint set C. It seems a reasonable strategy to postpone the resolution with a non ground clause until sufficient information is available. To extend the algorithm by allowing also non-unit clauses on the *set of support* seems unwise, even when units are given preference, in particular since the technique of splitting allows to generate a model even when non-units are disallowed to have support.

Complexity. The worst case complexity of the sequential procedure clearly is $O(2^n)$ in the ground case, where n is the number of literals. If linear processing suffices, that is when no splitting is necessary, the sequential procedure has polynomial (more precisely quadratical) complexity. The speedup due to parallel case splitting is linear, that is in the order of the number of processors. The speedup

by using a cluster of slave processes seems in the ground case to be also linear. However there is some overhead in processing the delta's. For the ground case the length of a delta sequence will not exceed the size of the knowledge base. With respect to expert system reasoning, it seems a reasonable assumption that in the average case only a limited number of units will be generated.

6. Discussion

The idea of delta processing was partly inspired by the approach to incremental text processing as allowed by SCCS under UNIX. Similar approaches are found in keeping logs of databases, and incremental parser generation.

The advantage of delta processing is that the information needed to characterize (and recompute) an inference state is kept to a minimum.

The approach to parallelism, applying delta processing, was inspired also by [2], in which a general approach to parallelism in deduction systems is described. The idea explored there is to use a collection of processes, attached to a master process that distributes tasks as determined by the clauses selected from the *set of support*, to perform the actual inference, which involves to determine the clashes, to test for subsumption and to simplify clauses on behalf of positive equality units. To maintain a theory over several processes in general, however, clauses that are added due to an inference step must be broadcast to each slave. This may involve expensive indexing and integration activity for each slave process that receives such a clause. Utilizing appropriate delta's might reduce the cost of adding clauses. The restrictions that apply to expert system reasoning however allow to utilize very simple delta's, most of the time. Since the updating information is of a relatively small size, no shared memory is needed to attain a reasonable performance, as was suggested in [2]. These restrictions, which consist of having a clean distinction between a (ground) theory and (ground) data, effect that the *set of support* contains unit clauses only and allow for the possibility to perform subsumption in place, by merely marking a literal as deleted. Communication overhead is avoided since processing delta's allows to reconstruct any inference state from one given initial inference state, so that an arbitrary number of processes can be allocated in advance and adapted to the current needs rather efficiently. The reversibility of delta's moreover allows to re-use processes by simply undoing the changes recorded in the delta sequence. Experiments are needed to show that the approach presented in this paper is fruitful in practice.

References

[1] M. Bezem *Consistency of rule-based expert systems* CADE-9 LNCS 310 Springer (1988) 151-162

[2] R.M. Butler and N.T. Karonis *Exploitation of parallelism in prototypical deduction problems* CADE-9 LNCS 310 Springer (1988) 333-343

[3] W.F. Dowling and J.H. Gallier *Linear time algorithms for testing the satisfiability of propositional Horn formulae* J. of Logic Programming 3 (1984) 267-284

[4] R. Manthey and F. Bry *SATCHMO: A theorem prover implemented in Prolog* CADE-9 LNCS 310 Springer (1988) 415-434

[5] L. Wos, R. Overbeek, E. Lusk and J. Boyle *Automated Reasoning: Introduction and applications* Prentice Hall 1984

[6] J. Treur *Completeness and definability in diagnostic expert systems* Proc. ECAI-88 München