AUTOMATIC COMPILE–TIME PARALLELIZATION OF LOGIC PROGRAMS FOR RESTRICTED, GOAL–LEVEL, INDEPENDENT AND–PARALLELISM

K. MUTHUKUMAR, F. BUENO,
M. GARCÍA DE LA BANDA, M. HERMENEGILDO

▷ A framework for the automatic parallelization of (constraint) logic programs is proposed and proved correct. Intuitively, the parallelization process replaces conjunctions of literals with parallel expressions. Such expressions trigger at run-time the exploitation of restricted, goal–level, independent and–parallelism. The parallelization process performs two steps. The first one builds a conditional dependency graph (which can be simplified using compile-time analysis information), while the second transforms the resulting graph into linear conditional expressions, the parallel expressions of the &-Prolog language. Several heuristic algorithms for the latter ("annotation") process are proposed and proved correct. Algorithms are also given which determine if there is any loss of parallelism in the linearization process with respect to a proposed notion of maximal parallelism. Finally, a system is presented which implements the proposed approach. The performance of the different annotation algorithms is compared experimentally in this system by studying the time spent in parallelization and the effectiveness of the results in terms of speedups. ◁

Keywords: Automatic Parallelization, Parallelizing Compilers, Conditional Dependency Graphs, And-Parallelism, &-Prolog.

This paper is an extended and updated version of [58]. It was to appear originally in the special issue on High Performance Logic Programming Systems of the Journal of Logic Programming, Vol 29, Nov. 1996.

Address correspondence to M. Hermenegildo, F. Bueno, Universidad Politécnica de Madrid (UPM), Facultad de Informática, 28660-Boadilla del Monte, Madrid, SPAIN. email: {bueno,herme}@fi.upm.es; M. García de la Banda, Dept. of Computer Science, Monash University, Clayton Vic. 3168, Australia. email: mbanda@cs.monash.edu.au; K. Muthukumar, Apple Computer Inc., MS: 303-3D, 3 Infinite Loop, Cupertino, CA 95014, U.S.A. email: muthu@apple.com

THE JOURNAL OF LOGIC PROGRAMMING

655 Avenue of the Americas, New York, NY 10010 xxxx-xxxx/xx/$7.00

1. Introduction

Parallel execution (or- and and–parallelism [22, 19]) has now been proved to be an effective technique for achieving improved performance in logic programming systems. In general, in or–parallel models (see [55, 2, 81] and their references) all alternatives which match a given goal can be safely run in parallel. However, in and–parallel models goals in the body of a clause cannot in general be freely executed in parallel, since this can result in incorrect and/or inefficient executions.

There are several ways to solve the above mentioned problems. One, which is central to the work presented herein, is to allow parallel execution only of goals which are *independent* (we will discuss other possible solutions later). Parallel execution models for logic programs which adopt this solution are said to exploit *independent and–parallelism*. Early notions of independence of goals were proposed by Conery and DeGroot [22, 27]. They provided sufficient conditions for ensuring that the goals to be run in parallel would not produce "binding conflicts". The lack of such binding conflicts not only avoids erroneous results but it also simplifies the implementation and reduces the overhead of parallel execution (for example, no locking of variables is required). More recently, independence has been defined simply as a condition which guarantees both correctness and (time) efficiency (by ensuring that no "slow–down" will occur) with respect to the sequential execution [44, 46]. I.e., independence implies that parallel execution preserves the "observables" in terms of answers, side-effects, and computational complexity of the original program. Several notions of independence, including the traditional notions, have been proved to be correct and efficient in the above mentioned sense, by showing that the search space of the sequential program is preserved (in addition to ensuring that there will be no binding conflicts). This view has also allowed proposing new, more lax notions of independence [27, 83, 45, 46] which allow more goals to be run in parallel, and extending the notion of independence to constraint logic programming [24, 32].

In independence–based and–parallel models it is necessary to determine which goals are independent and therefore eligible for parallel execution. Although this can be done at run–time [22, 54], it can imply significant overhead. It is thus interesting to perform as much of the work as possible at compile–time. Chang [17] proposed an approach which generated a single dependency graph for a clause from a worst case analysis. The approach was somewhat limited, mainly because of the global analysis technology available at the time. DeGroot [27] proposed a way of representing a fixed set of execution graphs via an expression generated at compile–time, choosing among the alternatives at run–time through some checks. Hermenegildo [38] proposed an extension of Prolog, &-Prolog, which allows writing such conditional parallel expressions within the source (&-Prolog) language (they are in this case referred to as "annotations"). This has the advantage that the parallelization can be expressed at the user level, as a source to source transformation, and that the user can directly write parallel programs if so desired (the compiler then checking such code for correctness). An efficient implementation of &-Prolog was also proposed in the form of a parallel abstract machine (the RAP-WAM) [39], an extension of the Warren Abstract Machine [80, 1].

DeGroot also proposed restricting the expressions generated to *linear expressions*, i.e. parenthesized expressions without synchronization primitives. This restriction basically corresponds to a conditional "fork and join" paradigm, and the type of and–parallelism generated is thus called "restricted". We argue with DeGroot that the fork and join paradigm has certain advantages. First, the structure of the resulting parallelism is easier for the user to understand (for example, using visualization tools [16]). This allows programmers to more easily predict the effectiveness of the parallelization of their programs. The parallelized program consists only of parenthesized expressions using sequential or parallel conjunction and conditionals, which is arguably easier to understand than programs which use arbitrarily synchronization primitives. The backtracking behavior of the parallel program is also simpler and easier to understand [43]. Finally, the restrictions also simplify the implementation of the parallel system (specially the backtracking) somewhat, and allow a number of optimizations [42].

For the reasons mentioned above, we argue that goal-level, restricted, independent and–parallelism is, from a practical point of view, a quite interesting model for exploitation of and–parallelism in (constraint) logic programs. We address in this paper an essential aspect of the problem of automatic parallelization within this model: the generation at compile–time of linear expressions, with minimal loss of parallelism within the restrictions imposed by the model. We use &-Prolog as the target language for the sake of concreteness and because of the convenience of its Prolog–compatible syntax, which makes it possible to describe the parallelization techniques as a source to source transformation of the original (constraint) logic program.[1] However, we argue that our results are not only useful for the &-Prolog run–time system itself, but also for other systems using similar annotations and/or the same type of parallelism such as, for example, Kale's ROPM [64], the ACE and &ACE systems [35, 62], the IAP subset of the DDAS model [72], etc.

Several alternative approaches to the one that we address have been proposed in the context of and–parallelism. These include for example the interesting class of the committed–choice languages [20, 78, 77, 74, 71, 70, 76, 75, 66, 67], which exploit *stream* and–parallelism. Synchronization is in this case expressed directly in the language. Dependencies are taken care of by incrementally passing variable instantiations as streams between processes. Unfortunately, this class of languages does not support "don't know" nondeterminism [53]: once a branch has been chosen, it is never backtracked. The model is very interesting from the point of view of concurrent execution, but not as appealing for general purpose logic programming where backtracking is one of the most useful features. From the point of view of efficiency, interesting work is being done in the concurrent logic programming field in identifying schedulings which are both correct and efficient, possibly sequentializing some processes [84, 52]. A form of don't know non-determinism can be implemented in these languages by performing program transformations where the backtracking or or–parallelism is folded into the and–parallelism [79, 70]. This transformation has been extended by Bansal [5] and used to parallelize logic programs by translating them into committed–choice programs, using global analysis [6]. This work is interesting and has many objectives in common with ours. It provides for example quite useful transformations for parallelizing executions where a producer and a consumer are deeply intertwined. On the other hand the approach differs from ours in several key points. First, it does not focus on restricted and–parallelism, and thus, it does not address the particular and interesting problems that arise as a consequence of

[1] Recently, an extension to &-Prolog with constraints, among other things, has been defined: CIAO-Prolog [47, 41]. We will consider this constraint version of &-Prolog as the target language for our transformations, since we will deal with generic constraint logic programs.

using the fork and join paradigm. Also, it does not directly address independence, and thus it is unclear whether it can guarantee the correctness and efficiency of the parallel execution with respect to sequential execution. The need to intertwine the support for a "don't know" semantics in the transformation process makes the presentation less accessible in our view. Finally, although no performance figures are provided by Bansal, it appears that the final performance of this approach may suffer from the fact that there is a certain level of meta-interpretation and also some overhead (due to the support of general parallelism) in the underlying system.

Other interesting alternative models include the PNU-Prolog approach of Naish [61] and the Andorra model proposed by D.H.D. Warren [7, 65]. In these approaches only deterministic goals, or, more precisely, deterministic *reductions*, are allowed to run in parallel. The advantage of these models is that they achieve essentially the same results as the committed-choice languages, while preserving non-deterministic search. However, they have the disadvantage of not being able to parallelize non-deterministic computations that are independent, and also they involve a higher run–time overhead than that involved in supporting goal-level, restricted, independent and–parallelism. Another interesting approach has been proposed by Shen, the DDAS model [72], which essentially combines goal-level, restricted, independent and–parallelism with a form of dependent and–parallelism. The main idea regarding the exploitation of dependent and–parallelism is to allow dependent goals to run in parallel, but marking their shared variables specially. Synchronization is achieved at the binding level through the dependent variables by using a left-to-right binding priority scheme implemented via token passing. The main drawback of this approach is the overhead involved in the management of the dependent variables and the priority scheme. In any case, and as mentioned above, the solutions that we propose are directly applicable to the independent and-parallel subset of DDAS.

The Extended Andorra Model of D.H.D. Warren [82] is aimed at enhancing the basic Andorra model to also support independent and–parallelism. This model opens interesting possibilities, but a number of issues are left open and need to be resolved in a satisfactory way in an implementation. The Andorra Kernel Language AKL [51] is a concurrent language based on this model, in which synchronization is partly controlled by the programmer through guards and partly by the model, based on determinacy and "stability" conditions. Interestingly, stability has been found to be directly related to independence [47]. However, AKL offers an explicit concurrent programming model which is quite different from that of logic programming, which is our target. IDIOM [37] is another model aimed at parallelizing both deterministic and non-deterministic goals.

Finally, [47] studies the relation between the previously proposed models and proposes a unifying view. The observation is made that most of the models proposed for exploiting parallelism in logic programs (including all those mentioned above) can be explained and reconstructed by starting from a general model and applying a few basic parallelization conditions (such as, for example, "independence" or "determinacy") at different *levels of granularity* (such as, for example, the "goal" level or the "binding" level). Based on these ideas, a formal model capable of exploiting as much parallelism as any previously proposed model, and at a very fine level of granularity, was proposed in [11, 13, 12, 63].

The fork and join model for parallel execution has also been found interesting and studied in the context of functional languages. For example, in the context of functional programming, Sarkar [68, 69] defines an algorithm for finding optimal linearizations of a graph into fork and join non–nested expressions, based on available information on granularity. However, the approach is not in general applicable to the problem at hand, since it considers neither nested nor conditional expressions. Algorithms proposed in the context of imperative languages either do not deal with conditional parallelism or are not focused on the fork and join paradigm [29, 4, 34]. We argue that the results presented herein (parallelization framework using conditional dependency graphs and algorithms for linearizing such graphs into parallel conditional parallel expressions, as well as their correctness proofs) can be easily applied to other programming paradigms and can help deal with difficult problems such as dealing with irregular computations (see [40] for more details on this interesting issue).

Guidelines for constructing correct annotations at compile–time for goal-level, restricted, independent and–parallelism were to the best of our knowledge first proposed in [38]. DeGroot [28] proposed a technique for generating graph expressions using a simple heuristic. However, the expressions generated with this method tend to be rather large, with a significant number of checks. Furthermore, the method has no provision for conjunctions of checks, because DeGroot's original expressions did not include this possibility. However, conjunctions of checks appear to be quite useful in practice and are supported by the &-Prolog language [38]. Jacobs and Langen [49] describe a quite interesting framework for compiling logic programs to an extension of DeGroot's graph expressions equivalent to that introduced in [38]. They propose two rules (SPLIT and IF rules) for transforming a dependency graph (such as those used in [54, 22, 17]) into graph expressions. Interesting groundwork is set by describing such rules, but no algorithm or set of heuristics is given that would suggest how and when to use such rules in a parallelization process. The approach therefore doesn't represent a complete algorithm for our purposes. Complete algorithms in this sense were first given to the best of our knowledge in [58], of which this paper is an extension.

In general, the task of parallelizing a given program through compile–time analysis can be conceptually viewed in our framework as comprising two steps: (1) a local or global analysis of the program in order to gather as much information as possible regarding the terms to which program variables will be bound, and (2) given that information, a rewriting of the program into another one which contains expressions which will cause the parallel execution of some goals, perhaps under certain run–time conditions. Elaborating on the work presented in [58], we present (a) a methodology for automatically extracting parallelism at compile–time with the aid of program analysis, (b) algorithms which determine if a given clause can be compiled into an &-Prolog parallel expression without loss of parallelism (within the model exploited, i.e. restricted, goal–level, independent and–parallelism), and (c) algorithms for compiling (rewriting) logic programming clauses into &-Prolog clauses containing parallel expressions. The methodology is generic in the sense of being able to deal with several different notions of independence, and incorporating the role of program analysis information independently of the domain used. The algorithms are complete in the sense of incorporating not only rules for the transformation, as in previous approaches, but also heuristics to decide when to apply the rules. Essentially, the heuristics seek to lose as little parallelism as possible (hence, the motivation of (b)), while, at the same time, keeping the overhead associated with such parallelism low. The rest of the paper proceeds as follows: after initial preliminaries in Section 2, we present the methodology based on compile–time analysis and transformation in Section 3. In Section 4 we present the compilation process for unconditional parallelism. We first deal with the important problem of characterizing in which cases a clause can be compiled into *linear* parallel expressions without loss of parallelism in the linearization process, and then present an algorithm to actually perform the compilation. We also present an extension to the algorithm for the case when the requirement of not loosing parallelism in the linearization process is relaxed. A similar scheme is followed in Section 5 for the case of conditional parallelism. Section 6 then discusses practical issues, including two new algorithms based on simpler heuristics. Sections 7 and 8 present a comprehensive practical study based on an implementation of these algorithms. Finally, Section 9 presents our conclusions.

2. Notation and Preliminaries

Throughout the paper, we will use the convention that sets of atomic formulae are interpreted as the conjunctive formula of the atomic ones. Sometimes, we will also write an equivalent formula instead of the set; thus *true* for the empty set, and $false$ for the inexisting set. Set difference will be denoted by $A \setminus B = \{x \in A | x \notin B\}$, and the powerset of a set A by $\wp(A)$. The quantification $\exists!$ will be used for "there exists precisely one". A graph will be denoted as the pair (V, E), where V is the set of vertices or nodes, and E is the set of edges representing a binary relation on V (possibly including also a label). We will use E^* for the transitive closure of relation E, and G as a name for a graph. Given a graph $G = (V, E)$, $G|_P$ will denote the subgraph $(P, E|_P)$ of G induced by edges connecting only vertices in $P \subseteq V$. The syntax, and semantics, of the languages we use are introduced in the following. Sometimes, (meta-)expressions of these languages will be enclosed between angle brackets "$\langle \ldots \rangle$" to separate them from plain text.

2.1. Language Syntax and Semantics

Our starting point will be a Prolog or Constraint Logic (CLP) program: a *logic program*, for short. The classical left–to–right operational semantics of Prolog and CLP will be considered (see e.g. [50]). We will denote the computation states in this semantics by a goal and a (constraint) store, as in $\langle g, c \rangle$, where g is the goal and c the store.

Definition 2.1. [(Constraint) logic program] Let $\vec{t}$ be a tuple of terms, c a constraint predicate symbol, and p a non–constraint predicate symbol. The following (simplified) grammar defines the syntax of logic programs:
$Program ::= Clause\ .\ Program \mid \epsilon$
$Clause ::= Atom \mid Atom$:- $Body$
$Body ::= Literal \mid Literal\ ,\ Body$
$Literal ::= Atom \mid Constraint$
$Atom ::= \mathsf{p}(\vec{t})$
$Constraint ::= \mathsf{c}(\vec{t})$

Our target language will be (the constraint version of) &-Prolog. We consider the &-Prolog language as a vehicle for expressing goal–level, restricted, independent and–parallelism.[2] For our purposes, &-Prolog is essentially Prolog (or CLP), with the addition of the parallel conjunction operator "`&`" (used in place of "`,`" — comma — when goals are to be executed in parallel), and a set of parallelism–related built–ins, which includes suitable run–time checks for the notion of independence under which parallelism is to be exploited. For syntactic convenience an additional construct is also provided: the *Conditional Graph Expression (CGE).* A CGE has the general form `(`$cond$ `=>` $goal_1$ `&` $goal_2$ `&` ... `&` $goal_N$`)` and can be regarded as syntactic sugar for the if-then-else expression `(`$cond$ `->` $goal_1$ `&` $goal_2$ `&` ... `&` $goal_N$ `;` $goal_1$`,` $goal_2$`,` ...`,` $goal_N$`)`. &-Prolog if-then-else expressions and CGEs can be nested in order to create richer execution graphs.

Definition 2.2. [Restricted &-Prolog program] Let $\vec{t}$ be a tuple of terms, `c` a constraint predicate symbol, and `p` a non–constraint predicate symbol. The following (simplified) grammar defines the syntax of &-Prolog programs:

$Program ::= Clause\ .\ Program \mid \epsilon$
$Clause ::= Atom \mid Atom$ `:-` $Body$
$Body ::= Literal \mid Literal$ `,` $Body$
$Literal ::= Atom \mid Constraint \mid Body$ `->` $Body$ `;` $Body$
$\quad \mid Body$ `=>` $ParExp \mid ParExp$
$ParExp ::= Body$ `&` $Body$
$Atom ::= \mathtt{p}(\vec{t})$
$Constraint ::= \mathtt{c}(\vec{t})$

Note from the above grammar that "`&`" binds stronger than "`=>`", and this one in turn binds stronger than "`,`". The semantics of "`&`" is defined as follows. Given a state of the computation $\langle (g_1\ \&\ \ldots\ \&\ g_n).s, c\rangle$, its operational behavior is given by the parallel computation of the states $\langle g_1, c\rangle, \ldots, \langle g_n, c\rangle$, giving $\langle \epsilon, c \wedge c_1\rangle, \ldots, \langle \epsilon, c \wedge c_n\rangle$, respectively, and the sequential execution of the continuation $\langle s, c \wedge c_1 \wedge \ldots \wedge c_n\rangle$. See [46] for details.

By *linear expression* we will refer to an expression built according to the syntactic rules given above for the *Body* of a restricted &-Prolog clause, including if-then-elses and CGEs. Note that we do not consider if-then-elses in the source languages. This is no restriction, since a well known transformation can be done from programs with if-then-else to plain syntax by folding them into new predicates. Also, cut ("`!`") is not considered in the syntax. It will be regarded, for our purposes, as a side–effect built–in. Constraints will be also regarded as built–ins. Negation by failure will be regarded as a meta–call. The treatment of built–ins, meta–calls, and side–effects will be considered later.

2.2. Independence in Logic Programs

As mentioned in the introduction, independence refers to the conditions that the run–time behavior of the goals to be run in parallel must satisfy in order to guar-

[2] Note that the &-Prolog language is rich enough to express *unrestricted* and–parallelism through the use of `wait` primitives [57], and at levels of granularity other than the goal level. However, as mentioned before, there is an efficiency penalty associated with this.

antee the correctness and (time) efficiency of the parallelization with respect to the sequential execution. Correctness is guaranteed if the answers obtained during the parallel execution are equivalent to those obtained during the sequential execution. Efficiency is guaranteed if the no "slow-down" property holds, i.e., if the (idealized) parallel execution time is guaranteed to be shorter or equal than the sequential execution time. Though and–parallel execution of goals can generate speculative work in the case of failure, it has been proven in [46] that when goals are independent no slow–down occurs even in this case. Informally, the general, intuitive notion of independence that we want to characterize can be expressed as follows: a goal `q` is independent of a goal `p` if the execution of `p` does not "affect" `q`.

Definition 2.3. [Independent goals] Goal g_2 is independent of goal g_1 for store c iff the execution of $\langle g_2, c\rangle$ does not change the number of computation steps, nor their cost, nor their answers, w.r.t. the execution of $\langle g_2, c'\rangle$, for every store c' resulting from the execution of $\langle g_1, c\rangle$. Goals g_1 and g_2 are independent for c iff g_1 is independent of g_2 for c, and vice versa. Goals in the set $I = \{g_1, \ldots, g_n\}$ are pairwise independent for c iff for every $g_i \in I$, $g_j \in I$, $i \neq j$, g_i and g_j are independent for c.

The conditions for ensuring independence can be divided into two main groups: *a priori* and *a posteriori* conditions. An a priori condition is one that can be checked prior to the execution of the goals involved, and thus can be used as run–time test. For this to be possible, this condition must only be based on the characteristics of the current store and the variables belonging to the goals to be run in parallel. As a result, a priori conditions are restrictive in the sense that they can miss independent goals, due to the lack of information regarding their run–time behavior. On the other hand a posteriori conditions can be based on the actual behavior of the goals to be run in parallel. This has the advantage that the conditions can be defined in such a way that fewer independent goals are missed. In fact, it is possible to define conditions which are not only sufficient but also necessary for ensuring independence, thus detecting *all* independent goals. The problem of course is that it might not be possible to check such conditions without actually running the goals.

Example 2.1. In the context of LP, the first notion of independence was proposed in [22, 27] and formally defined and proved correct in [44, 46]. This condition (referred to as *strict independence*) states that two goals g_1, g_2 are independent w.r.t. a given substitution θ if they do not share variables, i.e. if $\mathrm{vars}(g_1\theta) \cap \mathrm{vars}(g_2\theta) = \emptyset$, where $vars(t)$ is the set of variables in term t. For example, while goals $g_1(x)$ and $g_2(x)$ are independent w.r.t. substitution $\theta = \{x/1\}$ they are not w.r.t. the empty substitution. Note that strict independence is an a priori condition.

Strict independence was later generalized in [27, 83, 45, 46] to different concepts of *non–strict independence*. The intuition behind such generalizations is that goals sharing variables can still be run in parallel, provided the bindings established for those shared variables satisfy certain characteristics. In particular, non–strict independence in [45, 46] requires that at most one goal further instantiate a shared variable, and that no aliasing (of different shared variables) be created during the execution of one of the parallel goals which might affect goals to the right. Obviously, this is an a posteriori condition since the behavior of the goals is taken into account.

Consider the goals `p(X,Y)`, `q(Y)`, and the program:

```
p(X,Y) :- X = Y.
q(X)   :- X = a.
```

It is easy to check that `p(X,Y)` and `q(Y)` are non–strictly independent for the empty substitution, since only `q(X)` further instantiates `X`, and the aliasing created by `p(X,Y)` does not affect two variables shared by `p(X,Y)` and `q(X)` (only `X` is shared). □

Example 2.2. In the context of CLP, several a priori and a posteriori conditions have been defined in [24, 32], showing also that the LP notions can not in general be applied in this context. The most general a posteriori condition proposed in this work (referred to as *search independence*) states that two goals g_1, g_2 are independent from a given constraint store c, if for any partial answer p_1 of g_1 for c and any partial answer p_2 of g_2 for c, p_1 and p_2 are consistent. In fact, this condition is not only sufficient but also necessary. Note also that the condition relies heavily on the run–time behavior of the goals.

The most general a priori condition proposed therein (referred to as *projection independence*) states that goals g_1 and g_2 are independent for constraint c if for any variable $x \in vars(g_1) \cap vars(g_2)$, x is uniquely defined by c (ground in the LP context), and the constraint obtained by conjoining the projection of c over $vars(g_1)$ and the projection of c over $vars(g_2)$ entails (i.e. logically implies) the constraint obtained by projecting c over $vars(g_1) \cup vars(g_2)$. For example, consider the goals $g_1(y), g_2(z)$ and constraint $c \equiv \{y > x, z > x\}$. The projection of c over $\{y\}$ is the empty constraint *true*. The projection of c over $\{z\}$ is also *true*. Since the projection of c over $\{y, z\}$ is also *true*, the condition is satisfied and we can ensure that $g_1(y), g_2(z)$ are search independent for c.

Unfortunately, the cost of performing a precise projection at run-time may be too high. A pragmatic solution [24] is to simplify the run-time tests by just checking if the variables involved are "linked" through the constraints in the store, thus sacrificing accuracy in favor of simplicity. In particular, for the previous example, y and z are linked (through x) in the store, and therefore $g_1(y), g_2(z)$ would not run in parallel. □

In our context, the parallelization process is parameterized by a particular notion of independence. For our purposes, the only requirements are that the independence condition chosen guarantee correctness and efficiency of the parallel execution of the goals involved, that it can be used as a run–time test (i.e., is a priori), and that it satisfies what we call the *grouping property.* If the condition is not a priori, the solution is to first use compile–time analysis to infer as much information about the run–time behavior of the goals as possible. If such information is enough to ensure that the a posteriori conditions are satisfied (i.e. that the goals are independent), no run–time tests will be needed. Otherwise, an a priori condition will then be used for the parallelization of the goals.

Definition 2.4. [Correct and efficient a priori condition (*i_cond*)] A set of conditions associated to a pair of goals g_1 and g_2, is an *i_cond* iff it can be evaluated for any constraint store and if each condition evaluates to true for a given constraint store c then g_1 and g_2 are independent for c.

The importance of the a priori notions of strict and link independence for automatic parallelization does not only come from their run–time efficiency. It is also due to the fact that they satisfy the "grouping" property. This property is important in that it will allow us to consider conditions between goals pairwise, rather than in sets, thus greatly simplifying our framework.

Definition 2.5. [Grouping set of conditions] Consider the goal g, the set of goals $S = \{g_1, \ldots, g_n\}$, and the set $I = \{i\_cond(g_i, g) \mid i \in \{1, \cdots, n\}\}$. The set I is grouping iff if each condition evaluates to true for a given constraint store c, then for any sequence G built from the goals in S, g and G are independent for c. In other words, if for any such G, $i\_cond(G, g)$ is equivalent to the conjunction of the conditions $i\_cond(g_i, g)$ for each g_i in G.

Example 2.3. Although conditions based on strict and link independence are grouping, conditions based on projection independence do not always satisfy the grouping property. For example, for the arithmetic linear constraint $x + y = z$, although $p(x)$ is projection independent of $p(y)$ and of $p(z)$, $p(x)$ is not projection independent of the sequence $p(y), p(z)$. □

When sets of conditions are grouping, the condition between a goal and a set of goals is simply built from the conjunction of the *i_cond*s of the goal with each of the goals in the set. When sets of conditions are not grouping, a much more expensive condition has to be built and tested at run–time. Basically, for each goal which could run in parallel, a set of conditions on this goal w.r.t. all other goals which can run in parallel with it should be tested. Such set, although obviously different from the simple conjunction of the *i_cond*s, can generally be easily built from them. However, the extensions to the framework in the case of a posteriori or non–grouping conditions are beyond the scope of this paper.

A priori conditions also enjoy an important property, which is formalized in the following proposition.

Proposition 2.1. If g_1 and g_2 are a priori independent for c, they are also a priori independent for any constraint c' defined as $c \wedge c_1 \wedge c_2$, where c_1 and c_2 are constraints satisfying $vars(c_1) \subseteq vars(g_1)$ and $vars(c_2) \subseteq vars(g_2)$.

Proving the above proposition is straightforward given the fact that (a) projection independence is not only sufficient but also necessary for ensuring a priori independence, and (b) if g_1 and g_2 are projection independent for c, they are also projection independent for any such c'.

Note that the above proposition does not imply that if a particular *i_cond* is satisfied in c it will also be satisfied in c'. Consider, for example, the condition $i\_cond(p(x), q(y))$ defined as "x and y are strict independent for store c and $x = f(a)$". Although $p(x)$ and $q(y)$ are a priori independent for both $c \equiv true$ and $c' \equiv x = b$, $i\_cond(p(x), q(y))$ holds for c but not for c'. More general independence conditions, as those based on strict, link and projection independence notions, satisfy the above proposition. We call these conditions *reasonably general*. We will use this concept to simplify the proof of correctness of the method proposed in the next section. Note, however, that the crucial point is that Proposition 2.1 holds, and thus the method is still correct for "non-reasonably general" conditions.

3. Compiling Logic Programs into &-Prolog: General Approach

An (&-)*annotation* is a transformation of a program into a parallel version of it. Assuming an appropriate condition and run–time checks, the result of an annotation will then be an &-Prolog program with parallel expressions each annotated with an appropriate test made out of such checks.

Consider a set *Cond* of first order formulae[3] in which the sufficient conditions for independence on the goals of each clause can be expressed. Also, any relevant information on those goals can be captured by formulae of *Cond*. The mapping $i\_cond : \wp(Literal) \rightarrow \wp(Cond) \cup \{false\}$ provides the required condition for a given set of literals in the form of a set of atomic formulae, interpreted as their conjunction (the checks), or $false$. A characteristic of our framework, thanks to the grouping property, is that algorithms only need to inspect conditions between two given goals. For this reason, we restrict the above function to $i\_cond : Literal \times Literal \rightarrow \wp(Cond) \cup \{false\}$.

Example 3.1. Consider strict independence as defined in the previous section. In the Herbrand domain, $indep(x, y)$ is true if x and y do not share variables. Sufficient independence conditions for goals g_1 and g_2 can then be defined as $i\_cond(g_1, g_2) = \{indep(x, y) \mid x \in vars(g_1), y \in vars(g_2), x \neq y\} \cup \{ground(x) \mid x \in vars(g_1) \cap vars(g_2)\}$, where $ground(x)$ is introduced as an efficient run–time test for $indep(x, x)$. Note that, in a practical implementation, these conditions can be reduced further, since for example $indep(x, y)$ and $indep(y, x)$ need not both be considered. □

For the sake of concreteness, our examples and performance study will focus on strict independence for the Herbrand domain, using the checks defined in the previous example.

3.1. Dependency Graphs

The first step in the annotation is concerned with identifying the dependencies between each two goals in a clause and the minimum number of tests for ensuring their independence, based on the sufficient conditions applicable. This step can be viewed as a compilation of programs into (conditional) *dependency graphs*. Consider a relation $prec \subseteq Literal \times Literal$ which captures, in the case of sequential logic programs, the left–to–right precedence relation. A definition of conditional dependency graphs, which additionally provides a method for the first step in the compilation, follows.

Definition 3.1. [Conditional dependency graph] A conditional dependency graph (V, E) for a given sequence of literals $g_1 \ldots g_n$ is given by $V = \{g_1, \ldots, g_n\}$, and $E = \{(g_i, g_j, i\_cond(g_i, g_j)) \mid \{g_i, g_j\} \subseteq V, prec(g_i, g_j), i\_cond(g_i, g_j) \neq \emptyset\}$.

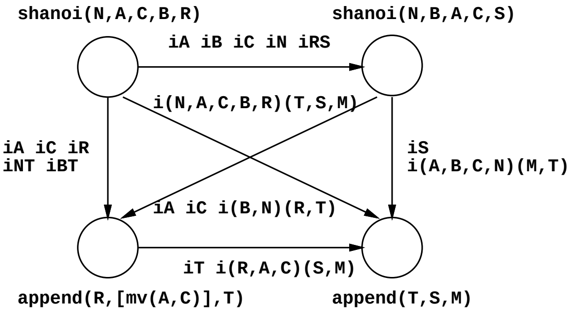
Conditional dependency graphs (CDGs) are labeled directed acyclic graphs (DAGs). Acyclicity is guaranteed by the precedence relation *prec*. The label of

[3]Though *Cond* is here defined over a first order language, its variables are the program clause variables, which can be regarded as constants. If this is done, the "first order" formulation is mere syntactic sugar for a truly propositional language.

an edge states the independence condition for the two goals connected by the edge. If the condition is satisfied, the two goals are independent. If the *i_cond* fails, the two goals are dependent and must be run sequentially. The direction of the edge indicates the proper order in which the goals have to be executed. Note that, since *i_cond* includes sets of atomic formulae or $false$, either of these can be a label. For the sake of conciseness, we will sometimes regard edges labeled $false$ as unlabeled. Also, since the empty set of formulae is equivalent to *true*, we will talk of *true* instead. Note however that from the definition of CDGs, edges labeled *true* are dropped. This is because while for a false condition the edge is still needed for indicating the order of the corresponding sequential execution, for a successful condition the edge is not needed anymore. An important special case of CDGs are graphs where conditions are always false, i.e. dependencies always hold. In this case we will talk of Unconditional Dependency Graphs (UDGs). This class of CDGs is very interesting, since it allows to exploit only unconditional parallelism, thus avoiding any overhead involved in the run–time checks.

Example 3.2. Consider the clause of the familiar program to solve the Towers of Hanoi problem given below. Assume for simplicity that its CDG has edges labeled $false$ from the two built–ins to any other literal on their right (see Section 3.2). The rest of the graph is shown in the figure below. For brevity in the figure, for any two terms `s` and `t` we write `ist` for *indep*(`s`,`t`), and `is` for *indep*(`s`,`s`). Note that for non–variable terms `s` and `t`, evaluating *indep*(`s`,`t`) requires pairwise evaluation of the condition for the variables in each term. Thus, given `s = f(X,Y)` and `t = g(Z,W)`, the test `ist` is equivalent to the test `i(X,Y)(Z,W)` and also to the set of tests {`iXZ, iXW, iYZ, iYW`}.

```
shanoi(N0,A,B,C,M) :-
    N0 > 1,
    N is N0 - 1,
    shanoi(N,A,C,B,R),
    shanoi(N,B,A,C,S),
    append(R,[mv(A,C)],T),
    append(T,S,M).
```



Conditional dependency graphs, as defined, give a parallel execution model for the bodies of the program clauses, i.e. for and–parallelism. This model has been defined in [60] as Maximal Efficient Goal–level Restricted Independent And–Parallelism, which we will call "μ-parallelism". It is maximal within the limits of being both independent and goal–level at the same time, since *goals* are executed *as soon as* they become independent. Because it is independence–based, it is also correct and efficient. The MEIAP model can be identified with the following operational semantics for dependency graphs. Let a node be *ready* if it has no incoming edges.

Definition 3.2. [CDG operational semantics] Given a conditional dependency graph (V, E), parallel execution of the goals to which nodes in V correspond is achieved by repeated application of the following rules:

- Goal initiation: Consider nodes whose incoming edges have source nodes which are ready. If the conditions labeling all these edges are satisfied for the current store c, remove them all. Repeat until no edges are removed. Initiate all goals g in ready nodes by executing $\langle g, c\rangle$ in different environments.

- Node removal: Remove all nodes whose corresponding goals have finished executing, and their outcoming edges. Add the associated answers to the current store c.

Note that, given the form in which CDGs are constructed, there is always at least one node which is ready: that which corresponds to the leftmost goal. Therefore, the execution model can always be initiated. The model is also correct w.r.t. sequential execution.

The proof of correctness is based on the special properties of a priori, grouping independence conditions and, in particular, on the characteristics of the constraints that can be added to the store by goals which are independent under such notions. Given a set of conditions I which is grouping for a goal g and a set of goals $S = \{g_1, \ldots, g_n\}$ the following result ensures that if each $i\_cond(g_i, g)$ in I is satisfied in store c, they will also be satisfied in any subsequent store c' resulting from the execution in c of some of the goals in S. This will later allow us to prove that, at any point in the execution, the independence conditions between each two ready goals are satisfied for the current store.

Lemma 3.1. Consider g, the set of goals $S = \{g_1, \ldots, g_n\}$, and the grouping set I of correct, efficient and reasonably general a priori conditions $\{i\_cond(g_i, g) \mid g_i \in S\}$. If every condition in I is satisfied in store c, then for every sequence G built from the goals in S, and for every store c' obtained by the execution of $\langle G, c\rangle$, each condition in I is still satisfied in c'.

PROOF. Since the set I is grouping, the condition $i\_cond(g_1 \ldots g_n, g)$, is equivalent to the conjunction of the conditions in I, and thus it is satisfied in store c. Since the conditions are reasonably general, $i\_cond(g_1 \ldots g_n, g)$ is also satisfied in any c' obtained by executing any goal defined over a subset of variables in either $g_1 \ldots g_n$ or g, and in particular, in the store c' obtained by the execution of $\langle G, c\rangle$. Thus, every condition in I is also satisfied in such c'. $\square$

Given the above result, it is straightforward to prove that if the conditions for the pairwise independence of a set of goals S are satisfied in store c, the conditions are also satisfied in any store obtained by executing in c some goals in S.

Corollary 3.1. Consider the set of goals $S = \{g_1, \ldots, g_n\}$, and the grouping set I of correct, efficient and reasonably general a priori conditions $\{i\_cond(g_i, g_j) \mid g_i, g_j \in S, i \neq j\}$. If each condition in I is satisfied in store c, then for any sequence G built from goals in S, each condition in I is satisfied in any store c' obtained from the execution of $\langle G, c\rangle$.

Let us now prove that, at any point in the execution, the independence conditions between each two goals determined as ready by the model are satisfied in the current constraint store c'.

Lemma 3.2. Consider a CDG obtained by applying an a priori, reasonably general, grouping notion of independence to the sequence $g_1 \dots g_n$. Let c' be the current constraint store obtained during the execution of the graph following the CDG operational semantics. For every two ready goals g_i and g_j, $1 \leq i < j \leq n$, $i\_cond(g_i, g_j)$ is satisfied in c'.

PROOF. Let us prove this by induction.

- Base case: Let c' be the initial store c. By definition of the model, for each ready goal g_j and each goal g_i, $1 \leq i < j$, $i\_cond(g_i, g_j)$ is satisfied for c.

- Induction step: assume that at some point of the execution the current store is c', and the set of ready goals is R_{old}. By hypothesis of induction, the independence conditions between each two ready nodes in R_{old} are satisfied in c'. If (a) a goal initiation step is then performed, a new (possibly empty) set of ready goals R_{new} will be found. By definition of the model, for each ready goal $g_j \in R_{new}$ and each goal g_i, $i < j$, $i\_cond(g_i, g_j)$ is satisfied for c'. Thus, the independence conditions between each two goals in R_{new} are satisfied for c'. Furthermore, since for every $g_i \in R_{old}$ and every $g_j \in R_{new}$ we have that $i < j$, then every $i\_cond(g_i, g_j)$ is also satisfied in c'. Thus, all independence conditions between each two ready goals in $R_{old} \cup R_{new}$ are satisfied for c'. If (b) a node removal step is performed, then the answers corresponding to some goals in R_{old} which have finished are added to c' obtaining the store c'', and the associated ready nodes eliminated. By Corollary 3.1, the independence conditions between each two ready goals which have not been eliminated are still satisfied in c''. $\square$

We can now prove the main correctness result.

Theorem 3.1 Correctness of the CDG operational semantics. Consider the CDG obtained by applying an a priori, reasonably general, grouping notion of independence to the sequence $g_1 \dots g_n$. Any execution obtained by applying the CDG operational semantics to the graph with initial store c is correct and efficient w.r.t. the sequential execution of $\langle g_1 \dots g_n, c\rangle$.

PROOF. Let us prove it by induction on the number of goals in the sequence $g_1 \cdots g_n$:

- Base case: By definition of the model g_1 is ready and will be executed in store c. This execution is obviously identical to the sequential one.

- Induction step: assume that the CDG operational semantics is correct for the sequence $g_1 \dots g_{n-1}$. Let c' be the store in which goal g_n is executed, according to the CDG operational semantics. By definition of the model, g_n has become ready in c' and thus all goals in the sequence $g_1 \dots g_{n-1}$ must be either already initiated, or ready to be initiated in c'. Therefore, the execution of goals in the sequence $g_1 \dots g_{n-1}$ cannot be affected by the execution of g_n and thus, by induction hypothesis, it is identical to the sequential execution. By Lemma 3.2 the set of independence conditions between each two ready goals is satisfied in c'. Since the set of conditions is grouping, by definition of independence, the execution of $\langle g_n, c' \rangle$ is identical to that of $\langle g_n, s' \rangle$ where s' is obtained from the execution in c' of any goal in $\{g_1, \ldots, g_{n-1}\}$ which have not finished yet. Since c' has been obtained by adding to c the answers to the goals in $\{g_1, \ldots, g_{n-1}\}$ which have already finished, and we have already proved that such answers are identical to those in the sequential execution, s' is equivalent to s and the induction is proved. $\square$

The CDG model is "maximal" in the sense that goals are run in parallel *as soon as* they become independent. However, this requirement can be dropped, allowing a more general model in which ready goals are ensured to be independent if run in parallel, but they are not actually required to be run in parallel. In fact, any possible correct goal–level parallel (or even sequential) execution is allowed. In particular, given a non–simplified graph, a parallel expression simply corresponds to a particular execution of the graph in the above model: that which is obtained following the particular heuristic of the annotator being used to build the parallel expression.

Proving the correctness of a non-maximal model is straightforward since, given the above two lemmas, and the fact that they hold even if the model is not maximal, we can conclude that, once a goal g has been determined ready for store c', its execution in c' is identical to its execution in any subsequent store obtained by the model while g is still ready.

Note that CDGs can also represent &-Prolog clauses that are already annotated. Since any goals joined by "`&`" can be run in parallel, the *i_cond* for these is *true*, and therefore no edge exists in the corresponding CDG. If the parallel expressions are embedded in either a CGE or an if-then-else, the *i_cond* labeling the corresponding edge is precisely the condition in that conditional structure. It is easy to see that such a CDG is equivalent to the (parallel) operational semantics of the given clause. Also, consider a CDG for a given clause, and another one which has either more edges or larger labels in some edges (a label l' is larger than another one l, $l \leq l'$, iff $l' \rightarrow l$). We call such a CDG a super–CDG of the original one. It is obvious that super–CDGs are correct, as long as the original ones are.

Definition 3.3. [Super–CDG] The CDG (V, E') is a super–CDG of the CDG (V, E) iff (1) all edges in E are in E', modulo labels, and (2) for all edges in E with label l the corresponding edge in E' with label l' is such that $l \leq l'$.

3.2. Dealing with Non–pure Features and Built-ins

It must be taken into account that, in general, side–effects cannot be allowed to execute freely in parallel with other goals. In order to avoid their parallelization, the annotation can use the information derived by an analyzer which propagates the side–effect characteristic of built–ins yielding side–effect procedures (see e.g. [57]). Quite powerful solutions exist for dealing with side–effect built–ins and procedures (e.g. [18, 57, 36]). In our framework, there is an elegant solution which can be defined in terms also of a notion of independence. Note that some side–effects are themselves *independent* (for example, writing to different files) and could be considered for parallel execution. A suitable analysis for these cases, as well as defining a set of conditions which allow capturing them, will make our framework readily applicable also to the parallelization of side–effects. For simplicity, however, the (instances of the) algorithms we have studied sequentialize side–effects. Sequentialization can be achieved by slightly modifying the building process of the CDG associated to a clause, so that every edge connecting a side–effect literal is labeled with $false$.

Definition 3.4. [Annotation front–end] The CDG (V, E) corresponding to a given sequence of literals $g_1 \dots g_n$ is given by $V = \{g_1, \dots, g_n\}$, and $E = \{(g_i, g_j, label(g_i, g_j)) \mid \{g_i, g_j\} \subseteq V,\ prec(g_i, g_j),\ label(g_i, g_j) \neq \emptyset\}$, and
$$label(g_i, g_j) = \begin{cases} false & \text{if } g_i \text{ or } g_j \text{ is a goal with side–effects} \\ i\_cond(g_i, g_j) & \text{otherwise} \end{cases}$$

In the following we will denote by $cdg(Lit)$ the function which computes the CDG corresponding to the sequence of literals Lit according to the above definition. Note that the above definition is just a modification of Definition 1 for the case in which side-effects are allowed.

Some limited knowledge regarding the granularity of the goals, in particular the built–ins, is used in the parallelization task. Built-ins which are known to have enough size to be worth forking in parallel are considered for parallelization. Those which are known to be "small" are not. Meta–calls are sequentialized unless the called goal is available in the program text or their independence can be otherwise determined.

3.3. Linearization of a Dependency Graph

The second step of the transformation will compile a CDG into a linear expression which will then be used as the corresponding restricted &-Prolog clause body. In doing this, dependencies represented in the CDG must be satisfied. This step is in general non–deterministic: several different annotations are possible. Given a clause, its CDG is deterministically built, and from it, the corresponding &-Prolog clause is reconstructed. Different heuristic algorithms implement different strategies to select among all possible parallel expressions for a given clause.

The back–end of the transformation is parameterized by a function *exp*. The function *exp* can be instantiated to a particular algorithm for annotation, such as one of those described in the following sections.

Definition 3.5. [Annotation] Given a program clause C, an annotation of it yields an &-Prolog clause $C' = annotate(C)$, where $annotate(h) = h$, $annotate(h\text{:- } g) = h\text{:- } g$, and $annotate(h\text{:- } g_1, \dots, g_n) = h\text{:- } exp(cdg(\langle g_1, \dots, g_n \rangle))$.

The algorithms we present are focussed on preserving all the available μ-parallelism in the input graph. It is clear that in order to achieve this, a goal should be initiated as soon as all goals on which it depends have finished executing. All dependencies are captured in the dependency graphs, therefore a desirable objective in the annotation process would be to linearize a graph in such a way that no independent goals are executed sequentially. Graphs which can be linearized without loss of μ-parallelism will be called μ-graphs. Algorithms which decide if this is possible for the case of UDGs and CDGs will also be presented.

3.4. The Role of Program Analysis

In general, the independence conditions (as defined by *i_cond*) are generated in such a way that for any substitution, if those conditions are satisfied, the literals are independent. However, when considering the literals in the context of a clause and within a program, the condition can be simplified since independence then only needs to be ensured for those substitutions that can appear during execution of that program. Furthermore, if the class of admissible queries to the program is specified in some way, then only the substitutions which might appear in the execution of those queries need be considered. This observation is the basis of the role of program analysis within the transformation process.

The independence conditions can thus be simplified, or eliminated altogether, by using compile–time information provided either by the user or by an analyzer. Labels in the CDG can be simplified based on this information: if a condition is ensured to succeed, it is removed from the label; if a condition is ensured to fail, the label can be reduced to $false$. If the label becomes the empty set (i.e. it is reduced to $true$), the edge can be removed. On the contrary, if it is reduced to $false$, the edge becomes unconditional.

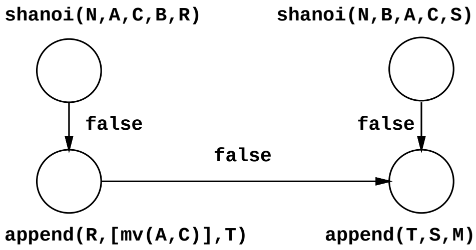
In the algorithms that we will propose, whether the CDG is already simplified or not is immaterial. Given a conditional graph for (part of) a clause C, its labels can be simplified based on the available information, prior to its linearization. However, it is worth noting that conditions can also be improved further in the back–end of the parallelization process: after a linear expression is built, the condition can possibly be reduced again. Both simplifications are parameterized by a function *improve*. Consider a propositional logic language in which the conditions can be expressed (e.g., the set *Cond* of Section 3). The compile-time information is translated into such language, capturing the subset of the compile-time information which is relevant for independence detection. In this context, we say that $improve(c, i) = c'$ if the simplification of condition c with (translated) information i yields the new condition c'. The only correctness requirement on the function *improve*, is that the propositional formula c can be proved from $c' \wedge i$ (see [9, 13]).

The updating of a set of edges of a CDG (V, E), identified by their source vertices $V_1 \subseteq V$, w.r.t. some information (condition) c is given by a function *update*:

$$update((V,E),V_1,c) = (V,(E \setminus \{(g_1,g,l)|g_1 \in V_1\}) \cup \\ \{(g_1,g,l') \mid g_1 \in V_1,\ (g_1,g,l) \in E,\ l' = improve(l,c) \neq true\})$$

We assume that the available information is valid before executing each goal g in the program, and denote it by $\mathcal{K}(g)$. Therefore, the correctness requirement over *improve* is also enough for *update*, as long as the information c is valid for all $g_1 \in V_1$. This requirement is also applicable to the initial simplification of a CDG (V, E) w.r.t. the information available, since it is just the result of applying $update((V, E), \{g\}, \mathcal{K}(g))$ for all $g \in V$.

Example 3.3. Consider the clause in Example 3.2 and its CDG. Local analysis can infer that `N` is a ground variable, and that `R`, `S`, and `T` are not aliased to any other variable until the point of their first occurrences. Moreover, global analysis can infer from other clauses in the program that `A`, `B`, and `C` are ground variables. The CDG after simplification is shown below.
Note that, of the original edges, three of them have been dropped, and the rest



have become unconditional. □

For non-grouping conditions the simplification process is different. Although conditions can be reduced to false before a particular annotation of the clause has been chosen, no condition can be reduced to true until the linear expression is built. This is because, in order to build the final conditions that will appear in the linear expression, we must first know which goals are possibly going to be run in parallel.

4. Unconditional Parallelism: Compiling UDGs

UDGs allow exploiting unconditional parallelism. Note that, even if the result of the front–end described before is not a UDG, it can be turned into a UDG on purpose with the aim of only exploiting unconditional parallelism. The rationale behind this is to avoid the overhead introduced by the run–time checks. We first turn our attention to this class of CDGs. We will first describe an algorithm which checks if a UDG is a μ-graph. Then, for UDGs which are known to be μ-graphs, we define an algorithm to actually do the compilation, which we call the UDG algorithm. Finally, for UDGs which are not μ-graphs we also present extensions to the UDG algorithm which allow to compile them. For clarity, the label $false$ present in all edges of a UDG will be dropped, all edges thus now being unlabeled.

4.1. Deciding whether a UDG is a μ-graph

The basic idea behind the decision algorithm is as follows. Let UDG $G = (V, E)$ be closed under transitivity (i.e. $E = E^*$), let P be the set of ready nodes in G, and $Q = V \setminus P$. Investigate whether the subgoals in Q can be executed in parallel with or should sequentially follow the subgoals in P. As a result, the set Q can be partitioned so that for each partition there is only a subset of P on which the elements of the partition depend. This induces another (pseudo-)partition in P: that of the corresponding subsets (which may not be disjoint). For the UDG to be a μ-graph, these sets have to be either pairwise disjoint or subsets of one another. If this is so, check if the sub–UDGs induced by the partition of Q satisfy these conditions again. All the sub–UDGs should satisfy these conditions so that the given UDG is a μ-graph. Note that the following results hold for transitively closed UDGs.

We first define the partitioning of the UDG, and discuss its characteristics. Let P be the set of ready literals computed by the function $ready(G) = \{p \in V \mid \forall x \in V\ (x, p) \notin E\}$. Consider the set $Q = V \setminus P$ of dependent goals. For each $q_i \in Q$, the set $\mathcal{E}(q_i) = \{p \in P | (p, q_i) \in E)\}$ is not empty. Let $Cover(P) = \{\mathcal{E}(q_i) \in 2^P | q_i \in Q\} = \{P_1, \ldots, P_n\}$, i.e., there is at least one literal in Q for each of these

P_i which depends on all elements of P_i. These literals are grouped together so that $\forall P_i \in Cover(P)$, $Dep(P_i) = \{q \in Q | \mathcal{E}(q) = P_i\}$, i.e. $Dep(P_i)$ is the set of all vertices in Q that *must* wait for *all* vertices in P_i (and *only* those, out of the vertices in P) to finish executing before their execution can be initiated.

The following results hold directly from the definition of $Cover(P)$ and $Dep(P_i)$ for $P_i \in Cover(P)$. They are instrumental in determining if the given UDG is a μ-graph.

Lemma 4.1. For any P_i, P_j in $Cover(P)$, $(i \neq j \Rightarrow Dep(P_i) \cap Dep(P_j) = \emptyset)$.

PROOF. Consider $\{P_i, P_j\} \subseteq Cover(P)$ s.t. $Dep(P_i) \cap Dep(P_j) \neq \emptyset$. This means that $\exists q \in Q$, $q \in Dep(P_i) \cap Dep(P_j)$, and therefore $\mathcal{E}(q) = P_i$ and $\mathcal{E}(q) = P_j$. Hence, $P_i = P_j$, which is a contradiction. □

Lemma 4.2. For each non–intersecting pair of sets P_i, P_j in $Cover(P)$, there are no edges between a vertex in $Dep(P_i)$ and a vertex in $Dep(P_j)$.

PROOF. Consider two disjoint sets P_i, P_j in $Cover(P)$, and assume that there exist two vertices $u \in Dep(P_i)$, $v \in Dep(P_j)$ s.t. $(u, v) \in E$. Since there exists $w \in P_i$ s.t. $(w, u) \in E$, then by virtue of transitivity, $(w, v) \in E$, and therefore $v \in Dep(P_i)$. But this contradicts the fact that $Dep(P_i) \cap Dep(P_j) = \emptyset$. □

In this context, no loss of μ-parallelism can occur when converting the graph into a linear (parallel) expression, if and only if the following conditions hold. Basically, for each $P_i \in Cover(P)$, the linear expression should satisfy the following requirements:

- There should be "&" operators between all the elements of P_i so that they can be run in parallel.

  Additionally, all elements of P should execute in parallel as well.

- The subexpressions involving elements of $Dep(P_i)$ should sequentially follow the subexpressions involving elements of P_i, and not of $P \setminus P_i$.

  Also, they should sequentially follow the subexpressions of $Dep(P_j)$ for each $P_j \subset P_i$.

Let us now informally discuss how to determine if a UDG satisfies these requirements. From the definition of the elements of $Cover(P)$, it always holds that for all P_1, P_2 in $Cover(P)$, $P_1 \neq P_2$ and either:

1. $P_1 \cap P_2 = \emptyset$, or
2. $P_1 \cap P_2 = P_1$, or $P_1 \cap P_2 = P_2$, or
3. $P_1 \cap P_2 = P$ s.t. $P \neq \emptyset, P \neq P_1, P \neq P_2$

Consider a UDG with $Cover(P) = \{P_1, P_2\}$. If these sets are in the first case, it is obvious that the subexpressions for $P_1 \cup Dep(P_1)$ and $P_2 \cup Dep(P_2)$ can be parallelized, thanks to the above two lemmas. No loss of μ-parallelism occurs in this case if it does not occur for the two subexpressions, i.e. if the UDGs for $Dep(P_1)$ and $Dep(P_2)$ are also μ-graphs. In the second case, let $P_1 \cap P_2 = P_1$. Since the elements of $Dep(P_1)$ should not wait for elements in $P_2 \setminus P_1$ and all elements in

$P_2 \setminus P_1$ must run in parallel with those in P_1, there should be a subexpression for $P_1 \cup Dep(P_1)$ in parallel with that for $P_2 \setminus P_1$. Call this (partial) expression Exp. Since the elements in $Dep(P_2)$ have to wait for all elements in P_2 (which includes P_1), the subexpression for $Dep(P_2)$ can only sequentially follow Exp. This does not lead to a loss of μ-parallelism iff each element of $Dep(P_2)$ depends on all elements of $Dep(P_1)$ (and, as before, the UDGs for $Dep(P_1)$ and $Dep(P_2)$ are also μ-graphs). In the third case, the expression should allow P, $P_2 \setminus P$, and $P_1 \setminus P$ to run in parallel. Also it should guarantee that $Dep(P_1)$ does not wait for $P_2 \setminus P$, and $Dep(P_2)$ does not wait for $P_1 \setminus P$. There is no expression which satisfies this.

We now formalize the above reasoning. First, if the UDG is a μ-graph, then the third case above cannot occur. This is stated in the following result.

Lemma 4.3. If the given UDG is a μ-graph, then for each P_i, P_j in $Cover(P)$, either

- *$P_i \cap P_j = \emptyset$, or*
- *$P_i \subset P_j$, or $P_j \subset P_i$.*

PROOF. Assume that there exist distinct sets P_i, P_j that do not satisfy the condition stated. Thus, they are not a subset one of the other, and $P_i \cap P_j \neq \emptyset$. If both P_i and P_j are singleton distinct sets, it can only be that $P_i \cap P_j = \emptyset$, which is a contradiction. If one of them is a singleton set, say P_i, but not the other one, since $P_i \cap P_j \neq \emptyset$, then $P_i \subset P_j$, which is also a contradiction. If none is a singleton set, let without loss of generality, $P_i \supseteq \{p_1, p_2\}$, $P_j \supseteq \{p_2, p_3\}$, $\mathcal{E}(q_1) = P_i$ and $\mathcal{E}(q_2) = P_j$. If the UDG is a μ-graph, its linear expression should allow (a) p_1, p_2, and p_3 to run in parallel, while (b) q_1 waits only for p_1 and p_2, and (c) q_2 only for p_2 and p_3. Therefore the expression should contain as subexpressions (a) $p_1 \& p_2 \& p_3$, (b) $\langle (p_1 \& p_2), q_1 \rangle$, and (c) $\langle (p_2 \& p_3), q_2 \rangle$, but no other one. To achieve this, consider introducing expressions (b) and (c) within (a). In (a) q_1 should sequentially follow $(p_1 \& p_2)$ (but not p_3) while q_2 sequentially follows $(p_2 \& p_3)$ (but not p_1). No parentherization (of any permutation) of (a) is possible which achieves this. $\square$

Also, if the UDG is a μ-graph, and some elements P_1 and P_2 of $Cover(P)$ are in the second case above, all elements of $Dep(P_2)$ must depend on all elements of $Dep(P_1)$, as stated below.

Lemma 4.4. If the given UDG is a μ-graph, then each pair P_i, P_j in $Cover(P)$, such that $P_i \subset P_j$, satisfies the following condition: $\forall uv(u \in Dep(P_i) \wedge v \in Dep(P_j)) \Rightarrow ((u,v) \in E)$.

PROOF. Assume that there exist P_i, P_j in $Cover(P)$ that violate the above condition. Without loss of generality, let $P_i \supseteq \{p_1\}$, $P_j \supseteq \{p_1, p_2\}$, $\mathcal{E}(q_1) = P_i$ and $\mathcal{E}(q_2) = P_j$. Since P_i, P_j do not satisfy the condition, the edge (q_1, q_2) does not exist in the given UDG, i.e. according to it, q_1 and q_2 can execute in parallel. Also, q_1 should sequentially follow only p_1, but q_2 both p_1 and p_2. This means that $\langle p_1, (q_1 \& q_2) \rangle$ and $\langle p_2, q_2 \rangle$ must be subexpressions of the resulting expression. There are only two ways in which the second one can be folded into the first one: either p_2 is attached to q_2, or to p_1. In the first case, consider the linear expression $\langle p_1, q_1 \& (p_2, q_2) \rangle$. It makes p_2 sequentially follow p_1, which contradicts the conditions in the UDG. In the second case, consider the linear expression $\langle p_1 \& p_2, q_1 \& q_2 \rangle$. It makes q_1 sequentially follow p_2, which is also in contradiction. □

Finally, the above conditions are not only necessary, but also sufficient for a UDG to be a μ-graph, as stated below.

Lemma 4.5. If the conditions in lemmas 4.3 and 4.4 hold for a given UDG, and its sub–UDGs, then it is a μ-graph.

PROOF. The result is shown to hold by constructing the linear expression. Consider the UDG $G = (V, E)$ with associated P and Q. From Lemma 4.3, every two sets in $Cover(P)$ are either disjoint or one a subset of the other. Therefore we can partition $Cover(P)$ into maximal subsets, each one containing the elements of $Cover(P)$ which are not disjoint:

$$\begin{aligned} Partition(Cover(P)) = \{\ Pt \subseteq Cover(P) \mid \forall P_i \in Pt \text{ it holds that}\\ \forall P_k \in Cover(P)\ (P_k \notin Pt \rightarrow P_k \cap P_i = \emptyset) \text{ and}\\ \forall P_j \in Pt\ (j \neq i \rightarrow P_j \subset P_i \vee P_j \subset P_i \vee \exists P_k \in Pt\ (P_i \cup P_j \subseteq P_k))\ \} \end{aligned}$$

For every $Pt \in Partition(Cover(P))$, let $Dep(Pt) = \bigcup_{P \in Pt} Dep(P)$. Since for any two elements $Pt_1, Pt_2 \in Partition(Cover(P))$, $Dep(Pt_1) \cap Dep(Pt_2) = \emptyset$, by Lemma 4.2 no dependencies exist for $Dep(Pt_1)$ on $Dep(Pt_2)$. Let $Partition(Cover(P)) = \{Pt_1, \cdots Pt_n\}$. Then, it is correct to convert G into the linear expression:

$$\langle\ exp(G|_{Pt_1 \cup Dep(Pt_1)})\ \&\ \cdots\ \&\ exp(G|_{Pt_n \cup Dep(Pt_n)})\ \rangle$$

and no loss of μ-parallelism occurs if no loss occurs for each $exp(G|_{Pt_i \cup Dep(Pt_i)})$. Now, for each $Pt = \{P_1, \ldots, P_m\}$ in $Partition(Cover(P))$, either (a) $P_1 \subset \ldots \subset P_m$ or (b) $\exists P_k \in Pt$ such that $\forall P_j \in Pt, j \neq k, P_j \subset P_k$. In case (a), by Lemma 4.4, it is correct to convert $G|_{Pt_i \cup Dep(Pt_i)}$ into the linear expression:

$$\langle\ (\cdots (\ \&_{p \in P_1}\ p,\ exp(G|_{Dep(P_1)}))\ \&\ \cdots\ \&_{p \in (P_m \setminus P_{m-1})}\ p\ ),\ exp(G|_{Dep(P_m)})\ \rangle$$

and no loss of μ-parallelism occurs. In case (b), let P_m be the superset (i.e., we, partially, order Pt by inclusion). Let also $\overline{P} = \bigcup_{i \in [1,m-1]} P_i$ and $\overline{D} = \bigcup_{i \in [1,m-1]} Dep(P_i)$. Then the linear expression:

$$\langle\ exp(G|_{\overline{P} \cup \overline{D}})\ \&_{p \in (P_m \setminus \overline{P})}\ p,\ exp(G|_{Dep(P_m)})\ \rangle$$

is correct, and no loss of μ-parallelism occurs if no loss occurs for $exp(G|_{\overline{P} \cup \overline{D}})$. The same reasoning applies to the induced sub–UDGs, which are guaranteed to at some point be partitioned in sets which satisfy case (a) above. □

The compilability decision algorithm recursively checks the conditions in lemmas 4.3 and 4.4 for the subgraphs induced by the partitions in $Cover(P)$.

Algorithm 4.1 Checking UDG compilability. The algorithm to check if a given UDG $G = (V, E)$ is a μ-graph is as follows:

```
function udg_is_μ-graph(G):  boolean
begin
  Let P = ready(G) and Q = V \ P;
  If Q = ∅ then return true;
```

```
  Let Cover = Cover(P) = {P_1, ..., P_n};
  For i := 1 to n-1 do
    For j := i+1 to n do
      If P_i ∩ P_j = P s.t. P ≠ ∅, P ≠ P_i, P ≠ P_j return false;
      If ∃u ∈ Dep(P_i) ∃v ∈ Dep(P_j) (u,v) ∉ E return false;
    od;
  od;
  Answer := true;
  i := 1;
  Repeat
    Answer := Answer AND udg_is_μ-graph(G|Dep(P_i));
    i := i+1;
  until (Answer = false) OR (i > n);
  return Answer;
end.
```

Theorem 4.1 Correctness of udg_is_μ-graph. The algorithm returns the answer true iff the given UDG is a μ-graph and the answer false iff it is not.

PROOF. If the algorithm returns $true$, then conditions in lemmas 4.3 and 4.4 hold for the UDG and its sub–UDGs. Then from Lemma 4.5, it is a μ-graph. If it returns $false$, then some condition in lemma 4.3 or 4.4 does not hold, and hence the UDG is not a μ-graph. □

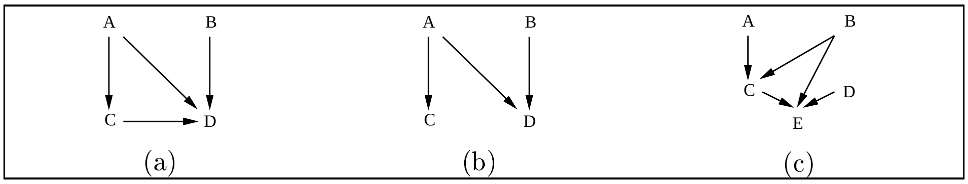


FIGURE 4.1. Example UDGs.

Example 4.1. Consider the UDG shown in Figure 4.1(a). We have $V = \{A, B, C, D\}$ and $P = \{A, B\}$. Hence, $Q = \{C, D\}$, with $\mathcal{E}(C) = \{A\}$, and $\mathcal{E}(D) = \{A, B\}$. Hence $Cover(P) = \{\{A\}, \{A, B\}\}$. Since $\mathcal{E}(C) \subset \mathcal{E}(D)$ and (C, D) is an edge of the given UDG, the conditions in lemmas 4.3 and 4.4 are satisfied.

We remove A, B and their edges from the graph. We have now $V' = \{C, D\}$ and the edge (C, D), with $P' = \{C\}$, and $Q' = \{D\}$, and $\mathcal{E}(D) = \{C\}$. Hence, $Cover(P') = \{\{C\}\}$. Since $Cover(P')$ is a singleton set, the conditions in the lemmas are trivially satisfied. We remove C and its edge from the graph. Now $V'' = P'' = \{D\}$, and $Q'' = \emptyset$, and the UDG is found to be a μ-graph. The corresponding expression will be given in Example 4.4. □

Example 4.2. Consider now the UDG in Figure 4.1(b), similar to the previous one. We have $V = \{A, B, C, D\}$ and $P = \{A, B\}$. Hence, $Q = \{C, D\}$, with $\mathcal{E}(C) = \{A\}$ and $\mathcal{E}(D) = \{A, B\}$. Hence, $Cover(P) = \{\{A\}, \{A, B\}\}$.

$Cover(P)$ satisfies the condition in Lemma 4.3. We have $Dep(\{A\}) = \{C, D\}$ and $Dep(\{A, B\}) = \{D\}$. Since $\{A\} \subset \{A, B\}$, and $C \in Dep(\{A\})$ and $D \in Dep(\{A, B\})$, but (C, D) is not an edge in the given graph, the condition in Lemma 4.4 is violated. Hence, this UDG cannot be a μ-graph. □

Example 4.3. Finally, consider the UDG in Figure 4.1(c). We have $V = \{A, B, C, D, E\}$ and $P = \{A, B, D\}$. Hence, $Q = \{C, E\}$. Also, $\mathcal{E}(C) = \{A, B\}$, and $\mathcal{E}(E) = \{B, D\}$. Hence $Cover(P) = \{\{A, B\}, \{B, D\}\}$. The elements in $Cover(P)$ do not satisfy the condition in Lemma 4.3. Therefore, this UDG cannot be a μ-graph. □

4.2. The UDG Algorithm for μ-graphs

This algorithm compiles a UDG which is a μ-graph, into a linear expression in such a way that all the parallelism present in the UDG is preserved. To do this, we will just follow the steps indicated by the proof of Lemma 4.5 and make use of the function $Partition$ defined within such proof.

Algorithm 4.2 UDG annotation. The expression built by the UDG algorithm from a transitively closed UDG $G = (V, E)$ which is μ-graphs, is given by $exp(G)$ as follows.

function $exp_{UDG}(G)$: expression
begin
  Let $P = ready(G)$ and $Q = V \setminus P$;
  If $Q = \emptyset$ then return $\langle\ \&_{p \in P}\ p\ \rangle$;
  Let $Partition = Partition(Cover(P)) = \{Part_1, \ldots, Part_n\}$;
  For i := 1 to n do
    Let $Part_i = \{P_1, \cdots, P_m\}$ s.t. $\forall P_k, P_l \in Part_i\ (P_k \subset P_l \rightarrow k < l)$;
    If $P_1 \subset \ldots \subset P_m$ then
      $Answer_i$:= $\langle\ \&_{p \in P_1}\ p,\ exp_{UDG}(G|_{Dep(P_1)})\ \rangle$;
      For j := 2 to m do
        $Answer_i$:= $\langle\ Answer_i\ \ \&_{p \in (P_j \setminus P_{j-1})}\ p,\ exp_{UDG}(G|_{Dep(P_j)})\ \rangle$;
      od;
    else
      Let $\overline{P} = \bigcup_{i \in \{1..m-1\}} P_i$ and $\overline{D} = \bigcup_{i \in \{1..m-1\}} Dep(P_i)$;
      $Answer_i$:= $\langle\ exp_{UDG}(G|_{\overline{P} \cup \overline{D}})\ \&_{p \in (P_m \setminus \overline{P})}\ p,\ exp_{UDG}(G|_{Dep(P_m)})\ \rangle$;
    fi;
  od;
  return $\langle\ \&_{i \in [1,n]}\ Answer_i\ \rangle$;
end.

Theorem 4.2 Correctness of UDG annotations. The execution of the expressions obtained by the UDG algorithm is correct w.r.t. their sequential semantics.

Proof. We only need to prove that the UDG for the obtained expression is a super–UDG of the original one. The proof of this is very similar to that of Lemma 4.5, since the algorithm follows exactly the construction steps in that proof. Since the linearizations at each step are perfect, no additional edges exist in the resulting

UDG. In fact, the original UDG is exactly preserved. □

Example 4.4. Consider a clause `h:- p(X), q(Y), r(X), s(X,Y)` whose UDG corresponds to that of Example 4.1 and Figure 4.1(a), with $A =$ `p(X)`, $B =$ `q(Y)`, $C =$ `r(X)`, and $D =$ `s(X,Y)`. There are dependencies for `C` on `A`, and for `D` on `A`, `C`, and `B`. There is no dependency for `B` on `A`. Algorithm 4.2 will compile this clause as follows. We have $Cover(P) = \{\{\mathtt{A}\}, \{\mathtt{A,B}\}\}$, with $Dep(P_1) = Dep(\{\mathtt{A}\}) = \{\mathtt{C}\}$ and $Dep(P_2) = Dep(\{\mathtt{A,B}\}) = \{\mathtt{D}\}$. $Partition = \{Cover(P)\}$, and the outer loop of the algorithm is performed only once for this single element. Since $\{\mathtt{A}\} \subset \{\mathtt{A,B}\}$, $Answer_1$ is first initialised to $e_1 = \langle\ \mathtt{A},\ exp_{UDG}(G|_{Dep(P_1)})\ \rangle$. The inner loop is also performed only once, for $\{\mathtt{A,B}\}$, giving $Answer_1 = \langle\ e1\ \mathtt{\&}\ \mathtt{B},\ exp_{UDG}(G|_{Dep(P_2)})\ \rangle$. Clearly, $exp_{UDG}(G|_{Dep(P_1)}) = \mathtt{C}$, and $exp_{UDG}(G|_{Dep(P_2)}) = \mathtt{D}$. Therefore, the final expression is:

```
h :- ( p(X), r(X) ) & q(Y), s(X,Y).
```
□

4.3. Extensions to the UDG Algorithm

In real program clauses, usually bodies are transformed to UDGs which are not transitively closed. Although the definition of a CDG implies that it will be transitively closed (because of the *prec* relation), if the conditions labeling its edges can be found to be true, by simplification based on available information, these labels will become empty sets, and the corresponding edges would then be dropped in the corresponding UDG. The more accurate the information is, the more this case can happen. We will consider extending the algorithm to these cases.

Algorithm 4.2 finds the best linearization of the dependency graph in such a way that no loss of μ-parallelism occurs. For this to be possible, we have seen that the body of a given clause must satisfy certain conditions, which restrict the class of UDGs which can be handled by that algorithm. In order to extend the algorithm to deal with all possible UDGs, the following possible graph linearizations have to be considered (as subexpressions of the final result $exp(G)$):
$\forall P_1 P_2 \in Cover(P)$, if

1. $P_1 \cap P_2 = \emptyset$

$$exp(G|_{P_1 \cup Dep(P_1)})\ \mathtt{\&}\ exp(G|_{P_2 \cup Dep(P_2)})$$

2. $P_1 \cap P_2 = P_1$

   (a) $\forall q_1 \in Dep(P_1) \forall q_2 \in Dep(P_2)((q_1, q_2) \in E)$
   $$exp(G|_{P_1 \cup Dep(P_1)})\ \mathtt{\&}\ exp(G|_{P_2 \setminus P_1}),\ exp(G|_{Dep(P_2)})$$

   (b) $\forall q_1 \in Dep(P_1) \forall q_2 \in Dep(P_2)((q_1, q_2) \notin E)$

      i. at the loss of parallelism between $Dep(P_1)$ and $Dep(P_2)$
      $$exp(G|_{P_1 \cup Dep(P_1)})\ \mathtt{\&}\ exp(G|_{P_2 \setminus P_1}),\ exp(G|_{Dep(P_2)})$$

      ii. at the loss of parallelism between $Dep(P_1)$ and $P_2 \setminus P_1$
      $$exp(G|_{P_2}),\ exp(G|_{Dep(P_1)})\ \mathtt{\&}\ exp(G|_{Dep(P_2)})$$

(c) $\exists q_1 \in Dep(P_1) \exists q_2 \in Dep(P_2)((q_1, q_2) \in E)$ but 2a does not hold

i. at the loss of parallelism between $q_2 \in Dep(P_2)$ and $q_1 \in Dep(P_1)$ s.t. $(q_1, q_2) \notin E$

$$exp(G|_{P_1 \cup Dep(P_1)}) \ \& \ exp(G|_{P_2 \setminus P_1}),\ exp(G|_{Dep(P_2)})$$

ii. at the loss of parallelism between $Dep(P_1)$ and $P_2 \setminus P_1$

$$exp(G|_{P_2}),\ exp(G|_{Dep(P_1) \cup Dep(P_2)})$$

iii. for $Q_{12} = \{q_1 \in Dep(P_1) \mid \exists q_2 \in Dep(P_2)(q_1, q_2) \in E\}$ and $Q_{11} = Dep(P_1) \setminus Q_{12}$

$$exp(G|_{P_1 \cup Q_{12}}) \ \& \ exp(G|_{P_2 \setminus P_1}),\ exp(G|_{Q_{11}}) \ \& \ exp(G|_{Dep(P_2)})$$

at the loss of parallelism between Q_{11} and $P_2 \setminus P_1$ and also between Q_{12} and $q_2 \in Dep(P_2)$ s.t. $\forall q_1 \in Dep(P_1)((q_1, q_2) \notin E)$

3. $P_1 \cap P_2 = P \mid P \neq \emptyset, P \neq P_1, P \neq P_2$

$$exp(G|_{P_1 \cup P_2}),\ exp(G|_{Dep(P_1) \cup Dep(P_2)})$$

at the loss of parallelism between $q_2 \in Dep(P_2)$ and $p_1 \in P_1 \setminus P$ and also $q_1 \in Dep(P_1)$ and $p_2 \in P_2 \setminus P$

Algorithm 4.2 deals with cases 1 and 2a. The natural extension of the algorithm to be able to deal with the whole of Case 2 is to make it force the assumption that the required condition in Case 2a holds and let it behave as in this case. This leads the extended algorithm to follow the strategy of cases 2(b)i and 2(c)i. To make the extension complete, it also has to deal with Case 3, for which the elements P_1 and P_2 involved are considered as a single one and replaced by $P_1 \cup P_2$. Note that each of these extensions implies a loss of parallelism.

The extension proposed is the simplest one that allows the UDG algorithm to deal with non-μ-graphs. However, there are other possibilities. It is interesting to reason about the execution cost of the parallel expressions that can be obtained. It then turns out that in general it may be more profitable to perform the extension in a different way. This can be seen with a simple experiment. Consider all the possible parallel expressions listed above for a situation like Case 2. Let us construct the minimum sets needed to cover all sub–cases in that case. These are $P_1 = \{p_1\}$, $P_2 = \{p_1, p_2\}$, $Dep(P_1) = \{q_1\}$ ($Dep(P_1) = \{q_{11}, q_{12}\}$ for Case 2c) and $Dep(P_2) = \{q_2\}$. The conditions in each sub–case of Case 2 then yield the following expressions:

2(b)i	(p_1, q_1) & p_2, q_2	2(c)i	$(p_1, q_{11}$ & $q_{12})$ & p_2, q_2
2(b)ii	p_1 & p_2, q_1 & q_2	2(c)ii	p_1 & p_2, q_{11} & (q_{12}, q_2)
		2(c)iii	(p_1, q_{12}) & p_2, q_{11} & q_2

We would like to evaluate the cost of each of these expressions. We assume that there is an upper bound on the execution cost (i.e. the granularity) of all literals. Since we are only interested in the relative computational cost of the goals (whatever metric is used to measure it), we just assign arbitrary units of “size” to each goal, from 1 to the upper bound. We then compute the cost for the expression for all possible combinations of costs of the single goals up to the upper bound. Note that the exact value of the upper bound is not important, but rather expresses the maximum difference in cost among the goals. Table 4.1 shows, for each parallel expression, the percentage of combinations where that expression gives the minimal cost. The percentage includes situations where more than one expression is best, and thus the total percentage can add up to more than 100%.

Max.G. Grain	% Best case 2(b)i	% Best case 2(b)ii
1	0	100
2	18	100
3	22	97
4	23	95
5	24	94
6	24	93
7	24	92
8	24	92
9	24	91
10	24	91

Max.G. Grain	% Best case 2(c)i	% Best case 2(c)ii	% Best case 2(c)iii
1	100	100	100
2	78	78	78
3	72	71	70
4	70	67	66
5	68	65	64
6	67	63	62
7	66	62	61
8	65	61	60
9	65	60	59
10	65	60	59

TABLE 4.1. Performance test for possible parallel expressions.

From the table, our previous choice of strategy, 2(c)i, appears as the best parallelization strategy in Case 2c. However, in Case 2b, the second option, 2(b)ii, instead of the one we chose previously, 2(b)i, behaves best. This is due to the fact that this strategy performs a better load balancing of parallel tasks with goals which are already balanced (i.e. have almost the same granularity, as with maximum grain of 1 or 2) or for which the differences in grain size are not high. When a bigger difference is allowed (increasing the maximum permitted goal cost) the average efficiency of 2(b)ii lowers a bit, while that of 2(b)i progressively behaves better. Therefore, the best parallelization strategies in order to extend the UDG algorithm seem to be those of cases 2(b)ii and 2(c)i.

It is worth noting that this result points out the importance of having granularity information on the literals being annotated, so that the annotators could take granularity into consideration in the load balancing algorithms. Unfortunately, having good measures for the granularity of literals is a difficult task.[4] Dealing fully with annotation in the presence of granularity information is beyond the scope of this paper: in the absence of information on granularity, the parallelization strategy of 2(b)ii and 2(c)i should be pursued.

4.4. The UDG Algorithm for non-μ-graphs

We will now propose an algorithm along the lines discussed above. Note however that, even if we assume the loss of parallelism which occurs when considering sets of $Cover(P)$ pairwise, there can be additional losses when coupling expressions for any two sets together. A radical solution to this is to consider literals pairwise, instead of in sets. Dependencies are considered in a literal–to–literal fashion, incrementally

[4]Although quite interesting progress has been made recently — see [25, 26, 84] and their references.

creating the expression from the one already generated by introducing a new literal each time (an algorithm in this style is presented in [14]). We take a similar solution, but applied to the sets of $Cover(P)$. We first order $Cover(P)$ so as to consider each set after its subsets have been considered, in order to reduce the loss of parallelism. I.e. if $Cover(P) \supseteq \{P_1, P_2, P_3\}$ s.t. $P_1 \subset P_2 \subset P_3$, we consider an expression for P_1 and P_2 and apply the corresponding strategy; afterwards, we consider an expression by adding P_3 to the previous one and building it by applying the strategy based on the relations among P_2 and P_3, thus ignoring those among P_1 and P_3. To guarantee that we will only find sets which are either disjoint or subsets of one another, we will modify $Cover(P)$ in order to first deal with Case 3. The modification consists in recursively selecting two sets P_i, P_j in $Cover(P)$ which are not disjoint nor subsets of one another, and substituting them by their union, until no more such sets exists:

$$Modify(C) = \begin{cases} Modify(C \setminus \{P_i, P_j\} \cup \{P_i \cup P_j\}) & \\ \quad \text{if } \exists P_i, P_j \in C \ (P_i \cap P_j = P \rightarrow P \neq \emptyset \wedge P \neq P_i \wedge P \neq P_j) & \\ C & \text{otherwise} \end{cases}$$

Note that the resulting C' might depend on the order in which P_i and P_j are selected. For example, consider $C = \{P_1, P_2, P_3\}$ with $P_1 = \{p_1, p_2\}$, $P_2 = \{p_1, p_3\}$, $P_3 = \{p_2, p_3, p_4\}$. On the one hand, by selecting first P_2 and P_3 we obtain $C' = \{P_1, P_2 \cup P3\}$ which cannot be further modified. On the other hand, by selecting first P_1 and P_2 we obtain $C' = \{P_1 \cup P_2, P_3\}$ which must be further modified, resulting in $\{P_1 \cup P_2 \cup P_3\}$. This suggests, in view of the discussion of the previous section, that it is better to select the biggest P_i, P_j satisfying the condition in order to apply the modification. In our implementation, the elements of $Cover(P)$ are ordered by length, so that, in the previous example, the second alternative will be taken. Abstraction made of this issue, the algorithm is as follows.

Algorithm 4.3 Extended UDG annotation. The expression built by the extended UDG algorithm from a UDG $G = (V, E)$ is given by $exp(G)$ as follows.

```
function exp_UDG(G):  expression
begin
  Let P = ready(G) and Q = V \ P;
  If Q = ∅ then return ⟨ &_{p∈P} p ⟩;
  Let Partition = Partition(Modify(Cover(P))) = {Part_1, ..., Part_n};
  For i := 1 to n do
    Let Part_i = {P_1, ···, P_m}, m ≤ n, ∀P_k, P_l ∈ Part_i (P_k ⊂ P_l → k < l);
    If P_1 ⊂ ... ⊂ P_m then
      Answer_i := ⟨ &_{p∈P_1} p ⟩;
      Qs := Dep(P_1);
      For each j := 2 to m do
         If P_j and P_{j-1} are in Case 2a or 2c then
            Answer_i := ⟨ (Answer_i, exp_UDG(G|Qs)) &_{p∈(P_j\P_{j-1})} p ⟩;
            Qs := Dep(P_j);
         else
            Answer_i := ⟨ Answer_i &_{p∈(P_j\P_{j-1})} p ⟩;
            Qs := Qs∪Dep(P_j);
         fi;
      od;
```

```
        Answer_i := ⟨ Answer_i, exp_UDG(G|Qs) ⟩;
      else
        Let P̄ = ⋃_{i∈{1..m−1}} P_i and D̄ = ⋃_{i∈{1..m−1}} Dep(P_i);
        Answer_i := ⟨ exp_UDG(G|P̄∪D̄) &_{p∈(P_m\P̄)} p, exp_UDG(G|Dep(P_m)) ⟩;
      fi;
    od;
    return ⟨ &_{Part_i∈Partition} Answer_i ⟩;
end.
```

Theorem 4.3 Correctness of extended UDG annotations. The execution of the expressions obtained by the extended UDG algorithm is correct w.r.t. their sequential semantics.

PROOF. We only need to prove that the UDG for the obtained expression is a super–UDG of the original one. We first prove that it holds for the `If` statement for $Q = \emptyset$, then that it holds for the expressions in the `If` statement of the inner `For` loop, then that it holds for those in the `If` statement of the outer loop, and finally that it holds for that in the `return` statement.

When $Q = \emptyset$ no edges exist in the original UDG; hence, the expression $\langle\ \&_{p \in P}\ p\ \rangle$ has the same UDG as the original one. In the inner loop, note that `Qs` always contains elements of $Dep(P_k)$ for some P_k's s.t. $P_k \subset P_j$ for the current j. $Answer_i$ always contains an expression for such P_k's and possibly some of the corresponding $Dep(P_k)$'s. In the "then" part of the `If` statement $P_j \setminus P_{j-1}$ is allowed in parallel with the goal formed by the sequential execution of $Answer_i$ and an expression for `Qs`. No edges will exist in the corresponding UDG between $P_j \setminus P_{j-1}$ and the elements of $Answer_i$ and `Qs`, which is to say of P_k's and $Dep(P_k)$'s s.t. $P_k \subset P_j$. By construction of $Partition$ and the $Dep(P_k)$'s, no edges exist between such elements in the original UDG. For the same reason, the resulting UDG in the "else" part does not lose any edges in the original one, either. In the outer loop, in the "then" part, an expression for `Qs` sequentially follows $Answer_i$. This can only have the effect of adding edges which were not in the original UDG from elements of some P_j's and $Dep(P_j)$'s to some other $Dep(P_i)$'s s.t. $P_j \subset P_i$. In the "else" part, the only parallel expression is between $\overline{P} \cup \overline{D}$ and $P_m \setminus \overline{P}$. Since $P_m \supseteq \overline{P}$, and no edges originally exist between $Dep(P_i) \in \overline{D}$ and $P_m \setminus \overline{P}$, this does not violate any of the original edges, either. Finally, in the returned expression elements of each $Answer_i$ are allowed in parallel. Since the set $Partition$ is built in such a way that each of the elements of $Part_i$ is disjoint with each of the elements of $Part_j$, where $\{Part_i, Part_j\} \subseteq Partition$, no edges exist in the original UDG between elements of each of the $Answer_i$. Thus, the UDG of the resulting expression is a super–UDG of the original one. □

5. Conditional Parallelism: Compiling CDGs

Let us now turn our attention to the general case of CDGs. Since we now have conditions, the most ambitious strategy would be to try to exploit all the available μ-parallelism in each of the situations determined by the combination of the conditions. Basically, the idea is to convert the given CDG into a set of UDGs, each of which is a dependency graph corresponding to one combination of truth values on the conditions labeling the edges of the CDG. It is easy to prove that if there is at least one feasible UDG which is not a μ-graph, then the given CDG is not a μ-graph either. Also, if the given CDG is a μ-graph, then each feasible UDG must also be a μ-graph.

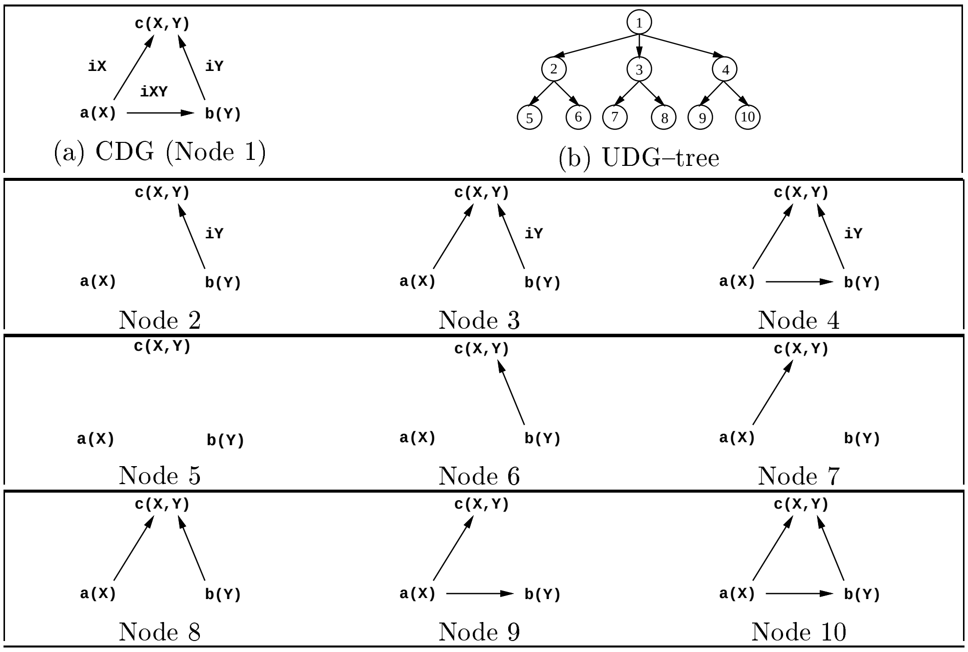


FIGURE 5.1. CDG and the UDG–tree for `a(X),b(Y),c(X,Y)`.

Example 5.1. Consider a clause whose body is `a(X),b(Y),c(X,Y)`. The CDG for this clause is given in Figure 5.1(a). Figure 5.1(b) shows the steps needed to derive the set of feasible UDGs (leaves of the tree) from the original CDG (root of the tree). The graphs corresponding to nodes from 2 to 10 are given in the same figure.

The combinations of the truth values of edges departing from the node `a(X)` yield three possibilities: $\{\mathtt{iX}, \neg\mathtt{iX} \wedge \mathtt{iXY}, \neg\mathtt{iX} \wedge \neg\mathtt{iXY}\}$ (note that `iX` implies `iXY`). Nodes 2, 3, and 4 are obtained by assuming one of the possibilities, respectively. Nodes 5 and 6, 7 and 8, and 9 and 10, are then obtained by applying the same process to the conditions labeling the edges departing from node `b(Y)` in graphs 2, 3, and 4, respectively.

Although all the UDGs (the graphs corresponding to nodes from 5 to 10) are μ-graphs, we cannot ensure that the initial CDG is a μ-graph. □

Since the algorithm informally described above is only complete, it has a limited interest. We will skip a formal definition of it and we will start with an annotation algorithm which deals with non–μ-graphs directly. Extensions to this algorithm are also discussed.

5.1. The CDG Algorithm

The CDG algorithm produces the linear expression of a given CDG $G = (V, E)$. Following the ideas mentioned before, it considers all possible states of computation which can occur w.r.t. the conditions in G, and annotates the body literals into the best parallel expressions achievable under such conditions. The algorithm starts with the same set P of independent literals as in UDGs. The main difference resides in that literals depending unconditionally on literals in P are not coupled to them (i.e. the close relation upon each P_i and corresponding $Dep(P_i)$ in the UDG algorithm is not followed here). This is because the CDG algorithm focuses on the conditional dependencies present in the graph, rather than on edges labeled $false$ (or unlabeled).

Consider the CDG $G = (V, E)$, with $P = ready(G)$, and $Q = V \setminus P$. We denote by $PConds(G, P, Q)$ the function which computes the sets of conditions (other than $false$) in labels of edges between literals in P and literals in Q, i.e., $PConds(G, P, Q) = \{l \in Cond \mid (p, x, l) \in E \wedge p \in P \wedge x \in Q \wedge l \neq false\}$. Also, we denote by $QConds(G, Q)$ the function which computes the sets of conditions in labels of edges among literals in Q, $QConds(G, Q) = \{l \in Cond \mid (x, y, l) \in E \wedge \{x, y\} \subseteq Q \wedge l \neq false\}$. The algorithm proceeds by incrementally building up the parallel expression $exp(G)$ as follows; let $P = \{p_1, \ldots, p_n\}$, $PConds = PConds(G, P, Q)$, $QConds = QCons(G, Q)$:

- if $Q = \emptyset$ then $exp(G) = \langle p_1 \ \texttt{\&} \ \ldots \ \texttt{\&} \ p_n \rangle$
- if $Q \neq \emptyset$, $PConds = QConds = \emptyset$ then $exp(G) = exp_{UDG}(G)$
- if $Q \neq \emptyset$, $PConds = \emptyset$, $QConds \neq \emptyset$ then $exp(G) = \langle p_1 \ \texttt{\&} \ \ldots \ \texttt{\&} \ p_n, \ exp(G|_Q) \rangle$
- if $Q \neq \emptyset$, $PConds \neq \emptyset$ then $exp(G)$ is recursively built up from the boolean combinations of the elements of $PConds$ as described below.

Let $BoolComb(Conds)$ be the function which returns the set of simplified boolean combinations of the conditions in $Conds$ which are different from $false$. Let $Bool = BoolComb(PConds)$. For each boolean combination $b \in Bool$ the graph G is updated as if the conditions in b hold by means of the function $update(G, P, b)$. Note that this is similar to the update performed in Example 5.1, but considering the set of edges with source in P. However, specialized versions of $update$ for the notion of independence under consideration can be defined. In [58] an instance of this function for the particular case of strict independence in the Herbrand domain is presented. The parallel expressions resulting from recursively applying the CDG algorithm after this updating are annotated as if-then-elses and combined in a simplified form.

Example 5.2. Consider a clause `p(W,X,Y,Z):- W is X+1,a(W),b(X,Y),c(Y,Z).` Given the information inferred from the built–in, its CDG corresponds to that of Node 2 of Figure 5.1 (modulo the arities of the predicates). The algorithm will consider all possible alternatives (nodes 5 and 6) and yield the following clause:

```
p(W,X,Y,Z) :- W is X+1,
              ( ground(Y) -> a(W) & b(X,Y) & c(Y,Z)
                           ; a(W) & (b(X,Y), c(Y,Z)) ).
```

whereas the UDG algorithm will yield:

```
p(W,X,Y,Z) :- W is X+1, a(W) & (b(X,Y), c(Y,Z)).
```

which is the worst case subexpression of the expression above. □

The simplified form of the resulting expression is formally defined by a function $simplify$. Let $Bool = \{b_1, \ldots, b_n\}$ be the result of the function $BoolComb(PConds)$ and $\{e_1, \ldots, e_n\}$ be their corresponding expressions, we define $simplify(b_1 \to e_1; \ldots; b_n \to e_n)$ as follows:

- If a partition of $Bool = \{b_{i_1}, \ldots, b_{i_M}\} \cup \{b_{j_i}, \ldots, b_{j_N}\}$, can be done such that for some $cond$, $\forall k \in [1, M]$ $(b_{i_k} = cond \wedge s_{i_k})$ and $\forall k \in [1, N]$ $(b_{j_k} = \neg cond \wedge s_{j_k})$, then $simplify(b_1 \to e_1; \ldots; b_n \to e_n) = \langle goal(cond) \to D1; D2\rangle$ where $D1 = simplify(s_{i_1} \to e_{i_1}; \ldots; s_{i_M} \to e_{i_M})$, and $D2 = simplify(s_{j_1} \to e_{j_1}; \ldots; s_{j_N} \to e_{j_N})$
- If such partition cannot be done, conditions are all atomic, and therefore $simplify(b_1 \to e_1; \ldots; b_n \to e_n) = \langle goal(b_1) \to e_1; \ldots; goal(b_n) \to e_n\rangle$

where $goal(cond)$ maps condition $cond$ into a suitable &-Prolog goal, after possibly simplifying it again with the same $improve$ function as in the simplification phase of the graph. This simplification must be done only based on the information valid for the leftmost literal of $e_1, \ldots, e_n$. Note that this literal is the same for all the e_i's, since they are obtained from the same graph in the following algorithm.

Algorithm 5.1 CDG annotation. The expression built by the CDG algorithm from a CDG G is given by $exp(G)$ as follows.

```
function exp_CDG(G):  expression
begin
  Let P = ready(G) and Q = V \ P;
  If Q = ∅ then return ⟨ &_{p∈P} p ⟩;
  Let PConds = PConds(G,P,Q) and QConds = QConds(G,Q);
  If PConds = QConds = ∅ then return exp_UDG(G);
  If PConds = ∅ then return ⟨ p_1 & ... & p_n, exp_CDG(G|_Q) ⟩;
  Let Bool = BoolComb(PConds) = {b_1,...,b_n};
  For i := 1 to n do
    e_i = exp_CDG(update(G,P,b_i))
  od;
  return simplify(b_1 → e_1;...;b_n → e_n);
end.
```

Theorem 5.1 Correctness of CDG annotations. The execution of the expressions obtained by the CDG algorithm is correct w.r.t. their sequential semantics.

Proof. We prove by induction that the CDG for the obtained expression is a super–CDG of the original one. First we reason for the base case. If $Q = \emptyset$ the original CDG has no edges, and the same happens for that of the expression returned: $\langle p_1$ & $\ldots$ & $p_n\rangle$. If $PConds = QConds = \emptyset$ then the original CDG has no conditional edges, and the graph is then a UDG. Correctness is guaranteed because the expression returned is $exp_{UDG}(G)$ which is correct by Theorem 4.2. Now we reason for the induction step. The induction hypothesis is that the recursive calls already return a correct expression (w.r.t. the graph with which the calls are made). If $PConds = \emptyset$ the expression returned is $\langle\, p_1$ `&` $\ldots$ `&` $p_n, exp_{CDG}(G|_Q)\,\rangle$. The subexpression for $P = \{p_1, \ldots, p_n\}$ has an associated CDG which is a μ-graph, since there are no edges between elements of P in the original CDG. The sequential conjunction respects the original edges from elements of P to elements of Q, and possibly adds more. Since the CDG $G|_Q$ is precisely the original one without the edges from elements of P to elements of Q, and the recursive call is correct by hypothesis, the CDG for this expression is a super–CDG of the original one. In the last case, consider expressions e_i and function $simplify$. The CDGs with which the recursive calls in the e_i's are made are sub–CDGs of the original one, in such a way that each one has been simplified w.r.t. some condition b_i. The results of the recursive calls are expressions whose CDGs are super–CDGs of each of these (by hypothesis). Since the effect of the function $simplify$ is to add b_i to the labels in all the edges of each of these CDGs, their labels can only be larger than those in the original subgraphs. Hence, the CDG of the resulting expression is also a super–CDG of the original one. □

5.2. *Variants to the CDG Algorithm*

The CDG algorithm seeks to obtain the best possible parallel expressions which can be generated on each of the different situations which may occur from the boolean combinations of conditions it considers. In doing this, it does not particularly focus on unconditional dependencies (as the UDG algorithm), rather it focuses instead on conditions which can allow independence of literals. Thus, in the third case of Algorithm 5.1 ($PConds = \emptyset$), an unconditional parallel expression is built for elements in P followed sequentially by another expression recursively computed for the rest Q of the literals. No consideration is given in this case to the unconditional dependencies which could occur from literals in Q on literals in P. Algorithm 4.2 for UDGs, on the other hand, does this, and groups literals depending unconditionally on those of P (i.e. $Dep(P_i)$ for $P_i \in Cover(P)$) together and with those on which they depend (i.e. each P_i), building an expression for the different groups of literals. A variant of the CDG algorithm is possible if the case for $PConds = \emptyset$ is omitted. Instead, when this case is detected, the sets P and Q should be computed again, as if the vertices in P and the edges with origin in them did not exist (but without deleting them). Unconditional dependencies will therefore persist, and will be taken care of by the UDG algorithm in a recursive call in which $PConds = QConds = \emptyset$ is detected. This variant will allow a one–to–one correspondence between both algorithms, so that the expressions built by Algorithm 4.2 will always be the worst case subexpression of those built by Algorithm 5.1. This happens in Example 5.2, but it is not so in general. E.g.:

Example 5.3. Consider the clause `h:- p(X),q(Y),r(X),s(X,Y)`. Algorithm 5.1 will work as follows. Since `X` has its first occurrence in `p(X)` and `Y` in `q(Y)`, `p(X)` and `q(Y)` are independent, and the dependencies of `r(X)` and `s(X,Y)` on `p(X)` and of `s(X,Y)` on `q(Y)` are unconditional. Thus, there is only one conditional dependency: for `s(X,Y)` on `r(X)`, labeled $indep(\mathtt{X},\mathtt{X})$. Therefore, $P = \{\mathtt{p(X)}, \mathtt{q(Y)}\}$ and $PConds = \emptyset$. The following expression is built: $\langle$ `p(X) & q(Y),` $exp_{CDG}(G|_{\{\mathtt{r(X)},\mathtt{s(X,Y)}\}}\ \rangle$, and the recursive call builds a CGE for the two goals involved. The resulting expression is:

```
h :- p(X) & q(Y), ground(X) => r(X) & s(X,Y).
```

which is very different from the one in Example 4.4 obtained by Algorithm 4.2:

```
h :- ( p(X), r(X) ) & q(Y), s(X,Y).
```

If Algorithm 5.1 is extended as mentioned, P and Q will be computed again when $PConds = \emptyset$ is found, giving a new $P = \{\texttt{r(X)}\}$ and $Q = \{\texttt{s(X,Y)}\}$. The boolean combinations of $indep(\texttt{X},\texttt{X})$ will be considered, but since $\neg indep(\texttt{X},\texttt{X})$ is known to hold at the neck of the clause, this one will be the only combination. The graph will be updated accordingly, giving a UDG, and thus Algorithm 4.2 will be called, resulting in the above unconditional expression. □

In less contrived cases, the resulting expressions of the extended CDG algorithm will give nested if-then-elses in which the final "else" case will always be an unconditional expression built by the UDG algorithm. Note however that the example shows why the conditions in the μ-checking algorithm for CDGs are not sufficient. The algorithm will return true for the clause in the example. However, the best linearization is that of the unconditional expression above, and in this expression there is no way to incorporate the check `ground(X)` between `r(X)` and `s(X,Y)`. Thus, if it were the case that `p(X)` made `X` ground, the available parallelism between these two goals would have been lost.

6. Compilation of CDGs made Practical

Algorithm 5.1 for compiling CDGs has the disadvantage of having exponential complexity. This suggests the need of more practical approaches that can either be used by themselves or serve as a "fall back" when the algorithm is faced with large inputs. Several more practical approaches are discussed in this section. First, an alternative algorithm (MEL) not necessarily based on graphs is presented. This algorithm exploits a very simple heuristic: partition the clause body at points in which parallel execution is not allowed. The aim is to find the longest parallel expression possible among those which are flat, i.e. such that nested subexpressions are not allowed. The resulting algorithm is quite simple, has polynomial complexity (quadratic, in fact), results in very simple parallel expressions, and, as we will show in Section 8, is quite effective. Secondly, and as further alternatives, we discuss two possible variants of algorithms for CDGs. These two methods are aimed at reducing the complexity of Algorithm 5.1 by seeking unconditional parallelism. Thus, they can be seen as ways to combine the CDG and UDG algorithms in order to obtain an algorithm for CDGs which has some of the good properties of the UDG algorithm (and thus they are called UCDGs algorithms). Both methods are parameterized by several functions whose definition depends on the kind of compile-time information available. In this sense, the algorithms can be considered more as skeletons of possible algorithms.

6.1. Non Graph–Based Compilation: the MEL Algorithm

The MEL (Maximum Expression Length) algorithm is based on a heuristic which tries to find out points in the body of a clause where it can be split into different expressions. One example of such a point, for the case of strict independence, is where a new variable appears. Consider a literal which has the first occurrence of a variable in a clause, and this variable is used as an argument of another literal to the right of the first one. The condition in strict independence which must hold for two literals which share variables establishes that these variables must be ground; obviously this is not the case for such two literals, and thus this is a point where it is not appropriate to annotate a parallel expression.

The heuristic can however be separated from the notions of independence used in parallelizing the programs. The heuristic can be read as "partition the body where a condition between two literals is first found to be false". In order to accommodate the MEL definition to this approach, it is necessary to define a framework for capturing when conditions can be turned to false for a particular concept of independence. This is done by the functions *i_cond* (and *label*) and *improve* of sections 3 and 3.4.[5]

The algorithm then proceeds in this manner from right to left, i.e. from the last literal in the body to the neck of the clause. The clause body is then broken into two at the points where the above condition is found, and a parallel expression (a CGE) built for the right part of the sequence split. The splitting is done right after the leftmost goal involved in the condition. The motivation to do this is to find the longest parallel expressions possible. An alternative heuristic will proceed forwards and split right before the rightmost goal involved. The reasoning behind proceeding backwards is based on the observation that goals are generally more instantiated, and thus more likely to be independent, towards the end of the clause. Since as the algorithm progresses it makes choices (by creating expressions) that prevent later opportunities for parallelization, it seems more profitable to start from the end of the clause.

Let a CDG $cdg(B)$ be built for each clause $C \equiv h$:- B with $B = \langle g_1, \ldots, g_n \rangle$. Define $\mathcal{I} = \{\mathcal{I}(g_i, g_j) \mid i < j\}$, where $\mathcal{I}(g_i, g_j)$ are the sets of conditions such that g_i and g_j are independent, which are already simplified w.r.t. the available information valid before the execution of g_i, denoted $\mathcal{K}(g_i)$.

Example 6.1. Consider the clause `h(X):- p(X,Y),q(X,Z),r(X),s(Y,Z)`. With a simple local analysis, we have the following (note that $free\_not\_aliased(\mathtt{X}) \Rightarrow \neg indep(\mathtt{X},\mathtt{X})$):

$$
\begin{array}{l}
\mathcal{K}(\mathtt{p(X,Y)}) = \{free\_not\_aliased(\mathtt{Y}), free\_not\_aliased(\mathtt{Z})\} \\
\mathcal{I}(\mathtt{p(X,Y)}, \mathtt{q(X,Z)}) = \{indep(\mathtt{X},\mathtt{X}), indep(\mathtt{Y},\mathtt{Z})\} \\
\mathcal{I}(\mathtt{p(X,Y)}, \mathtt{r(X)}) = \{indep(\mathtt{X},\mathtt{X})\} \\
\mathcal{I}(\mathtt{p(X,Y)}, \mathtt{s(Y,Z)}) = \{indep(\mathtt{Y},\mathtt{Y}), indep(\mathtt{X},\mathtt{Z})\} = \{false\} \\
\mathcal{K}(\mathtt{q(X,Z)}) = \{free\_not\_aliased(\mathtt{Z})\} \\
\mathcal{I}(\mathtt{q(X,Z)}, \mathtt{r(X)}) = \{indep(\mathtt{X},\mathtt{X})\} \\
\mathcal{I}(\mathtt{q(X,Z)}, \mathtt{s(Y,Z)}) = \{indep(\mathtt{Z},\mathtt{Z}), indep(\mathtt{X},\mathtt{Y})\} = \{false\} \\
\mathcal{K}(\mathtt{r(X)}) = \emptyset \\
\mathcal{I}(\mathtt{r(X)}, \mathtt{s(Y,Z)}) = \{indep(\mathtt{X},\mathtt{Y}), indep(\mathtt{X},\mathtt{Z})\}
\end{array}
$$

So C will be compiled, under strict independence, into the following parallel

[5] Although in the example we will use a notation which looks like predicate logic, the clause variables are in fact constants in the theory underlying *improve*. Thus, the framework of Section 3.4, based on propositional logic, is still valid for our purpose in this section.

clause:

```
h(X) :- ground(X) ⇒ p(X,Y) & q(X,Z),
        indep(X,Y), indep(X,Z) ⇒ r(X) & s(Y,Z).
```

Note that the body is split at `q(X,Z)` (because of `Z`) and not at `p(X,Y)` (because of `Y`), the largest expression being achieved in this way. In fact, if the clause were split at `p(X,Y)`, no parallel expressions would be possible. Note also that the first CGE does not have the condition `indep(Y,Z)` since this condition is automatically satisfied by virtue of the fact that $\texttt{free\_not\_aliased}(\texttt{Z}) \in \mathcal{K}(\texttt{p}(\texttt{X},\texttt{Y}))$. □

Example 6.2. Consider the same clause above. We could apply the alternative heuristic of proceeding forwards, which will cause the splitting at `s(Y,Z)` because of $\mathcal{I}(\texttt{p}(\texttt{X},\texttt{Y}), \texttt{s}(\texttt{Y},\texttt{Z})) = \{false\}$. The resulting expression will be:

```
h(X) :- ground(X) ⇒ p(X,Y) & q(X,Z) & r(X), s(Y,Z).
```

Note however that, unless `X` is ground upon clause entry, this expression will result in no parallelism. □

The algorithm starts with a sequence B of literals (initially the body of the clause under consideration) and computes its corresponding parallel expression $exp(cdg(B))$.

Algorithm 6.1 MEL annotation. The compilation of a CDG $G = (V, E)$ to a parallel expression is given by $exp(G)$ as follows. Let the elements $\{g_1, \dots, g_n\}$ of V be ordered by relation prec.

function $exp_{MEL}(G)$: expression
begin
  compute p as the largest $j \in [1, n]$ s.t. $\exists i \in [j+1, n]\ \mathcal{I}(g_j, g_i) = false$;
  If there is no such j then p := 0;
  Let B1= $\{g_1, \dots, g_p\}$ and B2= $\{g_{p+1}, \dots, g_n\}$;
  IConds := $\bigcup_{\substack{p<i<n \\ i<j\leq n}} \mathcal{I}(g_i, g_j)$;
  D2 := $\langle\ goal(\texttt{IConds}) \Rightarrow g_{p+1}$ & ... & $g_n\ \rangle$;
  If B1 = $\emptyset$ then return D2;
  return $\langle\ exp_{MEL}(G|_{B1})$, D2 $\rangle$;
end.

Note that the definition of the algorithm uses function *goal* introduced in Section 5.1, and that in applying this function there is a possibility of further simplifying the condition w.r.t. the available information (in this case, that of $\mathcal{K}(g_{p+1})$).

Theorem 6.1 Correctness of MEL annotations. The execution of the expressions obtained by the MEL algorithm is correct w.r.t. their sequential semantics.

PROOF. We show that the CDG for the obtained expression is a super–CDG of the original one by induction. First, if no unconditional edge exists, $p = 0$ and B1 $= \emptyset$. In this case, `IConds` is the union of all the labels in the original CDG, and `D2` has this as condition. Therefore, the labels in the resulting CDG for `D2` are `IConds` in all of the edges; hence they are larger than the original ones. Since in this case the resulting expression is precisely `D2`, the hypothesis holds. Second, for the induction step, assume recursion satisfies the hypothesis for a number of calls. In a new call, D2 also satisfies it, with a similar reasoning than in the base case. Clearly, the resulting expression in this case, $\langle\ exp_{MEL}(G|_{B1}),\ \mathrm{D2}\ \rangle$, also satisfies it, since the sequentialization here gives unconditional edges. If corresponding edges existed in the original CDG, the new label $false$ is always larger than the original one. If they didn't exist, still the final CDG is a super–CDG of the original one. □

6.2. Extensions to CDG: UCDG Algorithms

Any modification of the CDG algorithm will result in some loss of parallelism for certain input graphs. The question is then how to minimize the loss, or, in other words, which of all the possible simplified expressions is best. Unfortunately, the answer to such question depends on many different parameters, like the granularity of the goals to be parallelized, the cost of the tests to be performed, the success/failure ratio of such tests, etc. For this reason we present two algorithms which are parameterized by several functions. These functions can de defined in terms of the compile-time information available on the above mentioned issues, thus reducing the loss of parallelism. As we will see, the choice between the two algorithms will also depend on such information.

We first propose an algorithm which tries to reduce the complexity of the expressions, while keeping the good properties of the UDG algorithm. It checks if the CDG can be partitioned into subsets which can then unconditionally be run in parallel. Otherwise, a condition is selected and enforced on the graph, in the hope that the partition will now be possible. When a partition is found, subexpressions are built for each subset, and all of them annotated to run in parallel. Both the selection of a condition and the annotation of the subexpressions are parameterized. An algorithm in this style will work as follows. Let $V = P \cup Q$ be a partition of the nodes of a CDG $G = (V, E)$, as before. The algorithm will compute subsets of Q which depend on one and only one element of P:

- $\forall p \in P\ Conn(p) = \{p\} \cup \{x \in V \mid (p, x, l) \in E^* \wedge \nexists p' \in P\ (p', x, l') \in E^*\}$
- $Conn(P) = \{Conn(p) \mid p \in P\}$

Then the subsets of $Conn(P)$ are just the connected components of the graph $G|_{Conn(P)}$. The sets $Conn(p_i)$ will act as the sets $Dep(P_i)$ in the UDG algorithm, but unlike the corresponding P_i's, they will always be disjoint. Thus, an unconditional parallel expression can be built for the subexpressions arising from recursively applying the algorithm to these subsets. Let $select(L)$ denote a function which selects a condition from (a subset of) a set L of them. Let $linear(C)$ denote the sequential expression corresponding to the nodes of set C (according to the $prec$ relation of Section 3.1). This algorithm builds the expression $exp(G)$ as follows.

```
function exp_UCDG(G):  expression
begin
  If V = {g} then return g;
  Let P = ready(G);
  If P = {p} then
    Cond := select({l ∈ Cond | (g_i, g_j, l) ∈ E});
    If Cond = false then return exp(G);
```

```
    return ⟨ Cond -> exp_UCDG(update(G,V,Cond)) ; linear(V) ⟩;
  else
    Let ConnP = Conn(P) = {C_1,...,C_n} and ConnP‾ = ∪_{C∈ConnP} C;
    return ⟨ exp(G|_C1) & ... & exp(G|_Cn), exp(G|_{V\ConnP‾}) ⟩;
  fi;
end.
```

Note that the resulting expression returned by the algorithm is parameterized on a generic function *exp* which is left open. We can instantiate this function to exp_{UDG} of Algorithm 4.3 (if the corresponding component is a UDG) or any of the other instances of *exp* which are defined in the paper, including the variants of CDG, MEL, and many more that can be defined.

Example 6.3. Consider the clause `h(X):- p(X),q(Y),r(X),s(Y)`. There is an unconditional dependency for `s(Y)` on `q(Y)` and a dependency labeled with *indep*(`X`,`X`) for `r(X)` on `p(X)`. While CDG will annotate it as:

```
h(X) :- ground(X) -> p(X) & r(X) & ( q(Y), s(Y) )
                   ; ( p(X), r(X) ) & ( q(Y), s(Y) ).
```

and UDG will annotate it as:

```
h(X) :- ( p(X), r(X) ) & ( q(Y), s(Y) ).
```

which is the worst case subexpression of the expression above, the UCDG algorithm, using exp_{UDG}, will annotate it as follows, where we use `=>` for brevity:

```
h(X) :- ( ground(X) => p(X) & r(X) ) & ( q(Y), s(Y) ).
```

which contains the subexpressions of UDG, one of them additionally annotated with a condition. □

In this rather simple example, the last expression is equivalent to the expression produced by CDG. However, this is not always the case. In particular, the UCDG algorithm behaves better for clauses with big bodies (which sometimes pose serious problems to CDG — see Section 8). The reason for this is that the UCDG algorithm behaves in a stepwise manner, first allowing unconditional parallelism to be annotated, and then postponing the consideration of the conditions until no more unconditional parallelism can be exploited.

In the above algorithm there is an implicit choice in the definition of the *select* function, where there is room for different heuristics. For example, this function could select the label with the lowest associated run-time cost. Such a function could also be used in Algorithm 5.1 or its variants, in order to reduce the size of the expressions it builds. Another option is to choose the label whose associated tests are more likely to succeed at run-time. In summary, the definition of *select* would depend on the compile-time information available.

A more radical way of combining the CDG and UDG algorithms is to use the UDG algorithms (4.2 and 4.3) explicitly. For this purpose all dependencies will be considered unconditional and the UDG algorithm applied. Then, the labels of the resulting expression will be considered, the aim being to improve such expressions by exploiting the conditional parallelism. An algorithm in this style would work as follows. Let *nodes*(*Exp*) denote the set of nodes corresponding to the atoms in the expression Exp, and $unconditional(V_1, V_2)$ be true if all dependencies of the nodes in V_2 on nodes of V_1 are unconditional.

```
function exp_UCDG(G):  expression
begin
  return Improve(exp_UDG(G));
end.

function Improve(Exp):  expression
begin
  If Exp is atomic then return Exp;
  If Exp = Exp_1 & ··· & Exp_n then
    return ⟨ Improve(Exp_1) & ··· & Improve(Exp_n) ⟩;
  If Exp = Exp_1, Exp_2 and unconditional(nodes(Exp_1), nodes(Exp_2)) then
    return ⟨ Improve(Exp_1), Improve(Exp_2) ⟩
  else
    return exp(nodes(Exp));
  fi;
end.
```

Note that the effect of the last `If` statement is similar to that of the MEL algorithm: if $unconditional(nodes(Exp_1), nodes(Exp_2))$ succeeds, this is probably a place where Algorithm 6.1 would have broken the original clause body in two parts. However, this new UCDG algorithm has the advantage of exploiting first unconditional parallelism.

Example 6.4. Consider the clause in Example 5.3, augmented with a new independent literal:

```
h :- t(z), p(X), q(Y), r(X), s(X,Y).
```

Algorithms 6.1 (MEL) and 5.1 (CDG) will build:

```
h :- t(z) & p(X) & q(Y), ground(X) => r(X) & s(X,Y)
```

neglecting the independence between `t(z)` and the other goals. The UCDG algorithm proposed avoids this by using the UDG algorithm, building:

```
h :- t(z) & ((p(X), r(X)) & q(Y), s(X,Y))
```
□

In the above example, if `ground(X)` succeeds, the first expression will exploit all available parallelism, while the second one will not. On the other hand, if the computational cost of `t(z)` is much greater than that of `p(X)` and `q(Y)`, the first expression will unnecessarily force the CGE to wait. This suggests that both granularity information and information regarding the probabilities of success of the tests, should be taken into account when choosing between the two algorithms. This issue will not be discussed further here and is left as future work.

7. Experimental Results

We have implemented the parallelization framework in the context of the &-Prolog system. The result is an automatic parallelizer which is parametric in the type of independence and the parallelization algorithm supported. The system has been instantiated to the case of strict independence in the Herbrand domain for our experimental study. We have selected for evaluation one algorithm in each of the interesting classes to compare them. Algorithm 4.3 was selected as the most promising variant of the algorithms which exploit only unconditional parallelism. Algorithm 5.1 is arguably the most interesting variation of the conditional algorithms, though with an exponential cost. The heuristic underlying Algorithm 6.1 (MEL), rather than that of UCDG, seems the most interesting variation of the conditional algorithms with polynomial cost. We have compared the performance of these three algorithms, both from the point of view of their behavior when annotating a program, and that of the annotated program when running in and–parallel. A relatively wide range of programs has been used as benchmarks.[6] Not all of them are discussed here; instead, we have selected a representative collection. Table 7.1 gives a short description of each benchmark, and Table 7.2 gives an overview of the complexity of each of them, useful for the interpretation of the results.

In Table 7.2, columns are read as follows. Cl is the number of clauses being actually annotated (dead code, which is detected by the analyzers, is not considered, as well as facts and clauses with single literals); AvG the average and MG the maximum number of goals in these clauses. CDGs is the number of graphs processed by the annotators; and AvGan the average and MGan the maximum number of goals in the CDGs. The rationale behind the CDGs, AvGan, and MGan columns in the table lies in the treatment of built–ins and side–effects. The first step in the compilation is to sequentialize these ones, as explained in Section 3. As a result, the CDG for the clause is actually partitioned into subgraphs at the points where side–effects or built-ins occur. Column CDGs shows the number of these subgraphs (which have more than one node, and thus worth considering) that the annotators have received as input.

To measure the effectiveness of the annotators we have carried out two kinds of tests: *static* and *dynamic*, and in two different situations. In the first situation, no global analysis is used, i.e. only local, clause level analysis, is performed ("loc" in the tables). In the second situation, a quite powerful global analysis is performed, using the combination of the Sharing+Freeness and ASub abstract interpreters described in [59, 60, 73, 21] ("abs" in the tables). Note that the information obtained in this case includes that of the local analysis.

7.1. Annotation Efficiency

Table 7.3 presents the results in terms of the compilation time required for annotation in seconds (SparcStation 10, one processor, SICStus 2.1, native code). It shows for each benchmark and annotator the average time out of ten executions in the two different situations mentioned. Note that the time taken in the analysis phase is not considered.

[6]Both system and benchmarks are available either by ftp at `clip.dia.fi.upm.es`, or from `http://www.clip.dia.fi.upm.es`, or by contacting the authors.

Bench.	Description
aiakl	Initialization phase for abstract unification in the AKL analyzer (D. Sahlin and T. Sjöland)
ann	The &-Prolog implementation of the MEL annotator
bid	Computes an opening bid for a bridge hand (J. Conery)
boyer	Reduced Boyer/Moore theorem prover (E. Tick)
browse	Pattern recognition of regular expressions (T. Dobry and H. Touati)
deriv	Symbolic differentiation
fib	Fibonacci numbers
grammar	Generates/recognizes a small set of English
hanoiapp	Solves the Towers of Hanoi problem
mmatrix	Multiplies two matrices
occur	Checks occurrences of sublists within lists of lists (B. Ramkumar and L. V. Kale)
palin	Recognizes palindromes (D.H.D. Warren)
progeom	Builds a perfect difference set (W. Older)
qplan	Supplies control for execution of a database query – Chat'80 (D.H.D. Warren)
qsortapp	Quick-sort algorithm (with append)
query	Small query to a database (D.H.D. Warren)
rdtok	R.A. O'Keefe's public domain Prolog tokenizer
read	D.H.D. Warren and R.A. O'Keefe's public domain Prolog parser
tak	Computes the Takeuchi function
tictactoe	Plays tic-tac-toe by alpha-beta pruning (A.K. Bansal)
warplan	Builds plans for robot control (D.H.D. Warren)
zebra	Zebra puzzle (V. Santos-Costa)

TABLE 7.1. Benchmark Description.

7.2. Performance of CGEs and Tests

One way to measure the effectiveness of the annotators is to count the number of CGEs which actually result in parallelism and to study the overhead introduced in the program by the tests generated. For this purpose we have measured the total number of checks which occur in the annotated programs ("T" in the tables), the number of these which are not checked during the execution of the program ("N"), and for the rest, the number of them which always succeeded ("S"), which always fail ("F"), and which sometimes succeed and others fail ("SF"). Also, the times the checks have succeeded ("TS") or failed ("TF") during execution, and the number of goals which have been run in parallel as a result ("E"). The results for each benchmark and each of the situations are shown in tables 7.4, 7.5, 7.6, 7.7, and 7.8. Note that each column shows ground checks on the left and indep on the right (ground/indep), except for UDG, since it only exploits unconditional parallelism.

Table 7.4 shows programs for which the annotated result is identical in all cases. In the programs of Table 7.5 MEL ("M") and CDG ("C") yield the same result (not UDG), but it is different with and without global analysis. The same happens in Table 7.6, but in this case the result of UDG is the same with and without

Bench.	Cl	AvG	MG	CDGs	AvGan	MGan
aiakl	7	3.00	5	2	3.50	5
ann	65	3.32	6	26	2.62	6
bid	18	2.78	5	8	2.50	4
boyer	10	3.60	6	2	2.00	2
browse	9	2.89	5	5	2.20	3
deriv	5	3.20	4	4	2.00	2
fib	1	6.00	6	1	2.00	2
grammar	4	2.50	3	4	2.25	3
hanoiapp	1	6.00	6	1	4.00	4
mmatrix	3	2.33	3	2	2.00	2
occur	3	3.00	4	2	2.00	2
palin	6	3.17	4	2	3.00	3
progeom	6	3.00	5	3	3.00	4
qplan	47	4.00	9	31	2.68	5
qsortapp	2	3.50	4	1	4.00	4
query	2	4.50	6	2	2.00	2
rdtok	46	3.43	8	0	0.00	0
read	37	4.14	7	2	2.00	2
tak	2	5.00	7	1	4.00	4
tictactoe	37	4.24	48	5	2.20	3
warplan	26	3.69	10	16	2.56	5
zebra	2	10.50	19	3	3.33	6

TABLE 7.2. Benchmark Profile.

information. In Table 7.8 all algorithms give the same result when global analysis information is available, but different otherwise. The rest of the programs appear in Table 7.7.

7.3. Speedup Results

An arguably better way of measuring the effectiveness of the annotators is to measure the speedup achieved: the ratio of the parallel execution time of the program (ideally for an unbounded number of processors) to that of the sequential program. This has the additional advantage of allowing to measure the impact of the overhead of the checks: even if the number of goals run in parallel is the same for different annotations ("E" in the previously mentioned tables), the checks actually performed can differ and cause differences in speedup.

In order to concentrate on the available parallelism itself, without the limitations imposed by a fixed number of physical processors, a particular scheduling, bus bandwidth, etc., we use a novel evaluation environment, called IDRA, proposed in [30]. IDRA takes as input a special execution trace file generated from a sequential (or, also, parallel) execution of the parallel program and the time taken by the sequential program, and computes the achievable speedup for any number of processors. The trace files list the events occurred during the execution of a parallel program, such as a goal being started or finished, and the times at which the events occurred. Since &-Prolog normally generates all possible parallel tasks

Benchmark program	loc			abs		
	MEL	CDG	UDG	MEL	CDG	UDG
aiakl	0.26	0.26	0.24	0.37	0.36	0.36
ann	1.55	1.55	1.43	7.60	7.60	7.53
bid	0.39	0.39	0.36	0.48	0.45	0.46
boyer	0.34	0.31	0.31	0.68	0.66	0.64
browse	0.53	0.46	0.45	0.63	0.56	0.55
deriv	0.20	0.18	0.18	0.27	0.26	0.25
fib	0.13	0.11	0.11	0.15	0.15	0.14
grammar	0.17	0.15	0.15	0.21	0.20	0.20
hanoiapp	0.18	0.18	0.16	0.22	0.20	0.20
mmatrix	0.21	0.19	0.19	0.22	0.21	0.20
occur	0.26	0.25	0.24	0.28	0.27	0.26
palin	0.22	0.20	0.20	0.41	0.38	0.39
progeom	0.20	0.19	0.18	0.25	0.24	0.24
qplan	1.59	1.67	1.35	3.63	3.43	3.43
qsortapp	0.17	0.16	0.16	0.19	0.18	0.18
query	0.26	0.23	0.23	0.29	0.27	0.28
rdtok	0.87	0.79	0.80	1.87	1.82	1.84
read	0.90	0.82	0.82	2.02	1.99	2.01
tak	0.17	0.15	0.15	0.23	0.21	0.21
tictactoe	0.90	0.81	0.81	2.08	2.03	2.02
warplan	0.54	0.54	0.51	2.89	2.86	2.77
zebra	2.08	300.86	0.57	4.96	4.65	4.64

TABLE 7.3. Efficiency Results for Annotators.

in a parallel program, regardless of the number of processors in the system, information is gathered for all possible goals that would be executed in parallel. Using this data, IDRA builds a task dependency graph whose edges are annotated with the exact execution times. The possible actual execution graphs (which could be obtained if more processors were available) are constructed from this data and their total execution times compared to the sequential time, thus making quite accurate estimations of (ideal — in the sense that parallelization overheads are not taken into account) speedups. Though ideal, the results have been shown to be very good approximations of the *best possible* parallel execution [30], and to match closely the actual speedups obtained in the &-Prolog system for the number of processors available for comparison.

The results for a representative subset of the benchmarks used are presented in figures 7.1, 7.2, and 7.3. For each benchmark and situation of analysis, a diagram with speedup curves obtained with IDRA is shown. Each curve represents the speedup achievable for the parallelized version of the program obtained with one annotator.

8. Discussion

Annotation times are fairly acceptable for all annotators. MEL and CDG usually take the same time, with a slight difference favoring CDG for simpler programs.

Benchmark	E
fib	986
grammar	0
rdtok	0
tak	2372

TABLE 7.4. Expressions with no checks — Identical code in all cases.

Bench. Prog.	Info	Ann	ground/indep							E
			T	N	S	F	SF	TS	TF	
deriv	loc	M/C	4/16	0/ 0	4/ 16	0/ 0	0/ 0	538/ 2152	0/ 0	538
	abs	all	0/ 0	0/0	0/0	0/0	0/0	0/0	0/ 0	538
mmatrix	loc	M/C	2/ 8	0/0	2/8	0/0	0/0	182/728	0/ 0	182
	abs	all	0/ 0	0/0	0/0	0/0	0/0	0/0	0/ 0	182
occur	loc	M/C	2/ 5	0/0	2/5	0/0	0/0	252/279	0/ 0	252
	abs	all	0/ 1	0/1	0/0	0/0	0/0	0/0	0/ 0	252
palin	loc	M/C	0/ 4	0/0	0/4	0/0	0/0	0/36	0/ 0	9
	abs	all	0/ 0	0/0	0/0	0/0	0/0	0/0	0/ 0	9
qsortapp	loc	M/C	0/ 1	0/0	0/1	0/0	0/0	0/250	0/ 0	250
	abs	all	0/ 0	0/0	0/0	0/0	0/0	0/0	0/ 0	250
query	loc	M/C	1/ 4	0/0	0/4	1/0	0/0	0/4	2/ 0	1
	abs	all	0/ 0	0/0	0/0	0/0	0/0	0/0	0/ 0	1
read	loc	M/C	1/ 6	0/0	1/6	0/0	0/0	1/6	0/ 0	1
	abs	all	0/ 0	0/0	0/0	0/0	0/0	0/0	0/ 0	1
tictactoe	loc	M/C	10/ 3	0/0	10/3	0/0	0/0	29796/5176	0/ 0	11124
	abs	all	0/ 0	0/0	0/0	0/0	0/0	0/0	0/ 0	11124
all	loc	udg	–	–	–	–	–	–	–	0

TABLE 7.5. Cases where MEL=CDG — Identical code for all annotators in "abs".

Bench. Prog.	Info	Ann	ground/indep							E
			T	N	S	F	SF	TS	TF	
boyer	loc	M/C	4/ 2	0/1	3/1	1/0	0/0	42/14	38348/ 0	14
	abs	M/C	4/ 0	0/0	3/0	1/0	0/0	42/0	38348/ 0	14
	all	udg	–	–	–	–	–	–	–	0
browse	loc	M/C	3/ 7	0/2	1/4	2/0	0/1	60/16300	25/ 20	4105
	abs	M/C	2/ 2	0/0	0/1	2/0	0/1	0/4105	25/ 20	4105
	all	udg	–	–	–	–	–	–	–	0

TABLE 7.6. Cases where CDG and MEL produce identical code.

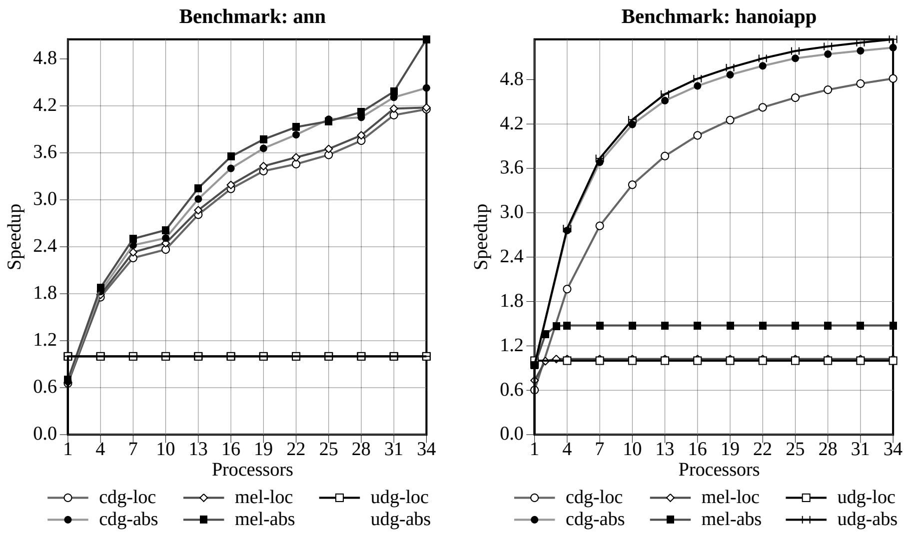


FIGURE 7.1. Effectiveness of Annotators: Dynamic Tests — ann/hanoiapp.

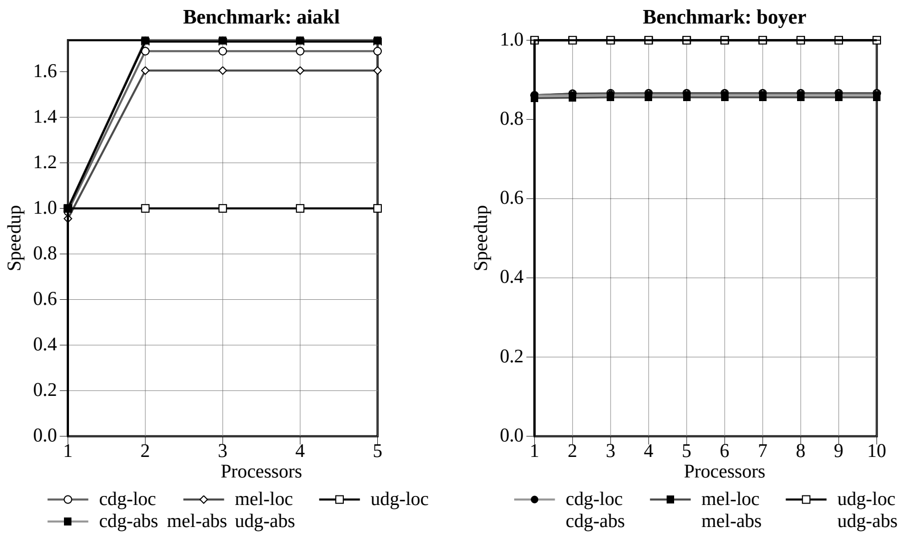


FIGURE 7.2. Effectiveness of Annotators: Dynamic Tests — Little Parallelism.

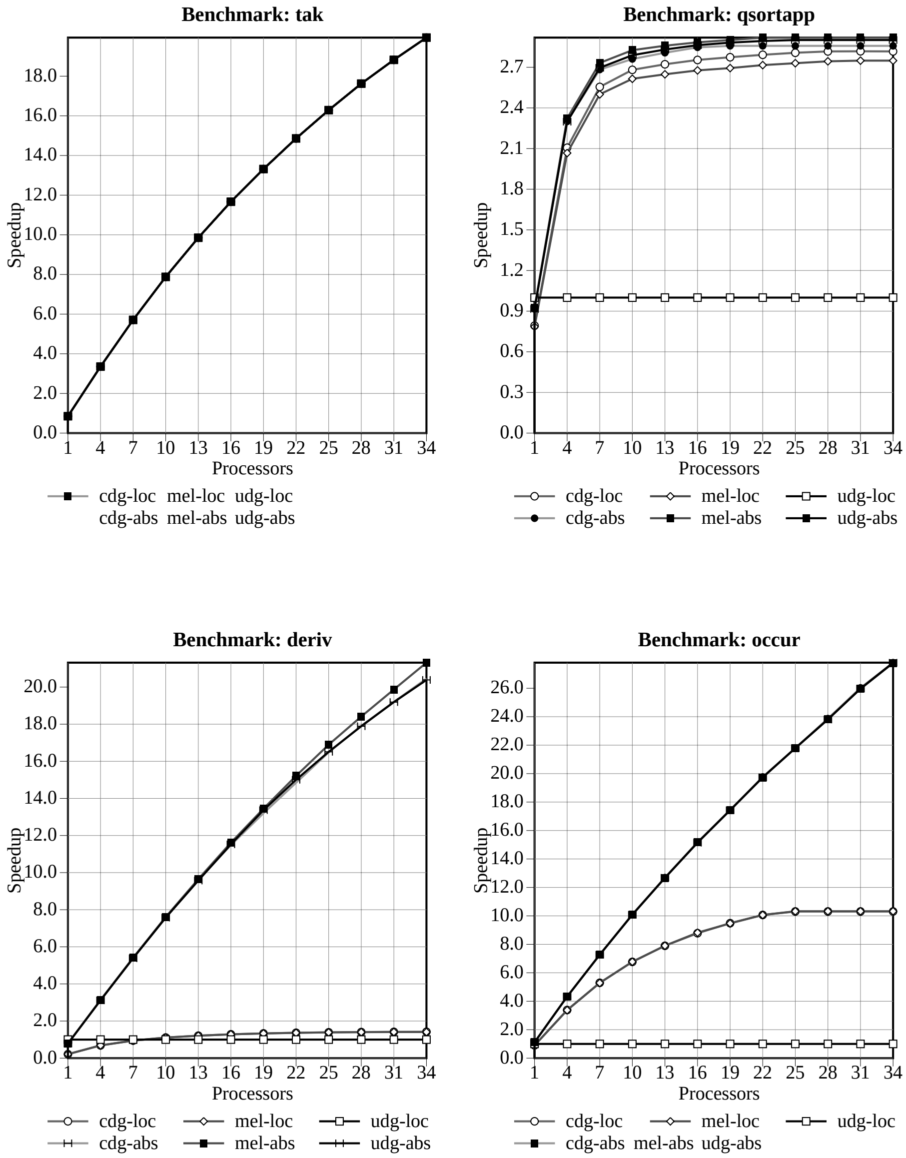


FIGURE 7.3. Effectiveness of Annotators: Dynamic Tests — Good Parallelism.

Bench. Program	Info	Ann	ground/indep							E
			T	N	S	F	SF	TS	TF	
ann	loc	mel	14/ 36	3/19	3/12	5/1	3/4	168/183	207/ 93	99
		cdg	22/ 46	6/29	5/12	8/1	3/4	180/183	297/ 93	99
		udg	–	–	–	–	–	–	–	0
	abs	mel	6/ 14	0/3	0/6	3/1	3/4	75/111	138/ 93	99
		cdg	12/ 18	2/8	1/5	6/1	3/4	81/105	228/ 93	99
		udg	–	–	–	–	–	–	–	0
hanoiapp	loc	mel	2/ 1	0/0	2/1	0/0	0/0	510/255	0/ 0	255
		cdg	5/ 1	2/1	3/0	0/0	0/0	765/0	0/ 0	255
		udg	–	–	–	–	–	–	–	0
	abs	mel	0/ 0	0/0	0/0	0/0	0/0	0/0	0/ 0	255
		cdg	0/ 0	0/0	0/0	0/0	0/0	0/0	0/ 0	255
		udg	–	–	–	–	–	–	–	255
qplan	loc	mel	13/ 57	9/47	3/10	1/0	0/0	6/12	3/ 0	7
		cdg	16/ 84	12/74	3/10	1/0	0/0	6/12	3/ 0	7
		udg	–	–	–	–	–	–	–	0
	abs	mel	2/ 1	2/1	0/0	0/0	0/0	0/0	0/ 0	7
		cdg	2/ 1	2/1	0/0	0/0	0/0	0/0	0/ 0	7
		udg	–	–	–	–	–	–	–	7
warplan	loc	mel	14/ 11	3/3	6/8	2/0	3/0	105/47	50/ 0	66
		cdg	28/ 15	13/9	8/5	3/1	4/0	113/45	58/ 4	66
		udg	–	–	–	–	–	–	–	6
	abs	mel	14/ 7	3/1	6/6	2/0	3/0	105/33	50/ 0	66
		cdg	28/ 10	13/6	8/3	3/1	4/0	113/29	58/ 4	66
		udg	–	–	–	–	–	–	–	6
zebra	loc	mel	0/ 250	0/247	0/2	0/1	0/0	0/112	0/ 56	1
		cdg	1/ 4835	0/4729	1/96	0/10	0/0	56/3346	0/ 420	1
		udg	–	–	–	–	–	–	–	1
	abs	mel	0/ 0	0/0	0/0	0/0	0/0	0/0	0/ 0	1
		cdg	0/ 0	0/0	0/0	0/0	0/0	0/0	0/ 0	1
		udg	–	–	–	–	–	–	–	1

TABLE 7.7. Other Programs.

On the contrary, MEL takes less time for complex programs, with zebra being an extreme example. Note that complexity here is measured as the number of literals in clauses: the higher the number of literals, the more linearizations of the clause graph are possible. This dominates the complexity of CDG, as it tries to consider all possible alternatives. UDG usually takes less than the other two without information (from global analysis), because in this case it can rarely find any opportunities for parallelization. When information from global analysis is available, UDG takes the same time as the other two. In several cases (like qplan, read, tictactoe, and also zebra) the annotation task is faster with global analysis. Since the input graph in this case is fairly simplified with the information from such analysis, the algorithms have to deal with less edges and shorter labels. This causes annotation with global analysis to be more efficient than without it. The unusually large annotation time for zebra is due to the low accuracy of the information provided by the local analyzer, which is unable to detect the definite dependencies which exist among all

Bench. Program	Info	Ann	ground/indep							E
			T	N	S	F	SF	TS	TF	
aiakl	loc	mel	0/10	0/0	0/10	0/0	0/0	0/10	0/0	2
		cdg	4/42	2/38	2/4	0/0	0/0	2/4	0/0	2
		udg	–	–	–	–	–	–	–	0
	abs	all	0/0	0/0	0/0	0/0	0/0	0/0	0/0	2
bid	loc	mel	7/12	2/0	4/12	1/0	0/0	17/44	1/0	27
		cdg	10/19	5/7	4/12	1/0	0/0	17/44	1/0	27
		udg	–	–	–	–	–	–	–	0
	abs	all	0/0	0/0	0/0	0/0	0/0	0/0	0/0	27
progeom	loc	mel	2/2	0/0	1/2	1/0	0/0	13/220	13/0	110
		cdg	2/2	0/0	1/2	1/0	0/0	13/220	13/0	110
		udg	–	–	–	–	–	–	–	0
	abs	all	0/0	0/0	0/0	0/0	0/0	0/0	0/0	110

TABLE 7.8. Cases where all annotated results have no checks with “abs”.

the six goals being considered for one clause. As a result, all possible combinations have to be explored. This is avoided when using the information provided by the Sharing+Freeness domain.

Regarding the parallelized programs resulting from annotation, we identify several classes of programs. Two purely sequential programs and two (simple) parallel programs appear in Table 7.4, the simplest cases. The annotators are successful at detecting such sequentiality and do not generate any parallel expression. In the case of simple parallel programs, where independence of goals can be inferred even with a local analysis of the clauses (global analysis in this case leads to no advantage), all the annotators are able to exploit this (unconditional) parallelism.

Programs whose parallelization is more complex, but still relatively easy, appear in Table 7.5. MEL and CDG (as well as UDG when having good information) are able to extract the available parallelism to a great extent. Also, all annotations produced lead to parallelism, i.e. no spurious parallel expressions (not really run in parallel, since their tests in fact fail at run–time) are generated. This is shown by the fact that none of the checks ever fail at execution time (“F” in the table). In fact, for MEL and CDG the annotated code is exactly the same, and thus the same parallelism is exploited. The worst case is that of UDG, which cannot exploit any parallelism without global analysis information.[7] When information is available, its annotated code is also identical to that of the other two: all annotators are able to extract the same amount of parallelism, and with expressions without run–time checks.

For more complex programs, like those of Table 7.8, the differences in the behavior of MEL and CDG are more apparent. Once again, for these programs the three annotators behave the same way when good global information is available, and extract the same parallelism as when not having such information, but without checks. Without information, though, annotators are forced to place some checks to be executed at run–time. In the case of CDG, it turns out that most of these

[7]This can actually be observed in all tables, except for the cases of warplan and zebra; the parallelism exploited in these cases is marginal, and with granularity analysis it would be avoided.

checks are not actually executed at run–time because many of the possible parallel expressions annotated by CDG are not used in the execution of the program. Nonetheless, note that in the case of aiakl, the expression exploited has much fewer checks than the corresponding one annotated by MEL (for the same goals in the program): 2 ground checks and 4 indep checks against 10 indep checks. This is due to the graph linearization performed by CDG, which takes all possibilities into account. If-then-elses built by CDG can be viewed as an "indexing" over the possible parallel expressions, based on some checks. In aiakl, this indexing is able to lead to the parallel expressions with less effort than that required by MEL, which simply puts conditions at certain points in the clause. Though this difference is not very relevant at execution time in the case of aiakl (see Figure 7.2), it can be so for other programs, and is an interesting feature (though expensive) of the CDG algorithm.

Table 7.6 shows two programs which are harder to parallelize. UDG cannot extract parallelism, because there is no unconditional parallelism. MEL and CDG extract the same amount of conditional parallelism, but for both algorithms the number of checks is less when global information is available. This is specially true for indep checks, since independence is not easy to reason about in "loc" (without global analysis). In fact, though, little parallelism is obtained. In the case of boyer, significant parallelism can be exploited but only using the concept of non–strict independence [46, 14]; in browse, although a good number of goals are executed in parallel, a critical part of the algorithm is still sequential.

Programs in Table 7.7 deserve more discussion. The first thing to be noticed is that in some cases UDG is not able to extract parallelism even with global information — this happens for ann, and for warplan and zebra, in which the parallelism extracted is marginal. On the contrary, for hanoiapp and qplan the same parallelism as the other two annotators is extracted by UDG. Considering the high complexity of qplan, global analysis turns out to be quite effective. Second thing is that global analysis shows also effective in reducing checks. This is precisely the reason of the speedups achieved with "abs" w.r.t. "loc" in ann (Figure 7.1) and hanoiapp (Figure 7.1), since the number of parallel goals run ("E" in Table 7.7) is actually the same.

Regarding MEL and CDG, it has to be noted that in most programs of Table 7.7 the overhead in number of checks of CDG is high. Although in some cases (e.g. qplan) it happens (as it happened in aiakl or bid) that these extra checks (and the corresponding expressions) are discarded at execution time, in other cases they do yield some overhead also at execution time. This is the case for ann, as can be seen in Figure 7.1, where speedups for CDG are always lower than for MEL. The same happens also for warplan.

An interesting case is that of hanoiapp. Its speedup curves (in Figure 7.1) illustrate a case where, with only local analysis, CDG achieves good speedups while MEL shows very little speedup. MEL correctly but inefficiently parallelizes a call to hanoi and a call to append, while CDG parallelizes a call to hanoi with a sequence composed of the other call to hanoi and a call to append. As shown in the example below, MEL needs an indep check, while CDG uses instead a ground check, which is much less expensive.

Example 8.1. For the clause of the Towers of Hanoi program whose CDGs appear in examples 3.2 and 3.3, the annotation result of CDG is shown below on the left, and that of MEL on the right.

```
shanoi(N0,A,B,C,M) :-
    N0 > 1,
    N1 is N0 - 1,
    ( ground([A,B,C]) ->
         shanoi(N,B,A,C,S)&
         ( shanoi(N,A,C,B,R),
           append(R,[mv(A,C)],T)
         ),
         append(T,S,M)
    ;
         shanoi(N,A,C,B,R),
         ( ground([A,C]),
           indep([[B,R]]) ->
              shanoi(N,B,A,C,S)&
              append(R,[mv(A,C)],T),
              append(T,S,M)
         ;
              shanoi(N,B,A,C,S),
              append(R,[mv(A,C)],T),
              append(T,S,M)
         ) ).
```

```
shanoi(N0,A,B,C,M) :-
    N0 > 1,
    N is N0 - 1,
    shanoi(N,A,C,B,R),
    ( ground([A,C]),
      indep([[B,R]]) ->
         shanoi(N,B,A,C,S)&
         append(R,[mv(A,C)],T)
    ;
         shanoi(N,B,A,C,S),
         append(R,[mv(A,C)],T)
    ),
    append(T,S,M).
```

In general, though, the differences in speedups between MEL and CDG are not very significant. Exceptions are hanoiapp, as discussed, and programs with very little parallelism, as aiakl (Figure 7.2). In this case, as in hanoiapp, CDG does better than MEL due to its ability to annotate different possibilities for the same clause body. In this program only one body with two parallel expressions is parallelized, and since the speedup achieved is very small, the differences between the annotations produced by the two algorithms are more relevant. For other programs with good speedups, as those in Figure 7.3, this does not happen.

9. Conclusions

We have proposed a proved correct a framework for the automatic parallelization of logic programs by program transformation. The transformation implies replacing conjunctions of literals with parallel expressions which at run-time trigger the exploitation of restricted, goal–level independent and–parallelism. Our framework consists of a two–step compilation process using conditional dependency graphs as an intermediate formalism. In the first step such graphs are constructed using a given notion of independence and simplified taking into account information gathered by program analysis. In the second step the conditional dependency graphs are converted into fork-join expressions and the original program rewritten by replacing the corresponding sequential conjunctions of goals with such expressions.

Several different algorithms for the second step in the process have been defined and studied. The UDG and CDG algorithms are based on the desirable objective of not losing a particular notion of "maximal" parallelism (which we have called μ-parallelism) available in the original program. Algorithms for determining whether this objective can be achieved at all using fork-join expressions have also been defined. Two alternatives of UDG for the case in which avoiding loss of parallelism is not possible have been presented and discussed. Our study suggests that one of these alternatives is more appropriate than the other one. Also, an alternative for CDG which makes it equivalent (modulo some conditions) to UDG has been proposed, as well as a new algorithm, UCDG, which combines the heuristics of UDG and CDG. A much less costly alternative for exploiting conditional parallelism, MEL, based on a simple but quite effective heuristic has also been proposed. Finally, we have also briefly discussed the importance of considering different alternatives for parallelization, but designing good heuristics (typically based on information regarding goal granularity) to select among them.

The three main annotation algorithms have also been implemented and studied experimentally. MEL and CDG have been shown to give very similar results in practice. Despite this, each one of them has demonstrated advantages and disadvantages. The results show CDG to be better when not having information from global analysis and if the programs are simple. Interestingly, CDG also shows advantage in more complex programs if good information from global analysis is available, because in these cases CDG can extract more sophisticated parallelism than MEL. On the contrary, for complex programs for which the analysis information is not accurate enough (or no analysis is available), the exponential nature of CDG can result in significant overhead, and thus MEL is a reasonable alternative. It appears that a good strategy to apply in practice may be to use the CDG algorithm in general, but apply MEL in clauses which are complex and/or for which there is imprecise analysis information, since for them CDG may be too expensive.

As expected, the UDG avoids any slow–downs caused by run-time independence checks. This makes this algorithm an obvious choice for completely transparent parallelization. However, our results show that the use of good analyses which make accurate information available is of crucial importance in this case. Otherwise UDG is not effective, obtaining small speedups or no speedups at all.

While not the main focus of this paper, our results point at the fact that the availability of accurate dependency information from global analysis is crucial in automatic parallelization. Although interesting speedups can be obtained in some cases using only local analysis, our overall conclusion, based on the improvements observed, is that global analysis based on abstract interpretation is indeed a powerful tool in this application. The effectiveness of this type of global analysis in automatic parallelization with the proposed model is studied in detail in [48, 33, 9, 10].

The general conclusion of our work, specially when seen in conjunction with the progress made in global analysis, is that, at least using the overall approach studied and the practical systems implemented, the task of automatic (constraint) logic program parallelization is feasible and practical. Useful speedups can be obtained for interesting programs while slow-downs can be avoided for those programs which the approach cannot parallelize.

However, much work remains to be done. The speedups described have been obtained on the current generation of medium-sized shared-memory multiprocessors, which are characterized by relatively small communication overhead. However, larger shared addressing space multiprocessors are starting to appear which support larger numbers of faster processors, but with higher communication overheads. Also, faster networks are starting to make exploiting parallelism in distributed platforms (multicomputers) more attractive. This requires accurate control of the sizes of the tasks to be parallelized: granularity control (see, e.g., [31] and its references). Taking into account granularity information requires some modifications to the annotation algorithms. Granularity information was already pointed out as one of the sources of heuristic information which can be used in CDG to choose among the alternatives it generates and reduce the overhead from the conditionals.

Another important avenue for improvement is the exploitation of more advanced notions of independence, specially a-posteriori ones. One such notion is "non-strict independence" [46]. Intuitively, this type of independence allows parallelizing procedures that share variables (i.e., pointers) by observing that the uses of such shared variables do not "interfere." We have recently developed an automatic parallelizer using non-strict independence [14]. This parallelizer uses the same framework (and implementation) proposed herein, although it was necessary to adapt the annotation algorithms. We are also working on applying the framework to the automatic parallelization of constraint logic programs, using as a starting point the generalized notions of independence presented in [24, 32]. Some results are reported in [23].

Larger programs tend to make more use of side-effects and sometimes of obscure features of the source language or operating system. A parallelizing compiler, and, especially, its global analysis phase, has to be able to deal correctly and as accurately as possible with these uses. We have addressed previously this problem [8] (and many of the solutions proposed are present in the analyzer used in this study), but this is also an area that requires additional work.

The compilation of programs into a language allowing goal-level, but unrestricted parallelism is another interesting topic. Restricted parallelism could be exploited when possible, with unrestricted expressions being annotated otherwise. In [64, 15] language primitives are proposed for expressing unrestricted parallelism. Moreover, another potentially important avenue for further improvement may be to detect parallelism at finer levels of granularity than the goal level used in our study [65, 37, 72]. An extension of the proposed parallelization framework in this direction is reported in [63]. In this context, the notion of *local* independence [56, 13, 12] allows the highest degree of parallelism proposed so far (to our knowledge). The tradeoffs between the additional parallelism obtained by finer grain parallelizations and the increased overheads involved need to be studied in detail. Finally, there remains the general issue of combining with or-parallelism [3, 55], which we have considered herein beyond our scope.

Acknowledgments

The authors would like to thank Maurice Bruynooghe, Kevin Greene, Roger Nasr, Francesca Rossi, Kish Shen, Richard Warren, Daniel Cabeza, and the anonymous referees for their very valuable comments and suggestions on the contents of this paper and previous versions of it.

REFERENCES

1. Hassan Ait-Kaci. *Warren's Abstract Machine, A Tutorial Reconstruction.* MIT Press, 1991.

2. K. A. M. Ali and R. Karlsson. Full Prolog and Scheduling Or-parallelism in Muse. *International Journal of Parallel Programming*, 1990. Vol. 19, No. 6, pp. 445–475.
3. K. A. M. Ali and R. Karlsson. The Muse Or-Parallel Prolog Model and its Performance. In *1990 North American Conference on Logic Programming*, pages 757–776. MIT Press, October 1990.
4. D. Bacon, S. Graham, and O. Sharp. Compiler Transformations for High-Performance Computing. *Computing Surveys*, 26(4):345–420, December 1994.
5. A. Bansal and L. Sterling. Transforming Generate-and-test Logic Programs to Committed-choice And-parallelism. *Int'l. Journal of Parallel Programming*, 18(5):401–446, 1989.
6. A. Bansal and L. Sterling. An Abstract Interpretation Scheme for Identifying Inherent Parallelism in Logic Programs. *New Generation Computing*, 7(2–3):273–324, 1990.
7. P. Brand, S. Haridi, and D.H.D. Warren. Andorra Prolog—The Language and Application in Distributed Simulation. In *International Conference on Fifth Generation Computer Systems*. Tokyo, November 1988.
8. F. Bueno, D. Cabeza, M. Hermenegildo, and G. Puebla. Global Analysis of Standard Prolog Programs. In *European Symposium on Programming*, number 1058 in LNCS, pages 108–124, Sweden, April 1996. Springer-Verlag.
9. F. Bueno, M. García de la Banda, and M. Hermenegildo. Effectiveness of Global Analysis in Strict Independence-Based Automatic Program Parallelization. In *International Symposium on Logic Programming*, pages 320–336. MIT Press, November 1994.
10. F. Bueno, M. García de la Banda, and M. Hermenegildo. Effectiveness of Abstract Interpretation in Automatic Parallelization: A Case Study in Logic Programming. *ACM Transactions on Programming Languages and Systems*, 21(2):189–238, March 1999.
11. F. Bueno, M. Hermenegildo, U. Montanari, and F. Rossi. From Eventual to Atomic and Locally Atomic CC Programs: A Concurrent Semantics. In *Fourth International Conference on Algebraic and Logic Programming*, number 850 in LNCS, pages 114–132. Springer-Verlag, September 1994.
12. F. Bueno, M. Hermenegildo, U. Montanari, and F. Rossi. Partial Order and Contextual Net Semantics for Atomic and Locally Atomic CC Programs. *Science of Computer Programming*, 30:51–82, January 1998. Special CCP95 Workshop issue.
13. F. Bueno Carrillo. *Automatic Optimisation and Parallelisation of Logic Programs through Program Transformation*. PhD thesis, Universidad Politécnica de Madrid (UPM), October 1994.
14. D. Cabeza and M. Hermenegildo. Extracting Non-strict Independent And-parallelism Using Sharing and Freeness Information. In *1994 International Static Analysis Symposium*, number 864 in LNCS, pages 297–313, Namur, Belgium, September 1994. Springer-Verlag.
15. D. Cabeza and M. Hermenegildo. Implementing Distributed Concurrent Constraint Execution in the CIAO System. In *Proc. of the AGP'96 Joint conference on Declarative Programming*, pages 67–78, San Sebastian, Spain, July 1996. U. of the Basque Country. Available from `http://www.clip.dia.fi.upm.es/`.
16. M. Carro, L. Gómez, and M. Hermenegildo. Some Paradigms for Visualizing Parallel Execution of Logic Programs. In *1993 International Conference on Logic Programming*, pages 184–201. MIT Press, June 1993.

17. J.-H. Chang, A. M. Despain, and D. Degroot. And-Parallelism of Logic Programs Based on Static Data Dependency Analysis. In *Compcon Spring '85*, pages 218–225, February 1985.

18. S.-E. Chang and Y. P. Chiang. Restricted AND-Parallelism Execution Model with Side-Effects. In Ewing L. Lusk and Ross A. Overbeek, editors, *Proceedings of the North American Conference on Logic Programming*, pages 350–368. MIT Press, Cambridge, MA, 1989.

19. J. Chassin and P. Codognet. Parallel Logic Programming Systems. *Computing Surveys*, 26(3):295–336, September 1994.

20. K. Clark and S. Gregory. Parlog: Parallel Programming in Logic. *Journal of the ACM*, 8:1–49, January 1986.

21. M. Codish, A. Mulkers, M. Bruynooghe, M. García de la Banda, and M. Hermenegildo. Improving Abstract Interpretations by Combining Domains. *ACM Transactions on Programming Languages and Systems*, 17(1):28–44, January 1995.

22. J. S. Conery. *The And/Or Process Model for Parallel Interpretation of Logic Programs.* PhD thesis, The University of California At Irvine, 1983. Technical Report 204.

23. M. García de la Banda, F. Bueno, and M. Hermenegildo. Towards Independent And-Parallelism in CLP. In *Programming Languages: Implementation, Logics, and Programs*, number 1140 in LNCS, pages 77–91, Aachen, Germany, September 1996. Springer-Verlag.

24. M. García de la Banda, M. Hermenegildo, and K. Marriott. Independence in Constraint Logic Programs. In *1993 International Logic Programming Symposium*, pages 130–146. MIT Press, Cambridge, MA, October 1993.

25. S. K. Debray, P. López García, M. Hermenegildo, and N.-W. Lin. Estimating the Computational Cost of Logic Programs. In *Static Analysis Symposium, SAS'94*, number 864 in LNCS, pages 255–265, Namur, Belgium, September 1994. Springer-Verlag.

26. S.K. Debray and N.W. Lin. Cost analysis of logic programs. *ACM Transactions on Programming Languages and Systems*, 15(5):826–875, November 1993.

27. D. DeGroot. Restricted AND-Parallelism. In *International Conference on Fifth Generation Computer Systems*, pages 471–478. Tokyo, November 1984.

28. D. DeGroot. A Technique for Compiling Execution Graph Expressions for Restricted AND-parallelism in Logic Programs. In *Int'l Supercomputing Conference*, pages 80–89, Athens, 1987. Springer Verlag.

29. D. Kuck et al. Dependence Graphs and Compiler Optimizations. In *8th Symposium on Principles of Programming Languages*, pages 207–218. ACM, January 1981.

30. M. Fernández, M. Carro, and M. Hermenegildo. IDeal Resource Allocation (IDRA): A Technique for Computing Accurate Ideal Speedups in Parallel Logic Languages. Technical report, T.U. of Madrid (UPM), June 1992.

31. P. López García, M. Hermenegildo, and S. K. Debray. A Methodology for Granularity Based Control of Parallelism in Logic Programs. *Journal of Symbolic Computation, Special Issue on Parallel Symbolic Computation*, 22:715–734, 1996.

32. M. García de la Banda. *Independence, Global Analysis, and Parallelism in Dynamically Scheduled Constraint Logic Programming.* PhD thesis, Universidad Politécnica de Madrid (UPM), Facultad Informatica UPM, 28660-Boadilla del Monte, Madrid-Spain, September 1994.

33. M. García de la Banda and M. Hermenegildo. A Practical Application of Sharing and Freeness Inference. In *1992 Workshop on Static Analysis WSA'92*, number 81–82 in BIGRE, pages 118–125, Bourdeaux, France, September 1992. IRISA-Beaulieu.

34. D. Gelernter, A. Nicolau, and D. Padua. *Languages and Compilers for Parallel Computing*. MIT Press, Cambridge, Mass., 1990.

35. G. Gupta, M. Hermenegildo, E. Pontelli, and V. Santos-Costa. ACE: And/Or-parallel Copying-based Execution of Logic Programs. In *International Conference on Logic Programming*, pages 93–110. MIT Press, June 1994.

36. G. Gupta and V. Santos-Costa. Cuts and Side-Effects in And-Or Parallel Prolog. *Journal of Logic Programming*, 27(1):45–71, April 1992.

37. G. Gupta, V. Santos-Costa, R. Yang, and M. Hermenegildo. IDIOM: Integrating Dependent And-, Independent And-, and Or-parallelism. In *1991 International Logic Programming Symposium*, pages 152–166. MIT Press, October 1991.

38. M. Hermenegildo. *An Abstract Machine Based Execution Model for Computer Architecture Design and Efficient Implementation of Logic Programs in Parallel*. PhD thesis, U. of Texas at Austin, August 1986.

39. M. Hermenegildo. An Abstract Machine for Restricted AND-parallel Execution of Logic Programs. In *Third International Conference on Logic Programming*, number 225 in Lecture Notes in Computer Science, pages 25–40. Imperial College, Springer-Verlag, July 1986.

40. M. Hermenegildo. Automatic Parallelization of Irregular and Pointer-Based Computations: Perspectives from Logic and Constraint Programming. In *Proceedings of EUROPAR'97*, volume 1300 of *LNCS*, pages 31–46. Springer-Verlag, August 1997. (invited).

41. M. Hermenegildo, D. Cabeza, and M. Carro. Using Attributed Variables in the Implementation of Concurrent and Parallel Logic Programming Systems. In *Proc. of the Twelfth International Conference on Logic Programming*, pages 631–645. MIT Press, June 1995.

42. M. Hermenegildo and K. Greene. The &-Prolog System: Exploiting Independent And-Parallelism. *New Generation Computing*, 9(3,4):233–257, 1991.

43. M. Hermenegildo and R. I. Nasr. Efficient Management of Backtracking in AND-parallelism. In *Third International Conference on Logic Programming*, number 225 in LNCS, pages 40–55. Imperial College, Springer-Verlag, July 1986.

44. M. Hermenegildo and F. Rossi. On the Correctness and Efficiency of Independent And-Parallelism in Logic Programs. In *1989 North American Conference on Logic Programming*, pages 369–390. MIT Press, October 1989.

45. M. Hermenegildo and F. Rossi. Non-Strict Independent And-Parallelism. In *1990 International Conference on Logic Programming*, pages 237–252. MIT Press, June 1990.

46. M. Hermenegildo and F. Rossi. Strict and Non-Strict Independent And-Parallelism in Logic Programs: Correctness, Efficiency, and Compile-Time Conditions. *Journal of Logic Programming*, 22(1):1–45, 1995.

47. M. Hermenegildo and The CLIP Group. Some Methodological Issues in the Design of CIAO - A Generic, Parallel, Concurrent Constraint System. In *Principles and Practice of Constraint Programming*, number 874 in LNCS, pages 123–133. Springer-Verlag, May 1994.

48. M. Hermenegildo, R. Warren, and S. K. Debray. Global Flow Analysis as a Practical Compilation Tool. *Journal of Logic Programming*, 13(4):349–367, August 1992.

49. D. Jacobs and A. Langen. Compilation of Logic Programs for Restricted And-Parallelism. In *European Symposium on Programming*, pages 284–297, 1988.

50. Joxan Jaffar and Jean-Louis Lassez. Constraint Logic Programming. In *ACM Symposioum on Principles of Programming Languages*, pages 111–119. ACM, 1987.

51. S. Janson and S. Haridi. Programming Paradigms of the Andorra Kernel Language. In *1991 International Logic Programming Symposium*, pages 167–183. MIT Press, 1991.

52. A. King and P. Soper. Schedule Analysis of Concurrent Logic Programs. In Krzysztof Apt, editor, *Proceedings of the Joint International Conference and Symposium on Logic Programming*, pages 478–492, Washington, USA, 1992. The MIT Press.

53. Robert A. Kowalski. *Logic for Problem Solving*. Elsevier North-Holland Inc., 1979.

54. Y. J. Lin and V. Kumar. AND-Parallel Execution of Logic Programs on a Shared Memory Multiprocessor: A Summary of Results. In *Fifth International Conference and Symposium on Logic Programming*, pages 1123–1141. MIT Press, August 1988.

55. E. Lusk et al. The Aurora Or-Parallel Prolog System. *New Generation Computing*, 7(2,3), 1990.

56. U. Montanari, F. Rossi, F. Bueno, M. García de la Banda, and M. Hermenegildo. Towards a Concurrent Semantics-based Analysis of CC and CLP. In *Principles and Practice of Constraint Programming*, number 874 in LNCS, pages 151–161. Springer-Verlag, May 1994.

57. K. Muthukumar and M. Hermenegildo. Complete and Efficient Methods for Supporting Side Effects in Independent/Restricted And-parallelism. In *1989 International Conference on Logic Programming*, pages 80–101. MIT Press, June 1989.

58. K. Muthukumar and M. Hermenegildo. The CDG, UDG, and MEL Methods for Automatic Compile-time Parallelization of Logic Programs for Independent And-parallelism. In *Int'l. Conference on Logic Programming*, pages 221–237. MIT Press, June 1990.

59. K. Muthukumar and M. Hermenegildo. Combined Determination of Sharing and Freeness of Program Variables Through Abstract Interpretation. In *1991 International Conference on Logic Programming*, pages 49–63. MIT Press, June 1991.

60. Kalyan Muthukumar. *Compile-time Algorithms for Efficient Parallel Implementation of Logic Programs*. PhD thesis, University of Texas at Austin, August 1991.

61. L. Naish. Parallelizing NU-Prolog. In *Fifth International Conference and Symposium on Logic Programming*, pages 1546–1564. University of Washington, MIT Press, August 1988.

62. E. Pontelli, G. Gupta, and M. Hermenegildo. &ACE: A High-Performance Parallel Prolog System. In *International Parallel Processing Symposium*, pages 564–572. IEEE Computer Society Technical Committee on Parallel Processing, IEEE Computer Society, April 1995.

63. E. Pontelli, G. Gupta, F. Pulvirenti, and A. Ferro. Automatic Compile-time Parallelization of Prolog Programs for Dependent And-Parallelism. In *Proc. of the Fourteenth International Conference on Logic Programming*, pages 108–122. MIT Press, July 1997.

64. B. Ramkumar and L. V. Kale. Compiled Execution of the Reduce-OR Process Model on Multiprocessors. In *1989 North American Conference on Logic Programming*, pages 313–331. MIT Press, October 1989.

65. V. Santos-Costa, D.H.D. Warren, and R. Yang. Andorra-I: A Parallel Prolog System that Transparently Exploits both And- and Or-parallelism. In *Proceedings of the 3rd. ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM, April 1990.

66. V. Saraswat. *Concurrent Constraint Programming Languages.* PhD thesis, Carnegie Mellon, Pittsburgh, 1989. School of Computer Science.

67. V. Saraswat, M. Rinard, and P. Panangaden. Semantic Foundation of Concurrent Constraint Programming. In *Proceedings of the 18th. Annual ACM Conf. on Principles of Programming Languages*. ACM, 1991.

68. V. Sarkar. *Partitioning and Scheduling Parallel Programs for Multiprocessors*. Pitman, London, (1989).

69. V. Sarkar. Instruction Reordering for Fork-Join Parallelism. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, volume 25, pages 322–336, June 1990.

70. E.Y. Shapiro, editor. *Concurrent Prolog: Collected Papers*. MIT Press, Cambridge MA, 1987.

71. E.Y. Shapiro. The Family of Concurrent Logic Programming Languages. *ACM Computing Surveys*, 21(3):412–510, September 1989.

72. K. Shen. Overview of DASWAM: Exploitation of Dependent And-parallelism. *Journal of Logic Programming*, 29(1–3):245–293, November 1996.

73. H. Sondergaard. An application of abstract interpretation of logic programs: occur check reduction. In *European Symposium on Programming, LNCS 123*, pages 327–338. Springer-Verlag, 1986.

74. S. Taylor, S. Safra, and E. Shapiro. A Parallel Implementation of Flat Concurrent Prolog. In E.Y. Shapiro, editor, *Concurrent Prolog: Collected Papers*, pages 575–604, Cambridge MA, 1987. MIT Press.

75. E. Tick. The Deevolution of Concurrent Logic Programming Languages. *The Journal of Logic Programming*, 23(1–3):89–125, 1995.

76. E. Tick and C. Bannerjee. Performance evaluation of monaco compiler and runtime kernel. In *1993 International Conference on Logic Programming*, pages 757–773. MIT Press, June 1993.

77. K. Ueda. *Guarded Horn Clauses*. PhD thesis, University of Tokyo, March 1986.

78. K. Ueda. Guarded Horn Clauses. In E.Y. Shapiro, editor, *Concurrent Prolog: Collected Papers*, pages 140–156. MIT Press, Cambridge MA, 1987.

79. K. Ueda. Making Exhaustive Search Programs Deterministic. *New Generation Computing*, 5(1):29–44, 1987.

80. D.H.D. Warren. An Abstract Prolog Instruction Set. Technical Report 309, Artificial Intelligence Center, SRI International, 333 Ravenswood Ave, Menlo Park CA 94025, 1983.

81. D.H.D. Warren. OR-Parallel Execution Models of Prolog. In *Proceedings of TAPSOFT '87*, Lecture Notes in Computer Science. Springer-Verlag, March 1987.

82. D.H.D. Warren. The Extended Andorra Model with Implicit Control. In Sverker Jansson, editor, *Parallel Logic Programming Workshop*, Box 1263, S-163 13 Spanga, SWEDEN, June 1990. SICS.

83. W. Winsborough and A. Waern. Transparent And-Parallelism in the Presence of Shared Free variables. In *Fifth International Conference and Symposium on Logic Programming*, pages 749–764, Seattle,Washington, 1988.

84. X. Zhong, E. Tick, S. Duvvuru, L. Hansen, A.V.S. Sastry, and R. Sundararajan. Towards an Efficient Compile-Time Granularity Analysis Algorithm. In *Proc. of the 1992 International Conference on Fifth Generation Computer Systems*, pages 809–816. Institute for New Generation Computer Technology (ICOT), June 1992.