Doctoral Thesis

A parallel resolution method for logic programs

Author(s):
Zolliker, Urs

Publication Date:
1993

Permanent Link:
https://doi.org/10.3929/ethz-a-000701573

Rights / License:
In Copyright - Non-Commercial Use Permitted
A Parallel Resolution Method for Logic Programs

A dissertation submitted to the
SWISS FEDERAL INSTITUTE OF TECHNOLOGY
ZURICH
for the degree of
Doctor of Mathematics

presented by
URS ZOLLIKER
Dipl. Math. ETH
born November 14, 1961
citizen of Embrach (ZH)

accepted on the recommendation of
Prof. Dr. E. Engeler, examiner
Prof. Dr. G. Jäger, co-examiner

1993
Contents

Abstract 3
Kurzfassung 5
Notation 7
Chapter I. Introduction 11
1. Resolution 11
2. Horn clauses 12
3. Model-Theoretic and Fixpoint Semantics 14
4. SLD-resolution 15
5. PROLOG 16
6. Breadth-first 18
7. Or-parallelism 18
8. And-parallelism 19
9. Stream-parallelism 19
10. Search-parallelism 20
11. Parallel unification 20
12. Distributed definite programs 20
13. Comparison with concurrent languages and Hyperresolution 21
14. Comparison with other proof methods 21
<table>
<thead>
<tr>
<th>Chapter II. Declarative semantics</th>
<th>27</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Distributed definite programs</td>
<td>27</td>
</tr>
<tr>
<td>2. Semantics</td>
<td>27</td>
</tr>
<tr>
<td>3. Approximation</td>
<td>29</td>
</tr>
<tr>
<td>4. Reduction of distributed definite programs</td>
<td>30</td>
</tr>
<tr>
<td>Chapter III. Procedural semantics</td>
<td>33</td>
</tr>
<tr>
<td>1. Equivalence classes</td>
<td>33</td>
</tr>
<tr>
<td>2. Derivation rules</td>
<td>33</td>
</tr>
<tr>
<td>3. Sequential derivation</td>
<td>34</td>
</tr>
<tr>
<td>3.1. Recursive definition</td>
<td>34</td>
</tr>
<tr>
<td>3.2. Properties of sequential derivations</td>
<td>35</td>
</tr>
<tr>
<td>3.3. Simple sequential algorithm</td>
<td>40</td>
</tr>
<tr>
<td>3.4. Extended sequential algorithm</td>
<td>41</td>
</tr>
<tr>
<td>4. Parallel derivation</td>
<td>42</td>
</tr>
<tr>
<td>4.1. Derivation rules</td>
<td>42</td>
</tr>
<tr>
<td>4.2. Simple parallel algorithm</td>
<td>45</td>
</tr>
<tr>
<td>4.3. Extended parallel algorithm</td>
<td>47</td>
</tr>
<tr>
<td>5. Extension</td>
<td>47</td>
</tr>
<tr>
<td>Chapter IV. Parallelization of definite programs</td>
<td>51</td>
</tr>
<tr>
<td>1. Definition</td>
<td>51</td>
</tr>
<tr>
<td>2. Example</td>
<td>52</td>
</tr>
<tr>
<td>3. Theorem</td>
<td>54</td>
</tr>
<tr>
<td>Bibliography</td>
<td>59</td>
</tr>
<tr>
<td>Curriculum vitae</td>
<td>61</td>
</tr>
</tbody>
</table>
Abstract

Parallelism in logic programming is often classified regarding different aspects such as Or-parallelism, And-parallelism, Stream-parallelism, and Search-parallelism. Most often, researchers consider SLD-refutations or a standard PROLOG interpreter and try to parallelize the algorithm or the work done by the interpreter.

Other researchers consider the elementary steps in refutations, namely unification, and work out parallel unification algorithms. However, instructions as to how these steps could be used to replace sequential unification in refutations are missing, since the result computed by parallel unification is not in appropriate form.

This work treats parallelism in a different way: We forgo SLD-refutations, seeing that as restriction of input refutations to definite programs they are suitable for sequential processing due to their linear form. Instead we restrict unit refutations to definite programs and investigate the parallelism based on the main intermediate clauses in unit refutations, namely the units. Unlike SLD-refutations, unit refutations are more naturally represented by trees than by sequences. In case the tree contains isomorphic subtrees the most appropriate representation of refutations is a directed acyclic graph. In our opinion, the space of such refutation graphs is well suited to parallel search.

We present a notion of distributed definite programs and a declarative semantics thereof. Apart from the distribution of program clauses to different nodes, semantics is not essentially extended by any new language constructs. This is illustrated by a simple reduction of distributed definite programs to conventional definite programs. Furthermore, we develop a resolution method for definite programs, as well as for distributed definite programs. The resolution method resembles a question-answer-game performed by one or several agents. Questions and answers are represented by unit clauses. We prove correctness and completeness of the resolution algorithm with respect to the declarative semantics. Dis-
tributed definite programs represent parallelizations of definite programs. In order to guarantee preservation of completeness, we specify conditions for transformations of definite programs into distributed ones.
Kurzfassung

In der Logikprogrammierung wird Parallelität oft klassifiziert nach ein paar wenigen Gesichtspunkten: Or-Parallelität, And-Parallelität, Stream-Parallelität und Search-Parallelität; Viele Forscher betrachten SLD-Widerlegungen oder Standard-PROLOG-Interpreter und versuchen den Algorithmus bzw. die Arbeitsweise des Interpreters zu parallelisieren.

Andere Forscher betrachten die zugrunde liegenden Schritte in den Widerlegungen, nämlich die Unifikation, und entwickeln parallele Unifikationsalgorithmen. Es fehlen jedoch Hinweise darauf, wie diese Schritte in die Widerlegungen eingebaut werden könnten, da die Resultate, die durch parallele Unifikation berechnet werden, nicht in geeigneter Form sind.


Wir erschaffen die Idee von verteilten definiten Programmen und deren deklarative Semantik. Ausser für die Verteilung der Programmklauuseln auf verteilte Knoten werden dabei keine neuen sprachlichen Ausdrucksweisen, die die Bedeutung grundsätzlich erweitern würden, eingeführt. Dies machen wir deutlich anhand einer einfachen Reduktion von verteilten definiten Programmen auf konventionelle definite Programme. Ferner
Notation

We use natural language to formulate lemmas, theorems, corollaries, and proofs. Wherever practical, however, we make statements and proof steps concise by using the formal symbols of classical logic:

- $\land$: conjunction;
- $\lor$: disjunction;
- $\rightarrow$: implication;
- $\exists$: existential quantifier;
- $\forall$: universal quantifier;
- $\iff$: equivalence;

We also use set theoretic operations and notations to express notions:

- $\{\ldots\}$: description of a set;
- $\in$: membership relation;
- $\{\}$: the empty set;
- $\subseteq$: set inclusion, subset relation;
- $\cup$: set union;
- $\bigcup$: set union over a set or sequence of sets;
- $\cap$: set intersection over a set of sets;
- $\setminus$: asymmetrical set difference;
- $\times$: Cartesian product of sets;
- $=$: equality;
- $2^\mathcal{H}, 2^\mathcal{F}, 2^\mathcal{D}$: powerset: the set of all subsets of $\mathcal{H}$ or $\mathcal{F}$ or $\mathcal{D}$, respectively.
- $\rightarrow$: function space: specification of domain and range of a function;
- $\mapsto$: function description that specifies the association of an argument and the corresponding function value;
- $\langle\ldots\rangle$: pair, triple, tuple, or sequence.

The lemmas, theorems, corollaries, and proofs of this work contain statements about concepts for the language of logic programming. We use the
following symbols for the syntax and for collections of syntactical objects of logic programming:

\[ J, K, L, N, U, W, X, Y, Z: \] variables occurring in program clauses;
\[ 0: \] constants such as, for instance, the zero element;
\[ e(.,.), s(.): \] functors such as, for instance, the binary embedding functor and the unary successor functor;
\[ P, even, gt, ge, minus, plus, times, binom, surj: \]

Often, we use strings of lower-case letters as predicate symbols;
\[ a, a', a_n, l, l_n: \] atomic formulas;
\[ b, b_k: \] atomic ground formulas;
\[ x, y: \] set of ground atomic formulas;
\[ F: \] set of all atomic formulas for a first-order language;
\[ \Lambda: \] (sequential) conjunction symbol within clauses, used as separator between negative literals;
\[ \leftarrow: \] implication symbol within clauses, used to separate the single positive literal from the body (i.e. sequence of negative literals); also used to designate goal clauses in case the left-hand side is empty;
\[ c, c_i: \] variables for clauses;
\[ d, d_j: \] variables for definite clauses;
\[ g: \] variable for goal clauses;
\[ S: \] variable for a set of clauses;
\[ D: \] definite programs (i.e. sets of clauses);
\[ \mathcal{D}: \] set of all (extended) definite programs;

In order to talk about the semantics of logic programs the following concepts and symbols are used:
\[ H: \] Herbrand universe (of a first-order language);
\[ t: \] element of a Herbrand universe;
\[ \hat{H}: \] Herbrand base (for a first-order language);
\[ I: \] Herbrand interpretation;
\[ \theta, \sigma, \rho, \gamma: \] substitutions;
\[ c\sigma: \] instance of clause \( c \) due to substitution \( \sigma \);
\[ =: \] syntactical identity;
\[ M_D: \] least Herbrand model of a set \( D \) of clauses;
\[ T_D: \] a certain mapping from Herbrand interpretations of \( D \) to Herbrand interpretations of \( D \).
\[ T_D^{(m)}: \] composition of \( T_D \) iterated \( m \) times;

In order to talk about the positions of elements within sequences and for the presentation of an intuitive example we use the following arithmetic notions:
\[ N: \] set of all natural numbers;
\[ 0, 1, \ldots: \] well-known constants in \( N \);
\( i, j, k, n \): variables for members of \( \mathbb{N} \);
\(+, -, \cdot\): binary functions over \( \mathbb{N} \) : addition, subtraction, multiplication; also used for infix notation of binary functors in the language of logic programming;
\( >, <, \leq \): ordering relations over \( \mathbb{N} \); also used for infix notation of binary predicates in the language of logic programming;

In this work some abbreviations and definitions are introduced. We use the following symbols:

- \( V \): set of processes represented by vertices in a graph;
- \( p_0, p_1, p_2 \): process constants: members of \( V \);
- \( u, v, w \): process variables ranging over \( V \);
- \( E \): set of channels represented by labeled edges in a graph;
- \( \delta \): mapping \( V \to 2^\mathbb{N} \) determining distribution of clauses in a distributed definite program;
- \( <V, \delta, E> \): distributed definite program;
- \( x \): function \( x : v \to \mathbb{H} \), \( v \mapsto x_v \);
- \( \pi_c \): mapping \( 2^\mathbb{H} \to 2^\mathbb{H} \) specifying selection function for relevant data regarding a channel;
- \( M_{\pi_c}^{\varepsilon} \): least local model for process \( w \) in the distributed definite program \( <V, \delta, E> \);
- \( \varepsilon(.,.), \Delta(.) \): certain reduction mappings;
- \( a, a', a_k, q, q' \): atomic formulas, answers, queries.
- \( \bar{a}, \bar{q} \): equivalence classes of atomic formulas;
- \( A, A_j, A_j, A_j, A', A'', \Delta A \):
  sets of atomic formulas representing answers;
- \( Q, Q_j, Q_j, Q_j, Q', Q', \Delta Q \):
  sets of atomic formulas representing queries;
- \( B \): set of atomic ground formulas;
- \( B^f \): set of atomic ground formulas;
- \( B \): approximation function;
- \( <D, Q, A> ? c \): relation representing the query derivation rule;
- \( <D, Q, A> ! c \): relation representing the answer derivation rule;
- \( F \): set of all equivalence classes of atomic formulas;
- \( f \): function representing the main procedure for query and answer derivation in the resolution algorithms;
- \( || \): parallel conjunction symbol used within clauses; this symbol is added to the language of logic programming;
- \( \circ_n \): place-holder for sequential and parallel conjunction;
CHAPTER I

Introduction

1. Resolution

The resolution principle as a basis for automated theorem-proving was introduced by J.A. Robinson [Rob65]. In comparison with the trial and error attempts on ground\(^1\) formulas (e.g. Herbrand procedure, procedure of Davis and Putnam [DP60]), resolution was soon considered to be more systematic and much more promising. However, when performing general resolution, a fast growth of clauses in size and number still takes place. Researchers then first focused their attention on strategies restricting the form of resolution steps and supplying preference rules in order to get feasible proof systems.

One famous and successful theorem-prover of this kind was presented and implemented in 1964 (cf. [WCR64]). It was called TPU\(^2\), since it tries to produce as short resolvents as possible and thus prefers unit resolution\(^3\) steps and obeys a function depth test. Like general resolution, TPU works on general problems, i.e. the domain of universal formulas in prenex conjunctive normal form. TPU is a complete proof system, since whenever unit resolution does not produce new resolvents or when the function depth is increased by one, some general resolution steps are also performed in a fair way.

Furthermore some researchers investigated unit refutations: A refutation of \(S\) is a clause sequence \(c_1, c_2, \ldots, c_n\) such that \(c_n\) is the empty clause \(\square\) and each \(c_i\) is either a member of \(S\) or is obtained by binary resolution or factoring from earlier clauses. A unit refutation is a refutation \(c_1, c_2, \ldots, c_n\) such that, if \(c_i\) is obtained by binary resolution, then it is obtained by unit resolution. Characteristics of unit resolution: The

\(^1\) A formula is called ground if it contains no variables.
\(^2\) Theorem Proving with Unit preference strategy.
\(^3\) A unit resolution is a resolution in which at least one of the two parent clauses is a unit clause.
elements of a refutation of \( S \) are instances of parts of members of \( S \). Thus the number of atoms in the longest clause of \( S \) is an upper bound for the number of atoms in each clause of a unit refutation.

Another attempt at restricting the number of immediately applicable resolution steps, was input resolution\(^4\): An **input resolution** is a resolution in which at least one of the two parent clauses is an input clause, i.e. a clause of the original set \( S \) of axioms. An **input refutation** is a refutation \( c_1, c_2, \ldots, c_n \) such that, if \( c_i \) is obtained by binary resolution, then it is obtained by input resolution. Characteristics of input resolution: Input resolution is essentially linear\(^5\) since if we start with the empty clause at the end of the refutation and follow the parents we can select a linear subsequence of the refutation. The remaining clauses are irrelevant to the proof.

Generally, neither unit resolution nor input resolution leads to a complete proof system. In 1970 however Chang [Cha70] showed that both systems are of equal expressive power: A set of clauses has a unit refutation if and only if it has an input refutation.

In 1971 Kowalski and Kuehner [KK71] introduced SL-resolution, i.e. linear resolution with selection function. The main purpose of a selection function is to avoid redundant derivations by choosing from each clause one single literal to be resolved upon. Selection functions determine the order of resolution steps. As far as selection functions are concerned, resolution steps commute in a similar way as in confluent replacement systems\(^6\). Thus completeness of fair resolution proof systems is not affected by any selection function. Working with sets of general clauses, linear refutations usually consist of ancestor resolution and factoring steps as well as input resolution steps. Without loss of generality, it is possible to impose on linear resolution the restriction that no resolvent is a tautology and that the top clause \( c_1 \) belongs to a given support set\(^7\) of \( S \). In addition there are some more restrictions defined for SL-resolution. The argument for increased efficiency of SL-resolution is based on considerations in [KK71] showing that in comparison with unrestricted resolution, SL-resolution does not significantly increase the number of resolution and factoring steps of simplest proofs.

### 2. Horn clauses

In the previous section we looked at how restriction of admissible resolution steps influences efficiency and completeness. In this section we

---

\(^4\)Sometimes called axiom resolution.

\(^5\)A precise definition for linear resolution is given on page 24.

\(^6\)See for instance terminology in introduction and referenced articles of [Boo82].

\(^7\)A subset \( T \) of \( S \) is a support set of \( S \) if \( S \setminus T \) is satisfiable.
consider a restriction of admissible axioms in a logic program and we outline how this affects resolution and the nature of problems that might be expressed this way.

A clause is a **Horn clause** if it contains at most one positive literal. The interest in Horn clauses goes back to Horn's theorem of 1951 about direct unions of algebras (see [Hor51] page 17):

Let $T$ be a closed sentence\(^8\) which has a prenex conjunctive normal form in whose matrix each factor contains at most one non-negative equation\(^9\). Then if $T$ is true of each factor algebra $\Gamma_i$ ($i \in I$), then it is true of the direct union $\Gamma$ of the algebras $\Gamma_i$.

Reversely Horn showed that besides tautologies and formulas as specified in the above theorem there can be no further criteria of a propositional kind that allow concluding the validity of formulas in the direct union from their validity in the factor algebras. Instead of reproducing the accurate form of the reverse theorem we quote a more concise theorem from Shoenfield's collection of principal results of mathematical logic. He outlines the relationship between Horn theories and model theory as follows (see [Sho67] page 95):

A theory $T$ is equivalent to a theory whose nonlogical axioms are Horn clauses if and only if every substructure of a model of $T$ is a model of $T$ and every direct product of models of $T$ is a model of $T$.

In 1972 Kuehner characterized unit resolution by proving that every minimally unsatisfiable set $S$ of ground clauses has a unit refutation if and only if there is a renaming $r$ such that $r(S)$ is a set of Horn clauses (see [Kue72]). He defined a renaming to be a function $r$ such that $r(S)$ is identical to $S$ except that for each atom $b$ of a given set each occurrence of $\neg b$ in $S$ is replaced by $b$ and each unnegated occurrence of $b$ in $S$ is replaced by $\neg b$.

In 1974 Henschen and Wos [HW74] characterized Horn clauses independently by proving that every minimally unsatisfiable set $S$ of ground clauses is a set of Horn clauses if and only if there is a **positive-unit refutation**\(^10\) of $S$. In one direction this result is also valid on the general level: If $S$ is an unsatisfiable set of Horn clauses, then there exists a **strictly-positive-unit refutation**\(^11\) of $S$ and a **strictly-input refutation**.
tion\textsuperscript{12} of $S$.

Roughly speaking we learn that the class of Horn clauses forms exactly the domain of clauses over which unit resolution, or respectively input resolution, are complete; Furthermore over the domain of Horn clauses factoring is redundant or, in other words, resolution as defined by Robinson [Rob65] can be restricted to binary resolution without losing completeness.

3. Model-Theoretic and Fixpoint Semantics

First, we will quote some definitions from [vEK76], in abbreviated form: The set of all terms containing no variables which can be constructed from the constants and other function symbols occurring in a set of clauses $D$ is called the **Herbrand universe** $H$ determined by $D$. The set of all ground atomic formulas $P(t_1,\ldots,t_n)$, where $P$ occurs in the set of clauses $D$ and $t_1,\ldots,t_n$ belong to the Herbrand universe $H$ of $D$, is called the **Herbrand base** $H$ of $D$. A **Herbrand interpretation** $I$ of $D$ is any subset of the Herbrand base of $D$. If $D$ is true in $I$, then we say that $I$ is a **Herbrand model** of $D$. (A ground atomic formula $b$ is **true** in $I$ iff $b \in I$.) Let $\mathcal{M}(D)$ be the set of all Herbrand models of $D$.

The model-theoretic semantics determines the denotation of a predicate symbol $P$ in a set of clauses $D$ as the set of tuples $<t_1,\ldots,t_n>$ over the Herbrand universe, such that $D$ logically implies $P(t_1,\ldots,t_n)$.

In 1974 Van Emden and Kowalski [vEK76] showed that a tuple $<t_1,\ldots,t_n>$ belongs to the denotation of $P$ if and only if

$$P(t_1,\ldots,t_n) \in \bigcap \mathcal{M}(D).$$

The fixpoint semantics for predicate logic programs has been defined in [vEK76] and is restricted to definite programs. A Horn clause containing exactly one positive literal is called a **definite program clause** and a set of definite program clauses is called a **definite program**. For every definite program $D$ there is an associated transformation $T_D$ mapping Herbrand interpretations to Herbrand interpretations:

$$T_D(I) \text{ contains a ground atomic formula } b \in H \text{ iff for some ground instance } d\sigma \text{ of a clause } d \text{ in } D, d\sigma = b \lor \neg b_1 \lor \ldots \lor \neg b_m$$

and $b_1,\ldots,b_m \in I$, $m \geq 0$.

The fixpoint semantics determines the denotation of a predicate symbol $P$ in a set of clauses $D$ as the the set of tuples $<t_1,\ldots,t_n>$ such that $P(t_1,\ldots,t_n)$ belongs to $\bigcap \{I : T_D(I) \subseteq I\}$.

\textsuperscript{12}A strictly-input refutation is an input refutation without factoring.
Van Emden and Kowalski proved that for definite programs the equality
\[ M(D) = \{ I : T_D(I) \subseteq I \} \]
holds. Furthermore the equation
\[ \bigcup_{m \in \mathbb{N}} T_D^{(m)}(\{\}) = \bigcap\{ I : T_D(I) \subseteq I \} \]
is usually proved in fixpoint theory by demonstrating the continuity of the transformation \( T_D \). Thus for definite programs model-theoretic semantics, fixpoint semantics and operational semantics
\[ \bigcup_{m \in \mathbb{N}} T_D^{(m)}(\{\}) \]
coincide.

Fixpoints are often infinite, even for very simple programs. This is especially the case, when terms inherit properties of subterms. For example the definite program clauses
\[
\begin{align*}
\text{even}(0) & \leftarrow. \\
\text{even}(s(s(X))) & \leftarrow \text{even}(X).
\end{align*}
\]
define even numbers. The fixpoint is an infinite set of terms. However for testing, if a given number is even, we certainly do not need to compute the whole fixpoint.

4. SLD-resolution

Combining linear resolution with selection function (section 1) with input resolution on Horn clauses (section 2) results in a complete proof procedure for definite programs (section 3). This proof procedure was first described in 1974 by Kowalski ([Kow74] page 570) and was later called SLD-resolution [AvE82]. We quote the definition of Kowalski:

The standard notion of computation, applied to Horn clause programs, concerns the repeated use of procedure invocation in order to derive new goal statements from old ones with the ultimate objective of deriving the halt statement\(^{13}\). More precisely, given a set \( D \) of Horn clauses and an initial goal statement \( d_1 \in D \), a computation is a sequence of goal statements \( d_1, \ldots, d_n \) such that \( d_{i+1} \) is derived by procedure invocation from \( d_i \) using a procedure\(^{14}\) in \( D \) whose name\(^{15}\) matches some selected procedure call\(^{16}\) in \( d_i \). A computation is successful if it ends with the halt

---

\(^{13}\)Empty clause.

\(^{14}\)Definite program clause.

\(^{15}\)Head of the clause.

\(^{16}\)Subgoal.
statement, i.e. if $d_n = \Box$. A computation terminates without success if the selected procedure call in the end goal statement $d_n$ matches the name of no procedure in $D$.

This notion of computation is non-deterministic, since there are often several definite program clauses whose heads match the selected procedure call. For each step of the computation one of the potential definite program clauses is chosen non-deterministically. The role of non-determinism and the influence of the selection function might be illustrated by the notion of SLD-tree, defined in [AvE82]. SLD-trees are of bounded and small degree\(^{17}\), due to input and linear resolution restriction and selection function. An upper bound for the number of resolution steps of simplest proofs can be transferred from SL-resolution to SLD-resolution if the clause set under consideration is already in Horn form. Due to these arguments SLD-resolution seems to be very efficient. Since the definition of SLD-resolution, however, does not contain any specifications, how non-determinism is to be dealt with, efficiency considerations are more appropriate in connection with deterministic proof procedures or, strictly speaking, in connection with proof procedures where non-determinism does not influence the result, e.g. whether or not a computation is successful (cf. above quotation). Nevertheless we argue that SLD-resolution contains an inherent source for inefficiency because the number of literals per clause in a derivation might increase without bound.

5. PROLOG

In 1973 the interpreter PROLOG was developed by the artificial intelligence group at the University d'Aix-Marseille. An outline of the implementation and a few references to internal research reports can be found in [Col78]. PROLOG tries to search all SLD-derivations for a given initial goal clause applying backtracking, i.e. traversing an SLD-tree with a depth-first search rule. The selection function of PROLOG interpreters usually is the leftmost computation rule.

The depth-first search rule on SLD-trees causes serious termination troubles. In other words: Depth-first search strategies are unfair. Brough and Walker [BW84] illustrated this problem in 1984 for different kinds of interpreters with loop traps. They considered a sub-language of definite programs, namely finite programs with ground assertions only and with clauses not containing function terms and such that variables occurring in the head occur in the body as well. Moreover they introduced the term preorder interpreter: an interpreter applying the depth-first search rule and possibly investigating the execution stack to detect and prune infinite branches of the SLD-tree. Brough and Walker proved that every

\(^{17}\)Maximum number of sub-trees of a node.
preorder interpreter terminating on all questions and simple programs of
the given kind produces some incomplete answer sets. However, they also
proved the existence of an interpreter (without preorder) that halts on
every simple program and every question and always produces complete
answer sets.

Many current PROLOG implementations are based on compilation to
an abstract PROLOG instruction set [War83] and an engine, WAM, (i.e.
Warren Abstract Machine) emulating this target language.

When traversing a SLD-tree two kinds of operations are needed:

- Unification: Subgoals of clause bodies are unified with clause
  heads. In WAM a considerable part of unification work is cov¬
ered by compiling clauses and optimizing the passing of values
between subsequent procedure calls, taking into account the argu¬
ment types (variables within different types of scopes, constants,
structures, etc.).
- Search: Considering the types of arguments in subgoals (variables,
ground terms, structures containing uninstantiated variables, etc.)
all potentially matching program clauses are selected. In WAM
this is solved by indexing, such that fast access to the correspond¬
ing clauses (and saving of space) is supplied in case of an instan¬
tiated first argument, while sequential search through a part of
the program (all clauses of one predicate) is used in case of an
uninstantiated first argument.

On the one hand, the leftmost computation rule provides great flexi¬
bility for PROLOG programmers in that the sequential order of subgoals
in the body of a clause allows to work with terms as special as possible,
preferably ground. By reordering subgoals within clauses programmers
can drastically trim the degree of corresponding SLD-trees and often force
finiteness of the SLD-tree for queries of a certain form. On the other hand,
such programming is clearly of procedural nature and thus attraction of
declarative programming is lost. Taking program clauses as declarative
specifications allows verification of the soundness of any computed re¬
sults (unless occur-check conflicts arise), but in no way helps to answer
the question of whether PROLOG terminates on queries of a certain form.

PROLOG computations require a great amount of computer mem¬
ory: Intermediate results computed during the traversal of one SLD-tree
might be computed several times. A SLD-derivation might include a
single program clause several times. A single subgoal might appear in
one SLD-derivation several times and thus the computation for one SLD-
derivation might contain the computation for a single subgoal several
times. Stack space usage is (more or less) proportional to the length of
a SLD-derivation and for problems of deterministic nature (e.g. induc-
tively defined functions) the length of a SLD-derivation is proportional to the time used for the traversal of a SLD-tree. In order to prevent such enormous memory requirements programmers insert cuts into program clauses, expressing that at distinguished points during execution a considerable part of the stack should be cut away.

Typical applications of PROLOG are implementations of recursive definitions of operations in many sorted algebras (see [CPP78]).

6. Breadth-first

According to [Lov78] a search is called breadth-first if and only if for every \( n \geq 0 \) there are times \( t_1 \) and \( t_2 \), such that most of the clauses of level \( n \) that are generated during the search are generated between times \( t_1 \) and \( t_2 \), and the time interval \( (t_1, t_2) \) is relatively small compared to the total time of search.

One version of breadth-first search applied on general resolution in order to prove the inconsistency of a set \( \mathcal{S} \) of clauses is computing stepwise the sets \( \mathcal{R}^n(\mathcal{S}) \) and testing membership of the empty clause. \( \mathcal{R} \) is defined as follows:

\[
\begin{align*}
\mathcal{R}^0(\mathcal{S}) & = \mathcal{S}, \\
\mathcal{R}^{n+1}(\mathcal{S}) & = \mathcal{R}^n(\mathcal{S}) \cup R(F(\mathcal{R}^n(\mathcal{S}))), \\
R(\mathcal{S}) & = \{ \text{"binary resolvents of members of } \mathcal{S}\} , \\
F(\mathcal{S}) & = \mathcal{S} \cup \{ \text{"factors of members of } \mathcal{S}\} . 
\end{align*}
\]

Regarding SLD-resolution, one version of breadth-first search is developing an SLD-tree for \( D \cup \{ \leftarrow g \} \) with the leftmost computation rule computing in the \( n^{th} \) step all clauses for nodes with distance \( n \) from the root and checking whether the empty clause is assigned to any node at level \( n \). Any other computation rule can be used likewise for breadth-first search.

Verification of inconsistency does not depend on the computation rule: For each answer \( \rho \) computed by one computation rule, breadth-first search combined with any other computation rule produces a computed answer \( \sigma \) such that \( \leftarrow g\rho \) is a variant of \( \leftarrow g\sigma \). Compare independence of the computation rule [Llo87] p. 49.

However, finiteness of the SLD-tree, and thus verification of consistency, depends in general on the computation rule.

7. Or-parallelism

Or-parallelism is the potential parallelism in the breadth-first growth of the SLD-tree (see [Rob92]). In Or-parallel PROLOG multiple processors explore different branches of the SLD-tree. Designers of Or-parallel PROLOG implementations want to keep the cost of all operations very
close to what they would be in sequential standard PROLOG implementations (cf. [War87]). Of particular concern are the costs of creating and accessing variable bindings. In comparison with backtracking through alternative tasks in standard PROLOG implementations, it is also vital to keep down the cost of creating multiple tasks at Or-parallel branch points. Therefore in Or-parallel implementations each processor explores as large subtrees as possible and only rarely switches tasks and jumps to a different part of the tree. For the most part each processor acts just like a sequential PROLOG processor in a depth-first, left-to-right manner with backtracking.

8. And-parallelism

And-parallelism means simultaneously starting processes to solve each subgoal of a conjunction, i.e. each subgoal in a clause body. Conery and Kibler [CK81] consider it reasonable to solve subgoals in parallel in cases where subgoals are not interrelated by shared variables. However, they recommend solving interrelated subgoals in sequence, solving first the subgoal from which a smaller number of solutions might be expected.

9. Stream-parallelism

Stream-parallel languages are often called concurrent logic programming languages. A short overview is given in [TF86]. Stream-parallelism can be exploited in SLD-refutations when two subgoals share a variable. One kind of Stream-parallelism is cooperative parallelism: For instance in a “divide and conquer” algorithm two non-overlapping sublists are constructed by two parallel processes and the parent process continues work on the concatenated list. Another kind of Stream-parallelism is pipelining parallelism: For instance one process constructs a list incrementally and another process starts execution on the available part of the list before the former process has completed construction. In papers about Concurrent Prolog and PARLOG this kind of parallelism is called And-parallelism.

Languages based on Stream-parallelism are Concurrent Prolog, PARLOG and Guarded Horn Clauses. These three logic programming languages share a common feature: guarded clauses of the form

\[ a_0 \leftarrow a_1, \ldots, a_n | a_{n+1}, \ldots, a_m \]

with a commitment operator "|". The role of the commitment operator is the selection of one subtree of the corresponding node in the SLD-tree when searching for SLD-refutations. A clause is selected as soon as its guard part \( a_1, \ldots, a_n \) succeeds, and processes for the other clauses of the predicate are aborted.
10. Search-parallelism

By Search-parallelism Conery and Kibler [CK81] refer to the possibility of partitioning the data base into disjoint sets of clauses, permitting parallel search on data bases containing a large number of assertions, i.e. unit program clauses.

11. Parallel unification

A data-parallel SIMD implementation of parallel unification is described in [Bar90]. In order to exploit inherent parallelism, expressions are thought of as being directed labeled graphs. Hyperresolution [Rob92] is used to combine several resolution steps into one inference step such that the costs of building such a graph and extracting a most general unifier from the resulting graph on termination of the unification process are justified. However, combining several steps prevents pruning of the search space by detecting failure or redundancy at an early step.

12. Distributed definite programs

In this work we introduce the notion of distributed definite programs. The idea was born while implementing a prototype with the intention of creating a tool that helps to experiment and detect new possibilities for parallel reasoning on logic programs. However, the implementation took longer than we had expected and the requirements grew and grew. For this reason we decided first to explore the theoretical background for such an implementation by proving theorems for the required crucial concepts. The goal of this work is the presentation of these theorems.

In chapter II we give a set theoretic definition for distributed definite programs. We translate the concepts of the Herbrand model and operational fixpoint semantics (cf. section 3) to corresponding concepts on distributed definite programs. Furthermore, we give a simple and direct reduction of distributed definite programs to definite programs in order to illustrate that semantics of programs is not essentially complicated by introducing new language constructs.

In chapter III we introduce a procedural semantics for definite programs as well as for distributed definite programs. Relationships between procedural and declarative semantics are described by proving correctness and completeness theorems. The lifting theorem shows a relationship between derivations for general queries and ground queries. We present different versions of algorithms obeying procedural semantics. Table I.1 in section 14 shows that the main difference between breadth-first search for SLD-refutations and procedural semantics in chapter III is that the former is based on linear resolution while the latter is based on unit resolution.
In chapter IV parallelizations, i.e. translations of definite programs to distributed definite programs, are considered. Sufficient conditions are given for preservation of completeness. Parallelizations can be effectively verified, such that it is guaranteed that the corresponding distributed program computes the same results as the original program. An example is given.

13. Comparison with concurrent languages and Hyperresolution

In implementations of And-parallelism and Or-parallelism the scopes of variables and environments usually range over almost a whole branch of an SLD-tree, i.e. over almost a whole SLD-refutation. In implementations of algorithms of chapter III, however, scopes of variables and environments range over a query or answer derivation step only, i.e. over the part of a path in a unit resolution graph between two nodes labeled with unit clauses.

In Stream-parallel implementations concurrent work is done on data structures, i.e. parts of formulas. In algorithms of chapter III, however, concurrency focuses on different parts of a unit resolution graph. In this work we do not explore concurrency on the lower level of data structures.

Search-parallelism has been defined with regard to large data bases of unit program clauses. Conery and Kibler [CK81] suggested applying Search-parallelism in implementations of Or-parallelism. The concept of distributed definite programs extends Search-parallelism to general definite program clauses and to intermediate results within the resolution algorithm.

At first sight, it seems that Hyperresolution could easily be applied to answer derivation steps in chapter III. Hyperresolution is less appropriate for query derivation, since only parts of program clauses are involved. Closer inspection, however, shows that query derivation for different initial segments of the clause body and answer derivation can be combined and unification work can be shared. Thus, Hyperresolution is not appropriate for resolution algorithms in chapter III. Nor can parallel unification without Hyperresolution be expected to be appropriate, since for each subgoal it would be necessary to transform the computed unifier into an appropriate form for further processing.

14. Comparison with other proof methods

Table I.1 gives a classification of the following algorithms and implementations:

Davis-Putnam: The proof procedure described in [DP60] is a uniform procedure which will ultimately prove any first-order for-
mula which is valid. The first step of the refutation algorithm leads to a universally quantified prenex formula with a matrix in conjunctive normal form. The resulting formula is inconsistent if and only if the original formula is valid. The second step produces conjunctions of ground instances of a formula in conjunctive normal form in an exhaustive way. Each produced conjunction is then tested for consistency. Davis and Putnam focused their attention on the efficiency of consistency tests for ground formulas in conjunctive normal form. Thus, in this comparative overview we consider the last part of the proof procedure only. In literature the Davis-Putnam procedure is also often presented as a satisfiability test for propositional formulas only.

**semantic tree:** Given a conjunction of general clauses. Consider a binary decision tree. Choose incrementally for each level a ground literal. Label each left edge with the affirmative literal of the level and each right edge with its negation. A node is a failure node, if and only if a ground instance of an original clause exists, such that for each literal in the clause the negation coincides with a label somewhere on the branch from the root to the node in question. Prune the tree at each failure node and make sure that each non-failure node has a left and a right outgoing edge with labels according to the level. If a finite tree can be created

### TABLE I.1. overview

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Size</th>
<th>Linear Search</th>
<th>Clauses</th>
<th>Query</th>
</tr>
</thead>
<tbody>
<tr>
<td>Davis-Putnam</td>
<td>NP</td>
<td>n.a.</td>
<td>finite</td>
<td>GCNF</td>
</tr>
<tr>
<td>sem. tree</td>
<td>g. atoms</td>
<td>n.a.</td>
<td>guess</td>
<td>arbitrary</td>
</tr>
<tr>
<td>fixpoint</td>
<td>g. atoms</td>
<td>no</td>
<td>exhaustive</td>
<td>SKB</td>
</tr>
<tr>
<td>ground</td>
<td>NP</td>
<td>no</td>
<td>finite</td>
<td>ground</td>
</tr>
<tr>
<td>resolution</td>
<td>arbitrary</td>
<td>no</td>
<td>guess</td>
<td>arbitrary</td>
</tr>
<tr>
<td>TPU</td>
<td>subclauses</td>
<td>no</td>
<td>parametrized</td>
<td>arbitrary</td>
</tr>
<tr>
<td>input</td>
<td>arbitrary</td>
<td>yes</td>
<td>support</td>
<td>arbitrary</td>
</tr>
<tr>
<td>SLD</td>
<td>stack</td>
<td>yes</td>
<td>guess</td>
<td>definite</td>
</tr>
<tr>
<td>SLDNF</td>
<td>stack</td>
<td>no</td>
<td>guess</td>
<td>normal</td>
</tr>
<tr>
<td>breadth-first</td>
<td>arbitrary</td>
<td>no</td>
<td>exhaustive</td>
<td>arbitrary</td>
</tr>
<tr>
<td>breadth-first</td>
<td>arbitrary</td>
<td>yes</td>
<td>exhaustive</td>
<td>definite</td>
</tr>
<tr>
<td>PROLOG</td>
<td>stack</td>
<td>yes</td>
<td>backtrack</td>
<td>definite</td>
</tr>
<tr>
<td>Or-Par.</td>
<td>stack</td>
<td>yes</td>
<td>parallel</td>
<td>definite</td>
</tr>
<tr>
<td>PARLOG</td>
<td>stack</td>
<td>yes</td>
<td>parallel</td>
<td>definite</td>
</tr>
<tr>
<td>GHC</td>
<td>stack</td>
<td>yes</td>
<td>parallel</td>
<td>definite</td>
</tr>
<tr>
<td>CP</td>
<td>stack</td>
<td>yes</td>
<td>parallel</td>
<td>definite</td>
</tr>
<tr>
<td>Unit-par.</td>
<td>atoms</td>
<td>no</td>
<td>exhaustive</td>
<td>definite</td>
</tr>
</tbody>
</table>
obeying these rules, the original set of clauses is unsatisfiable. Conversely, if ground literals are chosen due to an enumeration of the whole proper Herbrand base, then every unsatisfiable set of clauses yields such a finite tree (cf. [Lov78]).

**fixpoint:** The operational semantics as defined in [vEK76] does not lead directly to an algorithm: In general $T_D(I)$ contains an infinite number of atomic ground formulas since for most program clauses there are infinite numbers of instances. However, if the language is restricted as in [BW84] each step produces a finite set only and in this case operational semantics is an effective semi-decision algorithm: Each correct answer is computed after a finite number of steps.

**ground:** Resolution on ground formulas, see ground resolution in [Lov78].

**resolution:** Resolution on general formulas, strictly speaking on first-order formulas without quantifiers, representing the corresponding universally quantified Skolem prenex formulas; see resolution in [Rob65, Lov78].

**TPU:** Theorem Proving with Unit preference strategy, compare section 1 and [WCR64] and [Cha70].

**input:** Input resolution, compare [Cha70] and [Lee67].

**SLD:** SLD-resolution.

**SLDNF:** SLDNF-resolution, compare [Llo87] p. 84.

**breadth-first:** See section 6; one entry for general resolution and one for SLD-resolution.

**PROLOG:** Compare section 5.

**Or-Par.:** Or-parallelism, compare section 7 and [War87].

**PARLOG:** Compare section 9.

**GHC:** Guarded Horn Clauses, compare section 9.

**CP:** Concurrent Prolog, compare section 9.

**Unit-par.:** Unit-Parallelism: The algorithm described in chapter III. In the “size” column we specify complexity if the domain is restricted to ground formulas:

**NP:** The corresponding satisfiability problem belongs to NP. For the variant of the Davis-Putnam procedure given in [Lov78] this follows from the fact that with each rule application, each resulting subproblem is described by fewer symbols. If the empty conjunction is derived after guessing the satisfiable subproblem for each application of the splitting rule, then satisfiability is verified after a polynomially bounded number of steps.
Known algorithms, however, perform in exponential time. For instance, it has been shown (cf. [CR74, Tse66]) that every implementation based exclusively on the rules\textsuperscript{18} of the Davis-Putnam procedure needs an exponential number of rule applications in the worst case. In the original variant (see [DP60]) such effort can be observed during each application of the elimination rule which expands a disjunction of two conjunctive formulas.

For general clauses, however, we do not have a reasonable measure of complexity at our disposition. In this case the "size" column shows the number of atoms intermediate clauses of a proof might contain. For proof trees the size of edge labels or node labels are considered. In our view, large intermediate results increase the difficulties in parallelizing proof procedures. Intermediate clauses contain:

- **g. atoms**: One ground atomic formula.
- **atoms**: One atomic formula.
- **subclauses**: A bounded number of atoms, since each intermediate clause is an instance of a subclause of a program clause; representable by a pointer into the program area and an environment.
- **arbitrary**: An unlimited finite number of atoms.
- **stack**: An unlimited finite number of atoms; intermediate results are represented by the content of the execution stack. Computation steps modify the stack, mostly by pushing values onto the top or removing values from the top.

The "linear" column specifies whether or not the proof procedure is based on linear resolution. A refutation $c_1, c_2, \ldots, c_n$ is called linear if and only if for each $i$ $(1 < i < n)$ one of the following alternatives holds:

- $c_{i+1}$ is a binary resolvent of $c_i$ with an input clause, factor of an input clause, or $c_j$ $(j < i)$.
- $c_{i+1}$ is a factor if $c_i$.

If, however, resolution is already supposed to cover factoring, a linear refutation is a refutation $c_1, c_2, \ldots, c_n$ such that for each $i$ $(1 < i < n)$, $c_i$ is a parent clause of $c_{i+1}$ regarding resolution. Explanation of table entries:

- **n.a.**: Proof procedure is not based on resolution.
- **no**: Refutations are not linear.
- **yes**: Refutations are linear.

The "search" column specifies the search rule:

- **finite**: The search space is finite. Every heuristics is effective.

\textsuperscript{18}One-literal, affirmative-negative, elimination: cf. [DP60]; splitting, subsumption: cf. [Lov78].
**guess:** A refutation is found by guessing appropriate non-deterministic choices. For definite programs each correct answer is verifiable this way. For hierarchical normal programs each correct answer with respect to completion is verifiable in a similar way if the union of program and goal are allowed (for definitions see [Llo87]). In the case of semantic trees, a small finite tree can be found for each unsatisfiable formula by guessing appropriate ground literals.

**exhaustive:** The search space is recursively enumerable. The search rule is given by a semi-algorithm, possibly computing an infinite number of steps. Each correct answer is covered by a corresponding result produced after a finite number of steps. Sometimes exhaustive search is well suited for parallel search.

**parametrized:** The same as "exhaustive"; in addition the end user can influence the order of produced results by supplying priority rules.

**support:** Set of support, compare page 12.

**backtrack:** A depth-first search rule is applied performing backtracking on a von Neumann style computer, compare section 5.

**parallel:** There are parallel processes involved in the computation, compare sections 7 and 9.

The "clauses" column describes the program domain. A program as a set of clauses is a conjunction of clauses. A program represents:

**arbitrary:** A first-order formula of predicate calculus, in Skolem prenex conjunctive form.

**ground:** A ground formula in conjunctive normal form.

**definite:** A conjunction of definite program clauses (i.e. clauses with exactly one positive literal).

**normal:** A conjunction of normal program clauses (i.e. clauses with one designated positive literal as head and a sequence of arbitrary literals as body).

**GCNF:** A ground formula in conjunctive normal form.

**SKB:** A simple knowledge base as defined in [BW84]: a conjunction of definite clauses such that no clauses contain function terms, all atomic clauses are ground, and in each composite clause variables occurring in the head also occur in the body.

The "query" column specifies whether or not the algorithm starts with a query and explores only the part of the search space demanded by the query.
Leer - Vide - Empty
CHAPTER II

Declarative semantics

1. Distributed definite programs

We introduce a notion of distributed definite programs. In essence, such programs consist of a directed graph and two functions: One function associates a set of definite program clauses to each node of the graph. The other function associates an atomic formula to each edge of the graph. For simplicity the latter function is expressed by a third component of the edges. Instead of pairs we write triples, inserting the atomic formula as middle component:

Definition 1. Let \( \mathcal{D} \) be the set of all definite programs and \( F \) the set of all atomic formulas. A **distributed definite program** \( <V, \delta, E> \) consists of

- a set \( V \) of process identifiers (vertices);
- a function \( \delta : V \rightarrow \mathcal{D} \) assigning programs to process identifiers;
- a set of channels (edges) \( E \subseteq V \times F \times V \);

The nodes are called processes and the edges are called channels. The intention is that at every node one processor executes the associated program, sending atomic formulas through the adjacent channels to other processes. The labels of the channels are criteria for selection of data and for determining which processes data should be sent to.

The graph in figure II.1 is the graph underlying the example in chapter IV:

2. Semantics

In analogy to the fixpoint semantics for ordinary definite programs we define a declarative semantics for distributed definite programs. We express the meaning of the local program of a process by a closure condition with respect to a mapping defined for the program. The meaning of a channel is expressed by a closure condition with respect to a mapping
given by the channel label. We use the terms process property for the former condition and channel property for the latter.

For each process $v$ we consider sets $x_v$ of atomic ground formulas consistent with the axioms represented by the program clauses of $\delta(v)$, i.e. Herbrand models of $\delta(v)$. For this purpose we recall the definition of the function $T_{\delta(v)}$ described in section 3 of the introduction:

**Definition 2.**

$$
T_D : \mathcal{2}\hat{H} \rightarrow \mathcal{2}\hat{H}
$$

$$
y \mapsto \{ b_0 \in \hat{H} \mid (\exists c \in D)(\exists \sigma)(\exists b_1, \ldots, b_n \in y) \ \forall \sigma = (b_0 \leftarrow b_1 \land \ldots \land b_n) \}
$$

In order to be consistent with $\delta(v)$, the sets $x_v$ must be closed against the mapping $T_{\delta(v)}$.

With each atomic formula $a$ we associate a projection function in the power set of the Herbrand base:

**Definition 3.**

$$
\pi_a : \mathcal{2}\hat{H} \rightarrow \mathcal{2}\hat{H}
$$

$$
y \mapsto \{ b \in \hat{y} \mid \exists \sigma (b = a \sigma) \}
$$

and for each channel $<v,a,w>$ we demand that $\pi_a(x_v)$ is included in $x_w$.

**Definition 4.** A mapping $x : V \rightarrow \mathcal{2}\hat{H}, v \mapsto x_v$ is a distributed model of $<V,\delta,E>$ if the following two conditions are satisfied:

- process property: $(\forall v \in V) T_{\delta(v)}(x_v) \subseteq x_v$
- channel property: $(\forall v \in V \forall a \in F \forall w \in V) <v,a,w> \in E \rightarrow \pi_a(x_v) \subseteq x_w$

Let $w \in V$. We say a set $y \subseteq \hat{H}$ is a local model for $w$ in $<V,\delta,E>$ if and only if there is a distributed model $x : V \rightarrow \mathcal{2}\hat{H}, v \mapsto x_v$ with $y = x_w$.

Let $M_{<V,\delta,E>}$ denote the least local model for $w$ in $<V,\delta,E>$ with respect to set inclusion. We call a distributed model $x$ the least distributed model for $<V,\delta,E>$ if $x_v$ is the least local model for each process $v \in V$. 
3. Approximation

In analogy to the coincidence of the operational semantics for definite programs (\(\bigcup_{i \in \mathbb{N}} T^{(i)}_D(\{\})\)) and the least Herbrand model, we define an approximation for distributed definite programs as a proof tool for verifying procedural semantics, algorithms and implementations:

**Definition 5.** Let \(<V, \delta, E>\) be a distributed definite program and \(B\) the function

\[\mathbb{N} \times V \rightarrow 2^H, \langle j, v \rangle \mapsto B^v_j\]

recursively defined over \(\mathbb{N}\):

\[B^v_0 := \{\} \text{ and } B^v_{j+1} := T_{\delta(v)}(B^v_j) \cup \bigcup_{\langle w, a, v \rangle \in E} \pi_a(B^w_j)\]

We call this function the **approximation** corresponding to the distributed definite program \(<V, \delta, E>\).

**Theorem 1.** (approximation theorem) Let \(B\) be the approximation corresponding to the distributed definite program \(<V, \delta, E>\). Then the mapping

\[x : V \rightarrow 2^H, v \mapsto x_v := \bigcup_{j \in \mathbb{N}} B^v_j\]

is a distributed model of \(<V, \delta, E>\). Furthermore, this mapping \(x\) is the least distributed model for \(<V, \delta, E>\).

**Proof:**

**Process property:** Let \(b_0 \in T_{\delta(v)}(x_v)\). There is a clause \(d \in \delta(v)\) and a substitution \(\sigma\) and some \(b_1, \ldots, b_n \in x_v\) such that

\[d \sigma = (b_0 \leftarrow b_1 \land \ldots \land b_n)\]

Since \(B^v_0 \subseteq B^v_1 \subseteq \ldots\), there is \(i \in \mathbb{N}\) such that \(b_1, \ldots, b_n \in B^v_i\). \(b_0 \in T_{\delta(v)}(B^v_i)\). By definition of \(B\), \(b_0 \in B^v_{i+1}\). Thus \(b_0 \in x_v\).

**Channel property:** Let \(<v, a, w> \in E\) and \(b \in \pi_a(x_v)\). \(b\) is an instance of \(a\) and \(b \in x_v\). There is \(i \in \mathbb{N}\) such that \(b \in B^v_i\). \(b \in \pi_a(B^v_i)\). By definition of \(B\), \(b \in B^w_{i+1}\). Thus \(b \in x_w\).

**Least model:** Let \(y\) be a distributed model for \(<V, \delta, E>\). By induction over \(j \in \mathbb{N}\), \((\forall v \in V)B^v_j \subseteq y_v\). Thus \(x_v \subseteq y_v\). \(x_v\) is the least local model for each \(v \in V\). \(x\) is the least distributed model.
4. Reduction of distributed definite programs

In this section we show that the introduction of distributed definite programs does not extend the expressible power of the language of definite programs, because distributed definite programs can be reduced to definite programs in a simple way:

\[
\begin{array}{ccc}
\text{distributed} & \xrightarrow{\text{definite}} & \text{definite program} \\
\downarrow & & \downarrow \\
\text{least distributed model} & \xrightarrow{\text{least}} & \text{Herbrand model}
\end{array}
\]

The least local model for a process can be obtained, roughly speaking, by the following steps:

- Take the union of the definite programs for all processes;
- Determine the least Herbrand model of this union;
- Extract from this model all atomic formulas with predicate names occurring in the local process.

However, we might get an incorrect result if channels hide a predicate symbol occurring in the programs of two processes. In order to avoid such naming conflicts, we rename each predicate by putting all atomic sub-formulas into an envelope \(e(v,.)\) and supplying additional rules for the channel conditions:

\[
e(v, D) := \{ e(v, a_0) \leftarrow e(v, a_1) \land \ldots \land e(v, a_k) \mid (a_0 \leftarrow a_1 \land \ldots \land a_k) \in D \}
\]

\[
\Delta(<V, \delta, E>) := \bigcup_{v \in V} e(v, \delta(v)) \cup \bigcup_{(v, a, w) \in E} \{ e(w, a) \leftarrow e(v, a) \}.
\]

**Theorem 2. (Reduction theorem.)** Let \(<V, \delta, E>\) be a distributed definite program. For each process \(v \in V\) and all ground formulas \(b\)

\[
b \in M_{\Delta(<V, \delta, E>)} \iff e(v, b) \in M_{\Delta(<V, \delta, E>)}.
\]

**Proof:**

\(\Rightarrow\): We define a mapping \(x_v := \{ b \in \hat{H} \mid e(v, b) \in M_{\Delta(<V, \delta, E>)} \} \).

Let \(\theta\) be a substitution and \((a_0 \leftarrow a_1 \land \ldots \land a_k) \in \delta(v)\). Suppose \(a_1, \ldots, a_k \in x_v\). By definition of \(x\):

\[
e(v, a_0) \leftarrow e(v, a_1) \land \ldots \land e(v, a_k)
\]

we get \(e(v, a_0)\theta \in M_{\Delta(<V, \delta, E>)}\) and \(a_0 \theta \in x_v\). Thus \(x_v\) is a Herbrand model of \(a_0 \leftarrow a_1 \land \ldots \land a_k\) and \(x\) has the process property.
Let \( <v, a, w> \in E \). Suppose \( b \) is a ground instance of \( a \) and \( b \in x_v \).
Thus \( e(v, b) \in M_{\Delta(<V, \delta, E>)} \). Since \( M_{\Delta(<V, \delta, E>)} \) is a Herbrand model of
\[
e(w, a) \leftarrow e(v, a)
\]
we get \( e(w, b) \in M_{\Delta(<V, \delta, E>)} \) and \( b \in x_w \). Thus \( x \) has the channel property.

Let \( b \in M'_{<V, \delta, E>} \). Since \( x \) is a distributed model of \( <V, \delta, E> \) we get \( b \in x_v \) and \( e(v, b) \in M_{\Delta(<V, \delta, E>)} \).

\( \Leftarrow \): There is a distributed model \( x \) such that \( x_w = M^w_{<V, \delta, E>} \). Let
\[
y := \bigcup_{v \in V} \{ e(v, b) \mid b \in x_v \}.
\]
There are two classes of clauses \( d \in \Delta(<V, \delta, E>) \):

- If \( d \) is of the form
  \[
e(v, a_0) \leftarrow e(v, a_1) \land \ldots \land e(v, a_k)
  \]
  then
  \[
  (a_0 \leftarrow a_1 \land \ldots \land a_k) \in \delta(v).
  \]
  Suppose \( e(v, a_1)\theta, \ldots, e(v, a_k)\theta \in y \). This implies \( a_1\theta, \ldots, a_k\theta \in x_v \). Since \( x \) has process property we get \( a_0\theta \in x_v \) and \( e(v, a_0)\theta \in y \).

- If \( d \) is of the form
  \[
e(w, a) \leftarrow e(v, a)
  \]
  with \( w \neq v \) then \( <v, a, w> \in E \). Suppose \( e(v, a)\theta \in y \). This implies \( a\theta \in x_v \). Since \( x \) has channel property we get \( a\theta \in x_w \). Thus \( e(w, a)\theta \in y \).

Let \( e(v, b) \in M_{\Delta(<V, \delta, E>)}. \) Since \( y \) is a Herbrand model of each \( d \in <V, \delta, E> \) we get \( e(v, b) \in y \) and \( b \in x_v \). Thus \( b \in M'_{<V, \delta, E>} \). \( \square \)
CHAPTER III

Procedural semantics

1. Equivalence classes

In the following sections we treat procedural semantics, working mainly with atomic formulas. This is completely different from SLD-resolution, where derivations consist of sequences of clauses. Wherever we write any formula, we actually mean its universal closure. For our purpose two atomic formulas are equivalent if and only if their universal closures are logically equivalent. In other words, the atomic formulas $a$ and $a'$ are equivalent if and only if there is a renaming substitution\(^1\) $\sigma$ such that $a\sigma = a'$. Thus let us then define:

**Definition 6.**

$$F \rightarrow 2^F$$

$$a \mapsto \overline{a} := \{ a' \in F \mid \exists \theta \exists \sigma (a'\sigma = a \land a' = a\theta) \}.$$  

Furthermore, we write $\overline{F}$ for the set $\{ \overline{a} \mid a \in F \}$.

2. Derivation rules

In this section we introduce two basic derivation rules concerning the process property (see section 2 in chapter II).

The answer derivation rule takes some classes $\overline{a_1}, \ldots, \overline{a_n}$ and produces a new class containing ground formulas of the image of the mapping $T_{\delta(v)}$ applied on the set of ground instances of $a_1, \ldots, a_n$. The universal closure of formulas derived in this way are logical consequences of the local program $\delta(v)$ and formulas approved by other processes. Care is taken that the answer derivation rule does not produce formulas more general than required for investigation of a given query. This helps to keep the search space as small as possible.

The query derivation rule takes an atomic goal and produces a new goal regarding some formulas derived by the answer rule. The new goal is

\(^1\)For terminology see [Lio87] page 22.
intended to be joined to the set of queries or to be sent to another process according to the channel labels. The sets of queries derived in this way represent the search space in which the answer derivation rule should be applied.

**Definition 7.** Let \( D \) be a definite program, \( a, q \) atomic formulas, \( Q, A \) sets of equivalence classes.

**Query derivation:** We write \(<D, Q, A> ? q\) if and only if there is a clause \( l_0 \leftarrow l_1 \land \ldots \land l_n \) in program \( D \), \( k \in \mathbb{N} \) with \( k < n \), some atomic formulas \( q', a_1, \ldots, a_k \), a substitution \( \sigma \) such that

\[
\overline{q} \in Q \land \overline{a_1}, \ldots, \overline{a_k} \in A \land \overline{q} = l_{k+1} \sigma,
\]

and \( \sigma \) is the most general unifier with

\[
(l_0 \leftarrow l_1 \land \ldots \land l_k) \sigma = (q' \leftarrow a_1 \land \ldots \land a_k) \sigma.
\]

**Answer derivation:** We write \(<D, Q, A> ! a\) if and only if there is a clause \( l_0 \leftarrow l_1 \land \ldots \land l_n \) in program \( D \), some atomic formulas \( q, a_1, \ldots, a_n \) a substitution \( \sigma \) such that

\[
\overline{q} \in Q \land \overline{a_1}, \ldots, \overline{a_n} \in A \land \overline{a} = l_0 \sigma,
\]

and \( \sigma \) is the most general unifier with

\[
(l_0 \leftarrow l_1 \land \ldots \land l_n) \sigma = (q \leftarrow a_1 \land \ldots \land a_n) \sigma.
\]

3. Sequential derivation

**3.1. Recursive definition.** In this subsection a sequential procedural semantics for queries on definite programs is defined. In essence, it consists of incrementally computing the closure with respect to query and answer derivation:

**Definition 8.** Let \( q \) be an atomic formula. We say

\(<<Q_0, A_0>, <Q_1, A_1>, \ldots>\)

is a **sequential unit derivation** over program \( D \) for goal \( q \) if and only if

\[
\begin{align*}
Q_0 & := \{ \overline{q} \}; \\
A_0 & := \{ \} ; \\
Q_{i+1} & := Q_i \cup \{ \overline{p} | <D, Q_i, A_i> ? a \} ; \\
A_{i+1} & := A_i \cup \{ \overline{a} | <D, Q_i, A_i> ! a \} .
\end{align*}
\]

The members of \( Q_0, Q_1, \ldots \) are called **queries** and the members of \( A_0, A_1, \ldots \) **answers.**
3.2. Properties of sequential derivations. In this subsection theorems for correctness, completeness, and lifting property are proved. First, we present the correctness theorem:

**Theorem 3. (correctness of sequential derivation)**

Let \(<<Q_0,A_0>,<Q_1,A_1>,...>\) be a sequential unit derivation over the definite program \(D\). Ground instances of sequentially derived answers belong to the least Herbrand model i.e. for each atomic formula \(a\) and for each ground substitution \(\theta\) for the variables in \(a\):

\[
(\forall i \in \mathbb{N})(\bar{a} \in A_i \rightarrow a\theta \in M_D).
\]

**Proof:** Induction on index \(i\) over \(\mathbb{N}\):

- **Induction hypothesis for \(i\):** For each atomic formula \(a\) and for each ground substitution \(\rho\) for variables in \(a\):
  
  \[
  \bar{a} \in A_i \rightarrow a\rho \in M_D.
  \]

  **Initiation:** \(A_0\) is empty.

  **Inheritance:** Assume that \(\bar{a} \in A_{i+1}\) and let \(\theta\) be a ground substitution for the variables in \(a\); then either \(\bar{a} \in A_i\) or there is a clause \((l_0 \leftarrow l_1 \land \ldots \land l_n) \in D\) and a substitution \(\sigma\) such that \(l_0\sigma = a\) and \(l_1\sigma, \ldots, l_n\sigma \in A_i\). By induction hypothesis \(l_1\sigma, \ldots, l_n\sigma \in M_D\).

  Thus \(a\theta \in M_D\), since \(T_D(M_D) \subseteq M_D\).

Second, we present a lemma showing that all formulas in the intersection of the least Herbrand model and the constructed search space are instances of derived answers:

**Lemma 1. (Ground lemma)** Let \(<<Q_0,A_0>,<Q_1,A_1>,...>\) be a sequential unit derivation over the definite program \(D\) and \(M\) the least fixpoint

\[
\bigcup_{m \in \mathbb{N}} T_D^{(m)}(\{\})\text{.}
\]

If a member of \(M\) is an instance of a query, then it is also an instance of an answer:

\[
(\forall b \in \hat{H})(\forall q \in F)(\forall i \in \mathbb{N})(\forall \theta)(\exists a \in F)(\exists j \in \mathbb{N})(\exists \sigma)
\]

\[
b \in M \land q \in Q_i \land b = q\theta \rightarrow \bar{a} \in A_j \land b = a\sigma.
\]

**Proof:** We say there is a corresponding query \(q\) for \(b\) if there are \(i \in \mathbb{N}\) and \(\bar{q} \in Q_i\) such that \(b\) is an instance of \(q\). We say there is a corresponding answer \(a\) for \(b\) if there are \(j \in \mathbb{N}\) and \(\bar{a} \in A_j\) such that \(b\) is an instance of \(a\).

**Induction on fixpoint iteration:**

- **Hypothesis for \(k\):** For each element of \(T_D^{(k)}(\{\})\) there is a corresponding answer if there is a corresponding query.
initiation: $T_D^{(0)}(\{\})$ is empty.

inheritance: Assume that $b_0 \in T_D^{(k+1)}(\{\})$ and there is a corresponding query $q$ for $b_0$.

There is a clause $d \in D$ and an instance

$$b_0 \leftarrow b_1 \land \ldots \land b_n$$

of $d$ such that

$$b_1, \ldots, b_n \in T_D^{(k)}(\{\}).$$

By induction on the position $h$ we get for each subgoal of the clause:

- a corresponding query $q_h$ for $b_h$ by applying the query derivation rule on clause $d$, query $q$ and answers $a_1, \ldots, a_{h-1}$;
- a corresponding answer $a_h$ for $b_h$ by induction hypothesis for $k$, since we know that there is a corresponding query $q_h$.

We get a corresponding answer for $b_0$ by applying the answer derivation rule on clause $d$, query $q$ and answers $a_1, \ldots, a_n$. Thus the induction hypothesis for $k + 1$ is valid.

In order to treat completeness we need to say which atomic formulas are correct results. The term Herbrand model is not sufficient for this purpose, since it covers ground formulas only. Hence we use the following definitions, similar to [Llo87]:

**Definition 9.** Given a clause $c$, the **universal closure** of $c$ is the closed formula obtained by adding a universal quantifier for every variable having an occurrence in $c$. The symbols $\leftarrow$ and $\land$ of our notation for clauses correspond to the logical implication $\rightarrow$ and the logical conjunction $\land$, i.e. a clause $l_0 \leftarrow l_1 \land \ldots \land l_n$ corresponds to the logical formula $l_0 \lor \neg l_1 \lor \ldots \lor \neg l_n$.

Usually, the terms model and logical consequence are used only for closed formulas and sets thereof. Working with definite programs we omit quantifiers and thus the following definitions are appropriate for us:

**Definition 10.** Let $a$ be an atomic formula. We say $I$ is a model of $a$ if and only if it is a model of the universal closure of $a$. Let $D$ be a definite program. We say $I$ is a model of $D$ if and only if $I$ is a model for the universal closures of all clauses of $D$.

**Definition 11.** Given an atomic formula $a$ and a program $D$, we say $a$ is a logical consequence of $D$ if and only if for each interpretation $I$ of an appropriate language for $D \cup \left\{ \leftarrow a \right\}$, $I$ is a model of $D$ implies that $I$ is a model of $a$. 
We adapt the definition for correct answers as well. For reasons of simplicity we consider unit goal clauses only.

**Definition 12.** Let \( D \) be a definite program, \( q \) an atomic formula and \( \theta \) a substitution for variables in \( q \). We say \( \theta \) is a **correct answer** for \( D \cup \{ \leftarrow q \} \) if and only if \( q\theta \) is a logical consequence of \( D \).

Obviously, for unit goal clauses a substitution \( \theta \) is a correct answer in this sense if and only if it is a correct answer in the sense of [Llo87].

With above definitions we can restate the correctness theorem as follows:

**Corollary 1.** Each sequentially derived answer is a logical consequence of the definite program.

**Proof:** Let \( a \) be a sequentially derived answer over the definite program \( D \). By correctness theorem each ground instance of \( a \) belongs to the least Herbrand model \( M_D \). Thus \( M_D \) is a model of the universal closure of \( a \). Hence every model of \( D \) is a model of \( a \) (cf. model-theoretic semantics in section 3 of the introduction).

We now come to the third result in this subsection:

**Theorem 4.** (completeness of sequential derivation)
Let \( \langle Q_0, A_0 \rangle, \langle Q_1, A_1 \rangle, \ldots \rangle \) be the sequential unit derivation over the definite program \( D \) and the goal \( q \). Assume that \( a \) is a correct answer for \( D \cup \{ \leftarrow q \} \). Then there are \( i \in \mathbb{N} \), an atomic formula \( a_i \), a substitution \( \theta \) such that

\[
\bar{a} \in A_i \land q\sigma = a\theta.
\]

The proof idea originates from the completeness proof of SLD-resolution in [Llo87]. Here, however, we see that the proof is simpler than expected because the ground lemma is surprisingly strong.

**Proof:** Suppose \( q\sigma \) has variables \( x_1, \ldots, x_n \) (if ground: \( n = 0 \)). Let \( r_1, \ldots, r_n \) be distinct constants not appearing in \( D \) or in \( q \) or in \( \sigma \). Let \( \gamma \) be the substitution \( \{ x_1/r_1, \ldots, x_n/r_n \} \). \( q\sigma\gamma \) is ground. Since \( \sigma \) is a correct answer, \( q\sigma\gamma \) belongs to the least Herbrand model \( \mathbb{M}_D \cup \{ \} \). Furthermore, \( q\sigma\gamma \) is an instance of \( q \) and \( \bar{a} \in Q_0 \). By the ground lemma (page 35), there is \( i \in \mathbb{N} \) and \( \bar{a} \in A_i \) such that \( q\sigma\gamma \) is an instance of \( a \). There is a substitution \( \rho \) such that \( q\sigma\gamma = a\rho \). In order to get the substitution \( \theta \), we replace all occurrences of the constants \( r_1, \ldots, r_n \) within \( \rho \) by the corresponding variable of \( x_1, \ldots, x_n \). Since the constants \( r_1, \ldots, r_n \) occur neither in \( D \) nor in \( q \), they do not occur in \( a \). Furthermore, \( r_1, \ldots, r_n \) do not occur in \( \sigma \). Thus \( q\sigma = a\theta \).

The lifting theorem, as a last result in this subsection, describes the relationship of two sequential unit derivations, one of which is for a more general query than the other. On the one hand, for each derived query
or answer in the less general case there is a corresponding more general formula derived after the same number of completion steps in the more general case. On the other hand, each completion step for the more general query might consume more effort if more program clauses are involved. The theorem has been designed for the proof of the completeness theorem but has not been used since simpler ways have been found:

**THEOREM 5. (Lifting theorem)** Let \( D \) be a definite program and \( g, g' \) atomic formulas. Assume \( g' \) is an instance of \( g \). Let

\[
<<Q_0, A_0>>, <<Q_1, A_1>>, \ldots
\]

be the sequential unit derivation over \( D \) for \( g \) and

\[
<<Q'_0, A'_0>>, <<Q'_1, A'_1>>, \ldots
\]

the sequential unit derivation over \( D \) for \( g' \). For each answer of the second derivation there is a corresponding more general answer of the first derivation, i.e. for all \( i \in \mathbb{N} \):

\[
(\forall a' \in F)(\exists a \in F)(\exists \sigma)
\]

\[ a' \in A'_i \implies a \in A_i \land a' = a\sigma. \]

For each query of the second derivation there is a corresponding more general query of the first derivation, i.e. for all \( i \in \mathbb{N} \):

\[
(\forall q' \in F)(\exists q \in F)(\exists \sigma)
\]

\[ q' \in Q'_i \implies q \in Q_i \land q' = q\sigma. \]

**Proof:** Induction over \( j \in \mathbb{N} \).

**hypothesis for** \( j \): Above formulas (1) and (2) are valid for \( 0 \leq i < j \).

**initiation:** \( A'_0 \) is empty. \( Q'_0 = \{\overline{g'}\} \). \( g' \) is an instance of \( g \).

\( Q_0 = \{\overline{g}\} \).

**inheritance:** Assume induction hypothesis is valid for \( i \).

- Consider the answer \( \overline{a'} \in A'_{i+1} \).
  - If \( \overline{a'} \in A'_i \) then we get the corresponding \( a \in A_{i+1} \) by induction hypothesis since \( A_i \subseteq A_{i+1} \).
  - Otherwise program \( D \) contains a clause

\[
(l_0 \leftarrow l_1 \land \ldots \land l_n)
\]

and there is a query \( \overline{q} \in Q'_i \), answers \( \overline{a'_1}, \ldots, \overline{a'_n} \in A'_i \), and a most general unifier \( \sigma \) with

\[
(l_0 \leftarrow l_1 \land \ldots \land l_n)\sigma = (q' \leftarrow a'_1 \land \ldots \land a'_n)\sigma
\]

and \( l_0\sigma = a' \). By induction hypothesis there are \( \overline{q} \in Q_i \) and \( \overline{a_1}, \ldots, \overline{a_n} \in A_i \) such that \( q' \) is an instance of \( q \) and for each \( j \)
if $1 \leq j \leq n$ then $a'_j$ is an instance of $a_j$. By standardizing the variables apart, we can choose the atomic formulas $q, a_1, \ldots, a_n$ in such a way that there is a most general substitution $\rho$ with
\[(q \leftarrow a_1 \wedge \ldots \wedge a_n)\rho = (q' \leftarrow a'_1 \wedge \ldots \wedge a'_n)\]
and such that variables occurring in $q \leftarrow a_1 \wedge \ldots \wedge a_n$ are distinct from those occurring in $l_0 \leftarrow l_1 \wedge \ldots \wedge l_n$. Since $\rho$ does not act on variables of $l_0 \leftarrow l_1 \wedge \ldots \wedge l_n$ we get that $\rho \sigma$ is a unifier with
\[(l_0 \leftarrow l_1 \wedge \ldots \wedge l_n)\rho \sigma = (q \leftarrow a_1 \wedge \ldots \wedge a_n)\rho \sigma.\]

Let $\theta$ be the most general unifier. By answer derivation, $l_0\theta \in \Lambda_{i+1}$. $l_0 \rho \sigma$ is an instance of $l_0 \theta$. $l_0 \rho \sigma = l_0 \sigma = a'$. $l_0 \theta$ is the corresponding answer.
Thus formula (1) is valid for $i + 1$.

Consider the query $q' \in Q'_{i+1}$.
If $q' \in Q'_i$ then we get the corresponding $q \in Q_{i+1}$ by induction hypothesis since $Q_i \subseteq Q_{i+1}$.
Otherwise program $D$ contains a clause
\[(l_0 \leftarrow l_1 \wedge \ldots \wedge l_n)\]
and there is a query $\overline{a_0} \in Q'_i$, $k < n$, answers $\overline{a_1}, \ldots, \overline{a_k} \in A'_i$, and a most general unifier $\sigma$ with
\[(l_0 \leftarrow l_1 \wedge \ldots \wedge l_k)\sigma = (a'_0 \leftarrow a'_1 \wedge \ldots \wedge a'_k)\sigma.\]
and $l_{k+1} \sigma = q'$. By induction hypothesis there are $\overline{a}_0 \in Q_i$ and $\overline{a}_1, \ldots, \overline{a}_k \in A_i$ such that for each $j$ if $0 \leq j \leq k$ then $a'_j$ is an instance of $a_j$. By standardizing the variables apart, we can choose the atomic formulas $a_0, a_1, \ldots, a_k$ in such a way that there is a most general substitution $\rho$ with
\[(a_0 \leftarrow a_1 \wedge \ldots \wedge a_k)\rho = (a'_0 \leftarrow a'_1 \wedge \ldots \wedge a'_k)\]
and such that variables occurring in $a_0 \leftarrow a_1 \wedge \ldots \wedge a_k$ are distinct from those occurring in $l_0 \leftarrow l_1 \wedge \ldots \wedge l_n$. Since $\rho$ does not act on variables of $l_0 \leftarrow l_1 \wedge \ldots \wedge l_n$ we get that $\rho \sigma$ is a unifier with
\[(l_0 \leftarrow l_1 \wedge \ldots \wedge l_k)\rho \sigma = (a_0 \leftarrow a_1 \wedge \ldots \wedge a_k)\rho \sigma.\]
Let $\theta$ be the most general unifier. By query derivation, $l_{k+1} \theta \in Q_{i+1}$. $l_{k+1} \rho \sigma$ is an instance of $l_{k+1} \theta$. $l_{k+1} \rho \sigma = $

\(^2\)Common technique, see for instance [Llo87] page 41.
\[ l_{k+1}\sigma = q'. \] \[ l_{k+1}\theta \] is the corresponding query.
Thus formula (2) is valid for \( i + 1 \).

3.3. Simple sequential algorithm. The simple sequential algorithm successively applies the recursive equations of definition 8, thus computing all pairs \(<A_i, Q_i>\). The preceding pairs are thrown away as long as the chains increase. The algorithm terminates at the point after which the chain members remain constant, if there is such a point.

Furthermore, for each answer, it produces the corresponding representative instance of the initial query (if unifiable) as soon as the answer is derived. In the simple version of the algorithm we disregard efficiency; the set difference in step 2 below is a rather inefficient operation, since the left-hand side of the difference is (almost) a superset of the right-hand side.

(1) \( Q := \{q\}; A := \{\} \);
(2) \( \Delta Q := \{ \bar{a} \in \mathcal{F} | < D, Q, A > ? a \} \setminus Q \);
    \( \Delta A := \{ \bar{a} \in \mathcal{F} | < D, Q, A > ! a \} \setminus A \);
(3) If \( \Delta Q = \{ \} \wedge \Delta A = \{ \} \) terminate.
(4) For each class in \( \Delta A \) unify a representative answer with the initial query \( q \) and output result.
(5) \( Q := Q \cup \Delta Q \);
    \( A := A \cup \Delta A \);
(6) Go to 2.

Remarks:

- When performing step 5 the \( i^{th} \) time the value of \( Q \) is updated from \( Q_{i-1} \) to \( Q_i \) and the value of \( A \) from \( A_{i-1} \) to \( A_i \) as defined in the preceding section.
- The universal closure of each output formula is a logical consequence of the program \( D \).
- For each correct answer \( \sigma \) a formula is produced after finitely many steps such that \( q\sigma \) is an instance thereof. Hence the algorithm is a semi-decision procedure: Logical consequences of the program can be effectively verified. Conversely, however, the algorithm possibly does not terminate, and the attempt to prove that a given formula is not a logical consequence might fail.
- The fact that the algorithm might not terminate is no obstacle properly composing two programs, using the results of one program as assertions for the other. There is no need to delay starting the second program until the first one terminates: the clause sets of the two programs can simply be joined together, possibly after
renaming conflicting predicates. The algorithm is then applied on the union of the programs.
• In case two equivalence classes of formulas are produced such that one class contains more general formulas than the other one, the more general class can be kept and the less general one discarded. Completeness is not lost if such a subsumption check is applied. The corresponding modifications of the algorithm can be done in step 5 and possibly in step 2.

3.4. Extended sequential algorithm. The extended algorithm prunes the search space. During processing of a clause, a flag is set whenever the initial part contains an atomic formula unified with a new query or answer (representative of a class in Q', respectively A'). The flag is cleared however, if all involved queries and answers are old (representative of a class in Q, respectively A). New queries or answers (class to be joined to ΔQ, respectively ΔA) are only produced when the flag is set. Thus we save the effort deciding uniqueness of a derived formula, whenever the flag is clear.

Furthermore, when there are no new queries (Q' = { }), only those program clauses that contain goals corresponding to new answers (A') need be considered.

(1) Q' := {q}; Q := { }; A' := { }; A := { };
(2) <ΔQ, ΔA> := f(D, Q, A, Q', A');
(3) If ΔQ = { } ∧ ΔA = { } terminate.
(4) For each class in ΔA unify a representative answer with the initial query q and output result.
(5) Q := Q ∪ Q';
    A := A ∪ A';
    Q' := ΔQ;
    A' := ΔA;
(6) Go to 2.

The function f above treats new answers and queries differently from old answers and queries. If Q' and A' cover all queries or answers derived in one step from Q and A, then the result of f consists of the new queries and answers derived in one step from Q ∪ Q' and A ∪ A' : i.e. the function f must meet the following specifications, useful for verifying
implementations:

If
\[
\begin{align*}
\{ \overline{a} \mid <D,Q,A> \rightarrow a \} & \subseteq Q \cup Q' \\
\{ \overline{a} \mid <D,Q,A> \rightarrow ! a \} & \subseteq A \cup A'
\end{align*}
\]
then
\[
\begin{align*}
\Delta Q &= \{ \overline{a} \mid <D,Q \cup Q',A \cup A'> \rightarrow a \} \setminus (Q \cup Q') \\
\Delta A &= \{ \overline{a} \mid <D,Q \cup Q',A \cup A'> \rightarrow ! a \} \setminus (A \cup A').
\end{align*}
\]

**Lemma 2.** Assume that \(f\) meets the specifications above. Let
\[
<<Q_0,A_0>,<Q_1,A_1>, \ldots >
\]
be a sequential unit derivation over \(D\). Then the following equalities hold:
\[
(f(D,Q_i,A_i,Q_{i+1}\setminus Q_i,A_{i+1}\setminus A_i) = <<Q_{i+2}\setminus Q_{i+1},A_{i+2}\setminus A_{i+1}>>)
\]
Proof:
\[
\begin{align*}
\{ \overline{a} \mid <D,Q_i,A_i> \rightarrow a \} & \subseteq Q_{i+1} = Q_i \cup (Q_{i+1} \setminus Q_i); \\
\{ \overline{a} \mid <D,Q_i,A_i> \rightarrow ! a \} & \subseteq A_{i+1} = A_i \cup (A_{i+1} \setminus A_i); \\
\{ \overline{a} \mid <D,Q_{i+1},A_{i+1}> \rightarrow a \} & \subseteq Q_{i+1} = Q_{i+1} \setminus Q_{i+1}; \\
\{ \overline{a} \mid <D,Q_{i+1},A_{i+1}> \rightarrow ! a \} & \subseteq A_{i+1} = A_{i+1} \setminus A_{i+1}.
\end{align*}
\]

4. Parallel derivation

4.1. Derivation rules. Let \(<V,\delta,E>\) be a distributed definite program and \(q\) an atomic formula.

We say the function
\[
V \rightarrow (2^F \times 2^F)^N
\]
\[
v \mapsto <<Q_0^v,A_0^v>,<Q_1^v,A_1^v>, \ldots >
\]
is a parallel unit derivation over program \(<V,\delta,E>\) for goal \(q\) if and only if for each \(v,w \in V\), each \(i \in \mathbb{N}\), all atomic formulas \(a,b\), the following conditions hold:

• start conditions:
\[
A_0^v = \{ \overline{a} \} \land Q_0^v = \{ \overline{a} \};
\]

• inference rules:
\[
\begin{align*}
- A_i^v & \subseteq A_{i+1}^v \land (\exists k \in \mathbb{N})(\{ \overline{a} \mid <\delta(v),Q_i^v,A_i^v> \rightarrow a \} \subseteq A_{i+k}^v); \\
- Q_i^v & \subseteq Q_{i+1}^v \land (\exists k \in \mathbb{N})(\{ \overline{a} \mid <\delta(v),Q_i^v,A_i^v> \rightarrow ! a \} \subseteq Q_{i+k}^v);
\end{align*}
\]
• communication rules:
  - if \( \bar{a} \in A_i \land <v,l,w> \in E \)
    and \( \theta \) most general unifier of \( a \) and \( l \)
    then \( (\exists j > i) \bar{a} \theta \in A_j \)
  - if \( \bar{a} \in Q_i \land <v,l,w> \in E \)
    and \( \theta \) most general unifier of \( a \) and \( l \)
    then \( (\exists j > i) \bar{a} \theta \in Q_j \)

• inverse rules:
  - Allow answers derived by above rules only:
    If \( \bar{a} \in A_{i+1} \setminus (A_i \cup \{ a' \mid \delta(w), Q_i, A_i > ! a' \}) \)
    then \( (\exists j \leq i)(\exists u \in V)(\exists \sigma)(\exists a', \sigma = \delta(u), A_i > ! a', \sigma) \)
    and \( \sigma \) most general unifier of \( a' \) and \( l \);
  - Allow queries derived by above rules only:
    If \( \bar{a} \in Q_{i+1} \setminus (Q_i \cup \{ a' \mid \delta(v), Q_i, A_i > ? a' \}) \)
    then \( (\exists j \leq i)(\exists v \in V)(\exists a', \sigma = \delta(v), A_i > ? a', \sigma) \)
    and \( \sigma \) most general unifier of \( a' \) and \( l \).

In the following the inverse answer rule will be used to prove correctness. For completeness both inference and communication rules are needed. The inverse query rule is not needed for correctness nor for completeness. However, it affects efficiency and the termination behavior of algorithms (whether or not the derivation remains constant after some index).

There is quite a lot of freedom in above rules: There are many different parallel unit derivations for a single distributed definite program and a single goal. Due to the completeness theorem below, fairness is guaranteed no matter which derivation is chosen and the implicit non-determinism of the rules affects neither the set of answers nor the termination behavior of any algorithm computing any proper derivation. However, freedom is important for easy verification of algorithms or implementations and reflects the fact that only a little synchronization of processes is required. Different processes may perform with different speed and different intensity, depending on the local size of program and data.

The consistency of the rules, i.e. the existence of a parallel unit derivation for any distributed definite program and any goal, is shown in the next section by a lemma for the simple parallel algorithm.
**Theorem 6.** (Correctness of parallel derivation) Suppose \( <V, \delta, E> \) is a distributed definite program and
\[
\omega \mapsto \langle \langle Q_0^w, A_0^w \rangle, <Q_1^w, A_1^w >, \ldots \rangle
\]
is a parallel unit derivation over \( <V, \delta, E> \).

Ground instances of parallel derived answers belong to the least local model, i.e., for each atomic formula \( a \) and each ground substitution \( \theta \) for the variables of \( a \):
\[
(Vw \in V)(V_i \in N) \bar{a} \in A_i^w \rightarrow a \theta \in M_{<V, \delta, E>}^w.
\]

**Proof:** Induction on index \( i \) over \( N : A_0^w \) is empty. Suppose for each atomic formula \( a \) and each ground substitution \( \rho \) for the variables of \( a \):
\[
(Vw \in V)(\bar{a} \in A_i^w \rightarrow a \rho \in M_{<V, \delta, E>}^w).
\]

Consider process \( v \). Let \( \bar{a} \in A_{i+1}^w \); Owing to the inverse answer rule, there are three cases:

- \( \bar{a} \in A_i^w \) : By induction hypothesis \( a \rho \in M_{<V, \delta, E>}^w \).
- There is a clause \( (l_0 \leftarrow l_1 \wedge \ldots \wedge l_n) \in \delta(v) \) and a substitution \( \sigma \) such that \( l_0 \sigma = a \) and \( \overline{l_1 \sigma}, \ldots, \overline{l_n \sigma} \in A_i^w \). By induction hypothesis:
  \[
  l_1 \sigma \theta, \ldots, l_n \sigma \theta \in M_{<V, \delta, E>}^w,
  \]
  for each ground substitution \( \theta \) for variables occurring in \( l_0 \sigma, \ldots, l_n \sigma \). Because of process property, \( a \theta \in M_{<V, \delta, E>}^w \).
- There are \( <w, q, v> \in E \), a clause \( (l_0 \leftarrow l_1 \wedge \ldots \wedge l_n) \in \delta(w) \), and substitutions \( \sigma, \rho \) such that \( a = q \rho \) and \( a = l_0 \sigma \rho \) and \( \overline{l_1 \sigma}, \ldots, \overline{l_n \sigma} \in A_i^w \). By induction hypothesis:
  \[
  l_1 \sigma \rho \theta, \ldots, l_n \sigma \rho \theta \in M_{<V, \delta, E>}^w,
  \]
  for each ground substitution \( \theta \) for variables occurring in \( l_0 \sigma \rho, \ldots, l_n \sigma \rho \). Because of process property, \( a \theta \in M_{<V, \delta, E>}^w \). Owing to channel property, \( a \theta \in M_{<V, \delta, E>}^w \).

\( \square \)

**Theorem 7.** (Completeness of parallel derivation) Suppose \( <V, \delta, E> \) is a distributed definite program and
\[
\omega \mapsto \langle \langle Q_0^w, A_0^w \rangle, <Q_1^w, A_1^w >, \ldots \rangle
\]
is a parallel unit derivation over \( <V, \delta, E> \). If an instance of the initial goal or any derived query belongs to the least local model, then it is also an instance of a derived answer:
\[
(Vw \in V)(V_i \in N)(\forall q \in F)(\forall \theta)(\exists j \in N)(\exists a \in F)(\exists \sigma)
\]
\[\overline{q} \in Q_i^w \wedge q \theta \in M_{<V, \delta, E>}^w \rightarrow \bar{a} \in A_j^w \wedge q \theta = a \sigma.\]
Proof: We say process \( w \) acquires a corresponding query \( q \) for \( b \) if there are \( i \in \mathbb{N} \) and \( \bar{q} \in Q^w_i \) such that \( b \) is an instance of \( q \). We say process \( w \) acquires a corresponding answer \( a \) for \( b \) if there are \( j \in \mathbb{N} \) and \( \bar{a} \in A^w_j \) such that \( b \) is an instance of \( a \).

Let \( <i, v> \mapsto B_i^v \) be the approximation for \( <V, \delta, E> \). By approximation theorem the least local model coincides with \( \bigcup_{k \in \mathbb{N}} B_k^v \). Induction over the approximation:

**hypothesis for** \( k \): Each process \( v \) acquires for each element of \( B_k^v \), a corresponding answer if it acquires a corresponding query.

**initiation:** For each process \( v \), \( B_0^v \) is empty.

**inheritance:** Assume that \( b_0 \in B_{k+1}^v \) and process \( v \) acquires a corresponding query \( q \) for \( b_0 \). By definition of the approximation, there are three cases:

- \( b_0 \in B_k^v \). We know by induction hypothesis that \( v \) acquires a corresponding answer.
- There is a clause \( d \in \delta(v) \) and an instance \( b_0 \models b_1 \wedge \ldots \wedge b_n \) of \( d \) such that \( b_1, \ldots, b_n \in B_k^v \). By induction on the position \( h \), process \( v \) acquires for each subgoal of the clause:
  - a corresponding query \( q_h \) for \( b_h \) because of the query inference rule on clause \( d \), query \( q \) and answers \( a_1, \ldots, a_{h-1} \);
  - a corresponding answer \( a_h \) for \( b_h \) by induction hypothesis for \( k \), since we already know that there is a corresponding query \( q_h \).
- There are \( <w, a, v> \in E \) and \( b \in B_k^v \) such that \( b \) is an instance of \( a \). By query communication rule, process \( w \) acquires a corresponding query for \( b \). By induction hypothesis for \( k \), process \( w \) acquires a corresponding answer for \( b \). By answer communication rule, process \( v \) also acquires a corresponding answer for \( b \).

Thus the induction hypothesis for \( k + 1 \) is valid.

\( \square \)

4.2. Simple parallel algorithm. Let \( <V, \delta, E> \) be a distributed definite program and \( g \) an atomic formula as the initial query. Each process \( v \in V \) performs the following algorithm:

(1) \( Q := \{g\}; A := \{\} \)
(2) \( \Delta Q := \left\{ q \in F \mid <\delta(v), Q, A > \equiv q \right\} \setminus Q \);
\( \Delta A := \left\{ a \in F \mid <\delta(v), Q, A > \equiv a \right\} \setminus A \);

(3) For each class in \( \Delta A \) unify a representative answer with initial query \( g \) and output result.

(4) Try to unify a representative answer of each class in \( \Delta A \) with the label of each outgoing edge and send the result to the corresponding process if successful.

Try to unify a representative query of each class in \( \Delta Q \) with the label of each incoming edge and send the result to the corresponding process if successful.

(5) Receive answers \( A' \) and queries \( Q' \) from other processes. (Forward received answers if unifiable with an outgoing edge. Forward received queries if unifiable with an incoming edge.)

(6) \( Q := Q \cup \Delta Q \cup Q' \);
\( A := A \cup \Delta A \cup A' \);

(7) Go to 2.

**Lemma 3.** Given a distributed program \( <V, \delta, E> \) and an atomic formula \( g \) as the initial query. For each \( v \in V \) let us run a process obeying the simple parallel algorithm. There is a parallel unit derivation

\[ w \mapsto <Q_0^w, A_0^w>, <Q_1^w, A_1^w>, \ldots > \]

over \( <V, \delta, E> \) for \( g \) such that the variables \( Q \) and \( A \) of each process \( v \) take exactly the values \( Q_i^v \), respectively \( A_i^v \) for each \( i \).

**Proof:** For each \( v \in V \) let \( Q_0^v = \{ g \} \) and \( A_0^v = \{ \} \) owing to start conditions. Before reaching step 6, we always know that \( Q = Q_i^v \) and \( A = A_i^v \) for some \( i \), already used for definition.

For each \( q \) with \( q \in Q' \) let \( n_q \) be the least \( j \in N \) such that

\[ (\exists w \in V)(\exists q', l \in F)(\exists \sigma) \]
\[ <\delta(w), Q_j^w, A_j^w > \equiv q' \text{ and } \langle v, l, w \rangle \in E \land q' \sigma = l \sigma \land q = q' \sigma \]

For each \( a \) with \( a \in A' \) let \( m_a \) be the least \( j \in N \) such that

\[ (\exists w \in V)(\exists a', l \in F)(\exists \sigma) \]
\[ <\delta(w), Q_j^w, A_j^w > \equiv a' \text{ and } \langle w, l, v \rangle \in E \land a' \sigma = l \sigma \land a = a' \sigma \]

We choose \( k \in N \) such that \( k > i \) and \( k \) is greater than the maximum of

\[ \{ n_q \mid q \in Q' \} \cup \{ m_a \mid a \in A' \} \].

For all \( j \) with \( i < j < k \) we define \( Q_j^v := Q_i^v \) and \( A_j^v := A_i^v \). We further define \( Q_k^v \) and \( A_k^v \) as the value of the corresponding assignments in step 6.

\(^3\)Forwarding is not necessary for parallelizations in chapter IV owing to the existence of direct channels.
The sequence thus defined satisfies the inference rules since the algorithm repeatedly executes steps 2 and 6 as long as new formulas can be derived. The communication rules are satisfied according to steps 2 and 4 (and 5) of the sending processes and step 5 and 6 of the receiving ones. The validity of the inverse rules can be seen stepping through $i \in \mathbb{N}$ and simultaneously assigning the values for $Q''_i$ and $A''_i$ for all processes.

4.3. Extended parallel algorithm.

Let $<V, \delta, E>$ be a distributed definite program and $g$ an atomic formula as the initial query. Each process $v \in V$ performs the following algorithm:

1. $Q' := \{g\}; Q := \{\}; A' := \{\}; A := \{\};$
2. $<\Delta Q, \Delta A> := f(\delta(v), Q, A, Q', A');$
   Additionally, perform the following tasks during execution of function $f$:
   - Try to unify representative answers and queries with channel labels and send results to appropriate processes if successful (cf. section 4.2).
   - Receive answers $A''$ and queries $Q''$ from other processes. (Forward answers and queries if necessary, cf. section 4.2.)
   $\Delta Q := Q'' \cup \Delta Q;$
   $\Delta A := A'' \cup \Delta A;$
3. For each class in $\Delta A$ unify a representative answer with the initial query $g$ and output result.
   If $\Delta Q = \{\} \land \Delta A = \{\}$ wait until new answers or queries arrive and join them to $\Delta A$, respectively $\Delta Q$.
4. $Q := Q \cup Q';$
   $A := A \cup A'$;
   $Q' := \Delta Q;$
   $A' := \Delta A;$
5. Go to 2.

Remarks: $f$ is subject to the same conditions and plays the same role as in the section “Extended sequential algorithm” (3.4).

5. Extension

The definition of the term query derivation in section 2 is not quite appropriate in the case of deterministic algorithms, for instance if the definite program is a translation of composition and primitive recursion schemes and if such a program is used to compute the value of the corresponding function from some arguments. The author's experiences with an implementation prototype show that in such applications each computation step of the resolution algorithm produces only one query or one answer and that at every moment only one process is ready for computation. The other processes are awaiting new queries or answers.
Nevertheless, inherent parallelism in such deterministic programs can be well exploited if the query derivation rule is slightly modified, such that situations of real concurrency occur often. We define an extension of definite programs by introducing an annotation for sequential and parallel conjunctions:

**Definition 13.** We say a formula is an extended definite program clause if and only if it is of the form

\[ a_0 \leftarrow a_1 \circ_1 \ldots \circ_{n-1} a_n \]

with conjunction symbols \( \circ_1, \ldots, \circ_{n-1} \in \{ \land, \| \} \). We call \( \land \) the sequential conjunction and \( \| \) the parallel conjunction. We say \( D \) is an extended definite program if it is a set of extended definite program clauses.

We read clauses as strings containing symbols of \( \{ \leftarrow, \land, \| \} \) and sub-strings representing atomic formulas. We do not support nesting of conjunctions. Nevertheless, it might be easier to understand the next definition taking it that \( \| \) is of higher precedence than \( \land \).

Our intention in distinguishing two conjunction types is to allow the independent derivation of queries for subgoals joined together by the parallel conjunction. Thus a single query derivation step can produce several queries, and then atomic formulas are available for different concurrent processes. For sequential conjunctions, derivations of queries for subgoals on the right-hand side are delayed until answers are available for subgoals on the left-hand side. Thereby variable bindings created on the left-hand side are valid on the right-hand side as well. Thus a programmer can force derived queries for subgoals on the right-hand side to be less general. The precise definition follows:

**Definition 14.** Let \( D \) be an extended definite program, and \( Q, A \) sets of equivalence classes.

query derivation: Let \( q \in F \). We write \( <D, Q, A> \leftarrow q \) if and only if program \( D \) contains a clause \( l_0 \leftarrow l_1 \circ_1 \ldots \circ_{n-1} l_n \) and there are \( k, j \in \mathbb{N} \) with \( 0 \leq k < j \leq n \), some atomic formulas \( q', a_1, \ldots, a_k \), and a substitution \( \sigma \), such that

\[ \overline{q'} \in Q \land \overline{a_1}, \ldots, \overline{a_k} \in A \land q = l_j \sigma \]

and \( \sigma \) is the most general unifier with

\[ (l_0 \leftarrow l_1 \circ_1 \ldots \circ_{k-1} l_{k}) \sigma = (q' \leftarrow a_1 \circ_1 \ldots \circ_{k-1} a_k) \sigma \]

and if \( 0 < k \) then \( \circ_k \) is a sequential conjunction symbol (\( \land \)) and if \( k + 1 < j \) then \( \circ_{k+1}, \ldots, \circ_{j-1} \) are parallel conjunction symbols (\( \| \)).
**answer derivation:** Let $a_0 \in F$. We write $<D,Q,A> ! a_0$ if and only if program $D$ contains a clause $l_0 \leftarrow l_1 \circ_1 \ldots \circ_n l_n$ and there are some atomic formulas $q, a_1, \ldots, a_n$, and a substitution $\sigma$ such that

$$q \in Q \land \overline{a_1}, \ldots, \overline{a_n} \in A \land a_0 = l_0\sigma$$

and $\sigma$ is the most general unifier with

$$(l_0 \leftarrow l_1 \circ_1 \ldots \circ_{n-1} l_n)\sigma = (q \leftarrow a_1 \circ_1 \ldots \circ_{n-1} a_n)\sigma$$

and $\circ_1 \ldots \circ_{n-1} \in \{\land,||\}$.

It is apparent that this definition coincides with the definition in section 2 for extended definite programs containing definite program clauses only (i.e. no parallel conjunctions $||$). Modification of the query derivation rule does not require modifications of the algorithms in sections 3.3, 3.4, 4.2, and 4.3. Adaptation of the proofs for correctness and completeness theorems (lemma 1 and theorems 3, 6, 7) can be done in a straightforward way.
CHAPTER IV
Parallelization of definite programs

1. Definition

**Definition 15.** The distributed definite program \( <V, \delta, E> \) is a parallelization of the definite program \( D \) if and only if

\[
D = \bigcup_{v \in V} \delta(v) \text{ and }
\]

\[
(\forall v, v' \in V)(\forall a_0, \ldots, a_k, a'_0, \ldots, a'_j \in F)(\forall i \in \mathbb{N})(\forall b \in \tilde{H})(\forall \theta, \theta')
\]

\[
(\exists l \in \tilde{F})(\exists \sigma)
\]

\[
(a_0 \leftarrow a_1 \land \ldots \land a_k) \in \delta(v) \land
(a'_0 \leftarrow a'_1 \land \ldots \land a'_j) \in \delta(v') \land
\]

\[
a_0 \theta = b = a'_i \theta' \land
1 \leq i \leq j
\]

\[
\{ v = v' \lor
(\langle v, l, v' \rangle \in E \land b = l \sigma)
\}
\]

Parallelizations of extended definite programs are defined analogously. The parallel conjunction || might stand in place of any sequential conjunction \( \land \) in above definition.

In order to parallelize a definite program, we can partition the program clauses in an arbitrary way and then determine the necessary channels. We express this by the following lemma:

**Lemma 4.** Let \( V \) be a set. Let \( D \) be an (extended) definite program and \( \delta : V \rightarrow 2^D \) any function with \( \bigcup_{v \in V} \delta(v) = D \). Then there is a set of channels \( E \) such that \( <V, \delta, E> \) is a parallelization of \( D \).

**Proof:** We rename the variables of the clauses in \( D \) in such a way that there are not two different clauses with occurrences of the same variable. We modify \( \delta \) accordingly and denote the result by \( \delta' \). Let \( \text{mgu}(a, a') \) denote the most general unifier of the atomic formulas \( a \) and \( a' \). We choose a representative element of each equivalence class of atomic formulas, i.e.
we choose a function $r : F \rightarrow F$, such that $(\forall a \in F)r(a) \in \bar{a}$. We define:

$$E := \{ <v, r(a_0 \text{ mgu}(a_0, a'_0)), w> \mid (\exists a_0, \ldots, a_k, a'_0, \ldots, a'_l \in F)(\exists i \in \mathbb{N})$$

$$\hspace{1cm} (a_0 \leftarrow a_1 \land \ldots \land a_k) \in \delta'(v) \land$$

$$\hspace{1.5cm} (a'_0 \leftarrow a'_1 \land \ldots \land a'_l) \in \delta'(w) \land$$

$$\hspace{2.5cm} 1 \leq i \leq j \land v \neq w \}$$

Then $<V, \delta, E>$ is a parallelization of $D$ by definition of parallelizations.

\[ \square \]

2. Example

We illustrate the definition of the previous section with an example. For functions and relations we use a notation similar to built-in predicates in PROLOG. The built-in predicates correspond to the following program fragments:

- \texttt{gt(s(X),X).} \hspace{1cm} \texttt{gt(s(X),Y) \leftarrow gt(X,Y).}
- \texttt{ge(X,X).} \hspace{1cm} \texttt{ge(s(X),Y) \leftarrow ge(X,Y).}
- \texttt{minus(X,X,0).} \hspace{1cm} \texttt{minus(s(X),Y,s(Z)) \leftarrow minus(X,Y,Z).}
- \texttt{plus(X,0,X).} \hspace{1cm} \texttt{plus(s(X),Y,s(Z)) \leftarrow plus(X,Y,Z).}
- \texttt{times(0,X,0).} \hspace{1cm} \texttt{times(s(X),Y,Z) \leftarrow times(X,Y,W) \land plus(Y,W,Z).}

The following shorthand notations are used:

- \texttt{shorthand} \hspace{1cm} \texttt{substitute}
- \texttt{X>Y} \hspace{1cm} \texttt{gt(X,Y)}
- \texttt{X\leq Y} \hspace{1cm} \texttt{ge(Y,X)}
- \texttt{X is Y + Z} \hspace{1cm} \texttt{plus(Y,Z,X)}
- \texttt{X is Y + Z * W} \hspace{1cm} \texttt{times(Z,W,U) \land plus(Y,U,X)}
- \texttt{X is Y - Z} \hspace{1cm} \texttt{minus(Y,Z,X)}
- \texttt{1} \hspace{1cm} \texttt{s(0)}

The reader may read the example below as if the shorthand notations stand for the specified substitutes. However, experienced PROLOG programmers may leave the example unchanged and extend the algorithms in the previous chapter by versions using machine instructions.

The example solves the following combinatoric problem: Let $Dom$ and $Ran$ be finite sets. Let $n$ be the number of elements of $Dom$ and $m$ the number of elements of $Ran$. How many surjective functions with domain $Dom$ and range $Ran$ are there?

The solution has the following recursive definition: To find the number of surjective functions from $Dom$ to $Ran$, choose an element $e$ of $Ran$,
compute for each \( k \) (\( 1 \leq k \leq n \)) the number of surjective functions from a set with \( n - k \) elements to \( \text{Ran} \setminus \{e\} \), multiply by the number of subsets of \( \text{Dom} \) with \( k \) elements and take the sum of these products.

\[
s_{n,m} := \sum_{k=1}^{n} s_{n-k,m-1} b_{n,k}
\]

\[
b_{m,m} := 1
\]

\[
b_{m,0} := 1
\]

\[
b_{m,k} := b_{m-1,k-1} + b_{m-1,k} \text{ if } 0 < k < m
\]

\( b_{m,k} \) denotes the binomial coefficient \( \binom{m}{k} \).

The program must satisfy the following specifications:

\[
\begin{align*}
\text{surj}(N,M,X) & \iff X = s_{N,M} \\
\text{binom}(N,K,B) & \iff B = b_{N,K} \\
\text{surj}(N,M,K,X) & \iff X = \{ \text{number of surjective functions } f \text{ from } \text{Dom} \text{ to } \text{Ran}, \text{ such that } \{ x \mid f(x) = e \} \text{ contains at least } K \text{ elements.} \}
\end{align*}
\]

We define a function \( \delta : \{p_0, p_1, p_2\} \rightarrow \mathcal{D} : \)

\[
\delta(p_0) := \begin{cases} 
\text{surj}(0,0,1). & \\
\text{surj}(N,0,0) & N > 0. \\
\text{surj}(N,M,X) & M > 0 \land \text{surj}(N,M,1,X). \\
\text{surj}(N,M,K,0) & K > N. \\
\text{surj}(N,M,K,E) & K \leq N \land J \text{ is } K + 1 \land I \text{ is } N - K \land L \text{ is } M - 1 \land \text{binom}(N,K,D) \quad \| \quad \text{surj}(N,M,J,B) \quad \| \quad \text{surj}(I,L,C) \land E \text{ is } B + C \ast D.
\end{cases}
\]

\[
\delta(p_1) := \begin{cases} 
\text{binom}(N,K,B) & K > 0 \land N > K \land M \text{ is } N - 1 \land J \text{ is } K - 1 \land \text{binom}(M,K,C) \quad \| \quad \text{binom}(M,J,D) \land B \text{ is } C + D.
\end{cases}
\]

\[
\delta(p_2) := \begin{cases} 
\text{binom}(N,0,1). & \\
\text{binom}(N,N,1).
\end{cases}
\]

We get:

\[
\langle \{p_0,p_1,p_2\}, \delta, \langle \langle p_1, \text{surj}(N,1,X), p_0 \rangle, \langle p_0, \text{surj}(I,L,C), p_1 \rangle, \langle p_2, \text{binom}(N,K,B), p_1 \rangle \rangle \}
\]

is a parallelization of \( \delta(p_0) \cup \delta(p_1) \cup \delta(p_2) \).

\textbf{Remark:} The symbol \( \| \) denotes the parallel conjunction as defined in section 5 of chapter III.
3. Theorem

The term parallelization is defined in such a way that there is a close relationship between the distributed model of a parallelization of the definite program $D$ and the Herbrand model $M_D$:

Each local model of a process is a subset of the corresponding Herbrand model $M_D$. Conversely, for each atomic formula occurring in the body of any local program clause, the associated projection of the Herbrand model $M_D$ is a subset of the local model.

This is expressed by the following theorem:

**Theorem 8. (Parallelization theorem.)** If $<V, \delta, E>$ is a parallelization of the definite program $D$ then

$$\forall v \in V \exists v, \delta, E \subseteq M_D$$

and for each process $v \in V$ and each clause $(a_0 \leftarrow a_1 \land \ldots \land a_k) \in \delta(v)$

$$\bigcup_{i=1}^{k} \pi_{a_i}(M_D) \subseteq M_{<V, \delta, E>}^v.$$

**Proof:**

- We would like to prove $M_{<V, \delta, E>}^v \subseteq M_D$: Let $<j, v> \mapsto B_j^v$ be the approximation for $<V, \delta, E>$. By the approximation theorem:

$$M_{<V, \delta, E>}^v = \bigcup_{j \in \mathbb{N}} B_j^v.$$

Induction over the approximation $B$:

**induction hypothesis for n:** $(\forall v \in V)B_n^v \subseteq M_D$.

**initiation:** $(\forall v \in V)B_0^v = \{\}$.  

**inheritance:** Let $v \in V$ and $b \in B_{n+1}^v$. By definition of the approximation one of the following conditions holds:

- $b \in T_{\delta(v)}(B_n^v)$:
  By induction hypothesis, $B_n^v \subseteq M_D$, $\delta(v) \subseteq D$ since $<V, \delta, E>$ is a parallelization of $D$.
  $$T_{\delta(v)}(B_n^v) \subseteq T_D(B_n^v) \subseteq M_D.$$ Thus $b \in M_D$.

- $(\exists w \in V)(\exists l \in F)(<w, l, v> \in E \land b \in \pi_{l}(B_n^w))$:
  By induction hypothesis, $B_n^w \subseteq M_D$, $\pi_{l}(B_n^w) \subseteq B_n^w$.
  Thus $b \in M_D$.

Thus $B_{n+1}^v \subseteq M_D$.

- We would like to prove $\pi_{a_i}(M_D) \subseteq M_{<V, \delta, E>}^v$. By coincidence of operational semantics with the least Herbrand model (see section 3 of the introduction) we know that $M_D = \bigcup_{m \in \mathbb{N}} T_D^{(m)}(\{\})$. We use induction on fixpoint iteration:
induction hypothesis for $k$:

If $b \in T_D^{(k)}(\{\})$ and $v \in V$

and $(a_0 \leftarrow a_1 \land \ldots \land a_n) \in \delta(v)$ and $i \in \{1, \ldots, n\}$

and $b \in \pi_{a_i}(M_D)$ then $b \in M_{\leq V, \delta, E}$.

initiation: $T_D^{(0)}(\{\})$ is empty.

inheritance: Let $b \in T_D^{(k+1)}(\{\})$ and $b \in \pi_{a_i}(M_D)$. There is a clause $(a_0 \leftarrow a_1 \land \ldots \land a_n) \in D$ and a substitution $\sigma$ such that $l_0 \sigma = b$ and $l_1 \sigma, \ldots, l_n \sigma \in T_D^{(k)}(\{\})$. There is a process $w \in V$ such that $(l_0 \leftarrow l_1 \land \ldots \land l_n) \in \delta(w)$. Furthermore

$l_1 \sigma, \ldots, l_n \sigma \in M_D$ and $l_1 \sigma \in \pi_{l_1}(M_D), \ldots, l_n \sigma \in \pi_{l_n}(M_D)$.

By induction hypothesis for $k$: $l_1 \sigma, \ldots, l_n \sigma \in M_{\leq V, \delta, E}$.

By process property $b \in M_{\leq V, \delta, E}$. $b$ is an instance of $a_i$ since $b \in \pi_{a_i}(M_D)$. $<V, \delta, E>$ is a parallelization $D$. Thus if $w \neq v$ there exists $q \in F$ such that $b$ is an instance of $q$ and $<w, q, v> \in E$. By channel property $b \in M_{\leq V, \delta, E}$. Thus, induction hypothesis for $k+1$ holds.

Thus $M_D \cap \pi_{a_i}(M_D) \subseteq M_{\leq V, \delta, E}$. Since $\pi_{a_i}(M_D) \subseteq M_D$, we get $\pi_{a_i}(M_D) \subseteq M_{\leq V, \delta, E}$. □
Index

\|, 9, 48, 49, 51, 53
←, 8, 15, 18, 19, 28–31, 34–39, 44, 45, 48, 49, 51–55
∧, 8, 28–31, 34–36, 38, 39, 44, 45, 48, 49, 51–55

And-parallelism, 3, 5, 19, 21
answers, 34
approximation, 29
approximation theorem, 29
backtracking, 16
breadth-first search, 18

channel property, 28
commitment, 19
communication rules, 43
Concurrent Prolog, 19, 23
correct answer, 37

D, 8, 9, 27, 53
definite program, 14
definite program clause, 14
depth-first search, 16
distributed definite program, 20
distributed model, 28

F, 7–9, 27, 28, 33, 35, 40, 42–44, 46, 48, 49, 51, 52

ground, 11
Guarded Horn Clauses, 19, 23

H, 7, 9, 14, 28–30, 35, 51
Herbrand base, 14
Herbrand interpretation, 14
Herbrand model, 14, 36
Herbrand universe, 14
Horn clause, 13
Hyperresolution, 20

inference rules, 42
input refutation, 12
input resolution, 12
inverse rules, 43

least distributed model, 28
least local model, 28
leftmost computation rule, 16
linear, 24
local model, 28
logical consequence, 36

M_D, 8, 35, 37, 54, 55
model, 36
M_{Δ(<ϕ,δ,ε>)}, 30, 31
M'_{Δ(<ϕ,δ,ε>)}, 30, 31, 44, 54, 55
M_{<ϕ,δ,ε>}, 9, 28, 31, 44, 55
M(D), 14, 15
N, 8, 9, 15, 29, 34, 35, 37, 38, 42, 44–48, 51, 52, 54

Or-parallelism, 3, 5, 18, 21, 23
parallel unification, 20
parallel unit derivation, 42
parallelization, 54
PARLOG, 19, 22, 23
positive-unit refutation, 13
procedural semantics, 20
process property, 28
PROLOG, 3, 5, 16–19, 22, 23, 52
queries, 34

reduction, 20
refutation, 11

Search-parallelism, 3, 5, 20, 21
sequential unit derivation, 34
SLD-resolution, 15
Stream-parallelism, 3, 5, 19
strictly-input refutation, 14
strictly-positive-unit refutation, 13

T_{ψ(u)}, 28, 29, 33, 54
T_D, 8, 14, 15, 23, 28, 29, 35–37, 54, 55

unit refutation, 11
unit resolution, 11
universal closure, 36
Bibliography


Curriculum vitae

Nov. 14, 1961  Born in Bülach (ZH), citizen of Embrach (ZH), Switzerland.

1968-1974  Primary school in Bülach.
autumn 1980  Matura Typus B.
winter 84/85  Master thesis with the title “Manipulation von Pfeiltermen” (i.e. computations in Engeler graph models), supervised by Prof. E. Engeler.
spring 1985  Degree of “Diplomierter Mathematiker ETH”.
1985-1993  Assistant, system manager, and doctoral student of Prof. E. Engeler at the Department of Mathematics of ETH Zürich.