RAPiD
A Data Flow Model for Implementing Parallelism and Intelligent Backtracking in Logic Programs

Bernd Schwinn
Gerhard Barth*
Christoph Welsch
University of Stuttgart
West Germany

Abstract
Since further improvements in speed for sequential architectures are limited, the development of parallel implementation techniques for logic programming languages has become an increasingly important area of research. This paper presents a data flow model, which allows an efficient realization of the nondeterminisms contained in logic programs, especially of AND–, OR–nondeterminism and backtracking. The model is based primarily on a dependency analysis similar to Restricted AND–Parallelism [DeG84], which starts with a static classification of dependency situations among subgoals. The classification performed in our model results in a single data flow graph representation for each rule. These graphs contain simple tests, which can decide at execution time on the dependency of subgoals. Additionally, we use this kind of dependency analysis to organize intelligent backtracking. Finally, our model allows realization of OR–parallelism similar to the method used in the AND/OR Process Model [Con83].

1. Introduction

During the last years logic programming has become increasingly important, chiefly in the area of knowledge engineering. This trend continues although Prolog [ClM81], the most popular logic programming language, suffers from an inefficient inference strategy. This inefficiency is due to the underlying architectures, which do not exploit important properties of logic programming, like parallelization and flexibility.
When solving a (sub)goal, the inference strategy of Prolog probes the clauses (rules and facts) contained in the knowledge base in sequential order. An alternative clause is only used if the priviously chosen clause fails to satisfy the (sub)goal. The subgoals in the body of a rule are ordered from left to right and are persued in that order when the rule is applied. If a subgoal cannot be reached with any clause contained in the knowledge base, backtracking has to be performed. In this case, Prolog tries to find an alternative solution for the last subgoal, that had been investigated before the execution of the failed subgoal was started. This means that Prolog backtracks through the subgoals in reverse execution order (from right to left).
Since the subgoals to be fulfilled are conjunctively connected, a solution must be found for each of these subgoals. From the logical point of view, the subgoals

* German Artificial Intelligence Laboratory, Kaiserslautern

could be solved in any order (**AND–nondeterminism**) or even in parallel. The latter kind of parallelism is called **AND–parallelism**.
When a subgoal has to be reached, alternative clauses contained in the knowledge base can be applied in any order (**OR–nondeterminism**). Even a parallel search for alternative solutions is possible. This kind of concurrency is called **OR–parallelism**.
If the execution of a subgoal fails, this can be caused by a variable binding determined during the execution of a subgoal solved previously. Backtracking could lead to a different binding for such a variable, which is a necessary condition for the failed subgoal to be persued again. In the case of a failed subgoal, the previously solved subgoals could be backtracked through in any order. Therefore, **backtracking** is another kind of nondeterminism. Prolog performs backtracking no matter what really caused the failure. Thus, reinvestigations of subgoals are possible which do not eliminate the failure. To avoid such useless recalculations, **intelligent backtracking** methods should resolve only those subgoals which could be able to rebind variables participating in the failed subgoal.

This paper is organized as follows: In the second section the most popular methods for realization of AND–parallelism, OR–parallelism and intelligent backtracking are discussed. In this discussion, the advantages of a RAP–like dependency analysis according to AND–parallelism and intelligent backtracking are pointed out and an efficient process model to organize OR–parallelism and intelligent backtracking is sketched. RAPiD, a data flow model based on Restricted AND–Parallelism and this process model is introduced in the third section. There, each clause is represented by a single data flow graph, consisting of simple actions to be performed when the clause is executed. The construction of a graph for a given rule is described in the fourth section. The fifth section sketches a simulation system that has been implemented. Some results from tests about the degree of parallelism that can be activated with our RAPiD model are given in section six. Finally, some suggestions for further work are mentioned.

2. Parallelism and Intelligent Backtracking

To increase the efficiency of logic programming systems, many researching groups try to make a better use of the nondeterminisms mentioned in the introduction. This section gives a classification of the most popular methods for realization of AND–parallelism, OR–parallelism and intelligent backtracking and compares their performance. The most promising of these methods have been combined in the RAPiD model.

2.1. AND–Parallelism

AND–parallelism tries to satisfy subgoals belonging to a single calculation concurrently. A problem in this context arises if subgoals share variables. In those cases it must be guarenteed that a shared variable is uniquely bound. To avoid repeated recalculations only those subgoals should be executed in parallel, which do not share variables (independent subgoals). Nearly all popular methods use this kind of AND–parallelism, called **independent AND–parallelism** [Sch88].
To realize independent AND–parallelism, the subgoals that share variables ide-

pendent subgoals) must be synchronized. The **synchronization** could be done **explicitely**, like in the parallel languages Concurrent Prolog [Sha83], Parlog [ClG84] and GHC [Ued85]. The drawback of these techniques is the additional administrative overhead for the programmer which increases the danger of programming errors and often reduces the flexibility of programs.
On the other hand, **implicit synchronization** guarantees that only independent subgoals are executed concurrently. To implement this, a dependency analysis among the subgoals contained in the clauses has to be performed. It has to examine the following two kinds of dependencies:

• **variables with identical names** (common variables)
In a rule of the form

p(X) :- q(X) , r(X) , s(X) , ...

the three subgoals are usually dependent and should be executed sequentially. One of these subgoals must be chosen as producer to be executed first. If this execution leads to a constant binding (ground value) for the variable *X*, the two remaining subgoals might be executed in parallel. Furthermore, if the rule is activated by a goal of the form

?- p(a) .

all three subgoals are independent and could be executed concurrently.

• **binding between variables**
The two subgoals in a rule of the form

p(X,Y,...) :- q(X) , r(Y) , ...

are usually independent and can therefore be executed in parallel. But if the rule is activated to reach a goal like

?- p(A,A,...) .

there arises a dependency between the subgoals, which forces the subgoals to be executed sequentially.

The methods for implicit synchronization differ primarily in the instants they perform the dependency analysis. They result in a different execution overhead and in the degree of AND-parallelism to be gained:
Static dependency analysis [Cha85] is performed completely before the clauses are executed. As can be seen in the case studies given above, AND-parallelism depends heavily on the way a clause is activated. To guarantee that only independent subgoals are executed concurrently, this kind of analysis uses 'worst case' estimations of dependencies. In many cases this results in a suboptimal degree of AND-parallelism.

On the other hand, **dynamic dependency analysis** [Con83] is performed completely at execution time. Because this method can use all information about the given clause activation, the optimal degree of independent AND-parallelism can be realized. The drawback of this kind of analysis, being performed each time a clause is activated, is the additional amount of time it often consumes.
Restricted AND-Parallelism (RAP), proposed by DeGroot [DeG84], is a technique that combines the advantages of the two methods mentioned above. This kind of dependency analysis is performed in two steps: When the rules are defined, possible dependency situations among subgoals are stated. To represent these situations, the rules are transformed into conditional expressions containing tests to be performed at execution time. The tests examine dependencies of the two kinds discribed in the case studies given above. They are very simple and do not waste too much execution time. On the other hand, some AND-parallelism can be lost, especially if structured terms (like lists) are involved. Furthermore, DeGroot gives another cause for restriction of AND-parallelism, which is due to his way of representing conditional expressions as strongly nested lists (Conditional Graph Expressions (CGEs) [DeG84], [Her86]). The latter kind of restriction can be avoided, if data flow graphs are used to represent conditional expressions. For more details see section six and [Sch88].
Altogether, in many cases Restricted AND-Parallelism is more efficient than a pure static or a pure dynamic analysis, because it combines reduced execution overhead with a high degree of independent AND-parallelism [Sch88]. Therefore, this kind of dependency analysis is used in the RAPiD model.

2.2. OR-Parallelism

OR-parallelism aims at searches for alternative solutions in parallel. Since the alternative calculations do not depend on each other, a restriction on this kind of parallelism seems not necessary. On the other hand, **unrestricted OR-parallelism** would lead to a breadth-first search in the AND/OR search tree [Kow79], which could result in an enormous administrative overhead. Therefore, this kind of search can cause a serious problem, although no backtracking is needed.
To reduce overhead, the parallel logic languages Concurrent Prolog [Sha83], Parlog [ClG84] and GHC [Ued85] use **committed choice OR-parallelism**. In these languages the body of a rule may begin with a guard, consisting of a list of preconditions to be checked before the body is executed. If more than one rule may be applied to reach a goal, the guards of those rules can be executed in parallel. The guard succeeding first permits the body of the according rule to be executed and execution of the other rules is abandoned ('committed choice'). The drawback of this method is that the committed choice explores only one alternative. This can be very restrictive because possible (alternative) solutions will never be detected. Thus, committed choice is only suited if deterministic problems have to be solved.
The most promising realizations restrict OR-parallelism to the **availability of resources**, especially on the number of free processors [Cie84], [WRC87]. By exploiting OR-parallelism until all processors are busy, a good load balancing on the processors can be reached and administrative overhead is reduced. Furthermore, there is no restriction on the number of solutions that may be detected.

The latter method for organization of OR-parallelism may also be realized in RAPiD. In this model, the execution of each subgoal is managed by an independent process. As in Conery's AND/OR process model [Con83], where the execution of each subgoal is oranized by a single process, alternative clauses can be applied concurrently. The result obtained first is further investigated, while alternative results are stored in the process to be reactivated on backtracking. To realize this strategy the following four states of a process are distinguished:

GATHER — A process is in the GATHER state before the execution of the according subgoal is started.

WAITING — The execution of the subgoal has been started, but no result has been derived yet.

EXECUTING — The first result has been obtained, while the execution of some alternative clauses for the subgoal is still in progress.

SOLVED — The execution of each alternative clause for the subgoal to be reached has been finished.

If several results might be derived within a clause activation, this process model computes them sequentially. After the first result has been sent to the process that activated a clause, backtracking is performed in that clause to explore another result.
In our model, OR-parallelism can be realized if several clauses may be used alternatively to derive a subgoal. The alternative clauses can be executed on different processors. An efficient distribution could be reached if idle processors ask busy processors to execute alternative clause activations.

2.3. Intelligent Backtracking

If the execution of a subgoal has failed, the aim of intelligent backtracking is to redo those subgoals which may be responsible for the failure. In particular, these are the subgoals which might produce alternative bindings for the variables involved in the failure situation.

Example: Suppose, while executing the subgoal $q(X,Y,Z)$ in the rule

$$p(X,Y,Z) \;:-\; q(X,Y,Z)\;,\; r(X)\;,\; s(Y)\;,\; t(Z).$$

a ground value is bound to the variable Z. If the execution of the subgoals $r(X)$ and $s(Y)$ succeeds while the subgoal $t(Z)$ fails with the ground value produced for Z, an attempt to redo the subgoals $r(X)$ and $s(Y)$ will be useless, because none of these subgoals could change the value of Z. Therefore, the subgoal $q(X,Y,Z)$ should be backtracked immediately to produce an alternative binding for Z.

To find out which subgoals should be redone when the execution of a subgoal fails, intelligent backtracking methods perform a dependency analysis. The most popular strategies differ primarily in time and level on which the analysis is performed. They result in different execution overhead and amount of useless recalculations.
'Selective Backtracking', proposed by Pereira and Porto [PeP82], performs **dependency analysis** completely **at execution time**. This method uses a very detailled analysis, which is performed on the level of variables. Although this method leads to a small number of useless recalculations, it suffers from a high execution overhead.

Because the dependency analysis used to organize **intelligent backtracking** is similar to that used for **AND-parallelism**, combinations of both have proven to be very useful. In those cases, the dependency analysis has to be performed only once. It controls the resulting execution overhead and the overall number of useless recalculations.
Dynamic analysis is performed completely at execution time. Therefore, the intelligent backtracking scheme of Lin, Kumar and Leung [LKL86], also performing the analysis on the level of variables, is very similar to 'selective backtracking'. The result is a small number of useless recalculations, but the execution overhead is high.
A **RAP-like analysis** also collects the backtrack information at execution time, since the actual dependencies are not known earlier. If the dependency analysis is performed on the level of subgoals, the amount of information that has to be administrated will be reduced [Sch88]. Furthermore, the representation of rules being constructed to organize AND-parallelism may also be used for intelligent backtracking. On the other hand, a RAP analysisthe can restrict the degree of AND-parallelism. Therefore, compared to a dynamic analysis, the number of useless recalculations could be higher.
If a **static analysis** is used [Cha85], the backtrack paths for each subgoal are determined before execution time. Therefore, intelligent backtracking consumes only a small amount of additional time. Since a static analysis leads to more restrictions according to AND-parallelism, the number of useless recalculations can increase compared with the two kinds of analysis mentioned above.

If intelligent backtracking is performed in combination with AND-parallelism, special problems have to be solved. An example shall illustrate three cases that can be distinguished.

Example: Suppose the rule

$$p(X,Y,Z) \text{ :- } q(X,Y,Z) \text{ , } r(X,A) \text{ , } s(Y,B) \text{ , } t(Z,C) \text{ , } u(A,B,C).$$

is executed as follows: After the subgoal $q(X,Y,Z)$ has succeeded, the three subgoals $r(X,A)$, $s(Y,B)$ and $t(Z,C)$ are independent and may be executed in parallel. Finally, the results are used to examine $u(A,B,C)$.

• If the subgoal *q(X,Y,Z)* cannot be reached, the execution of the given rule is terminated without success.

• If one of the independent subgoals (e. g. *t(Z,C)*) fails, the subgoal *q(X,Y,Z)* should be redone. While doing so, the context for the two remaining subgoals (e. g. *r(X,A)* and *s(Y,B)*) will be destroyed. Therefore, their execution should be *cancel*ed.

• If the subgoal *u(A,B,C)* fails, this can be caused by a binding constructed during the execution of one of the three independent subgoals. Therefore, one of them should be redone. If one is chosen to be solved alternatively (e. g. *t(Z,C)*) and the execution of *u(A,B,C)* fails with all alternative solutions, another subgoal (e. g. *s(Y,B)*) should be redone. To guarantee completeness, each alternative solution for this subgoal should be combined with each solution for the subgoal(s) redone previously to be tested in *u(A,B,C)*. Therefore, the latter subgoal(s) should be *reset* to be able to reproduce all its(their) solutions.

In RAPiD this kind of intelligent backtracking is realized in combination with the process model for organization of OR–parallelism given in section 2.2. The method used is similar to the backtracking algorithm presented in [LKL86], which presumes a total order among the subgoals of each rule to guarantee completeness. If the execution of a subgoal P_i fails, the according process can look up a list of backtracking addresses (B–list) for a subgoal to be redone. The B–list, consisting of subgoals that may be able to cure the failure, is sorted in descending order. If the B–list of subgoal P_i has the form [P_j | REST], subgoal P_j is one of the most recently executed subgoals, which should be backtracked first. Therefore, the **backtracking method** used in RAPiD performs the following actions:

1. The execution of each subgoal being dependent from P_j is *cancel*ed.

2. Each subgoal, which is independent of P_j and has been redone previously, is *reset* to guarantee completeness.

3. To preserve the history of a failure, the B–list of the subgoal P_j and the REST of the B–list of P_i are *merge*d.

4. To search for an alternative solution for P_j, this subgoal is *redo*ne.

3. Data Flow Model

To explore parallelism and intelligent backtracking in logic programs we use a model, in which each clause is represented as an (independent) data flow graph. The edges of these graphs are used to transport data packets (tokens) between nodes. The nodes denote basic actions to be performed when the clause is executed. The data flow mechanism allows an operation to be executed as soon as all its operands are available. Since the operations can never be executed earlier,

a data flow model is an excellent base for increasing execution speed.
To activate the graphs, we use a dynamic interpretation model [Arv82], which allows several independent computations to proceed concurrently in a graph, each with its own 'color'. For further details see [Sch88].

3.1. Colored Tokens

A token T can be regarded as a message between two basic operations. In our model it consists of the following components:

T = (CT , RA , BL , DST , LD)

where
- CT = context (color)
- RA = return address
- BL = list of backtracking addresses
- DST = destination, specified by (G,N,P)
  - where G = graph number
  - N = node number
  - P = port number
- LD = literal data

The literal data portion of a token consists of a list of variable bindings established so far.

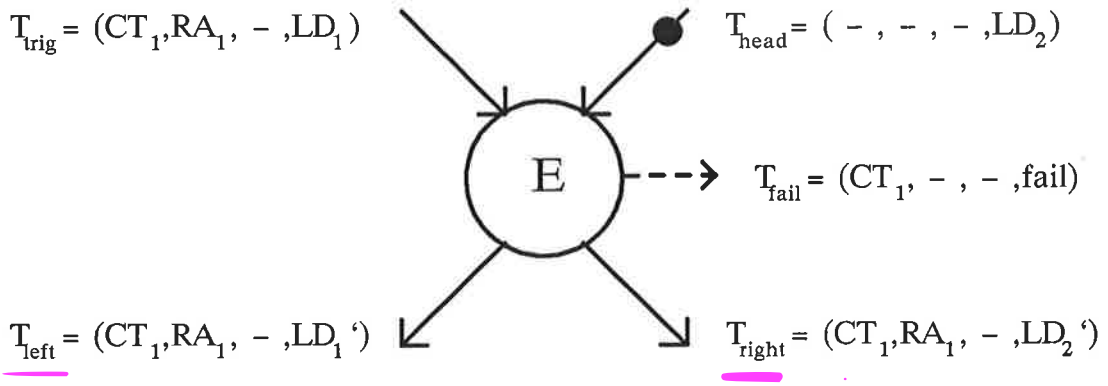
3.2. Basic Operations

In our data flow model we use the following basic operations:

(1) E = Entry–Unification
(2) U = Update
(3) C = Copy
(4) A = Apply
(5) R = Return
(6) G = Ground–Test
(7) I = Independence–Test

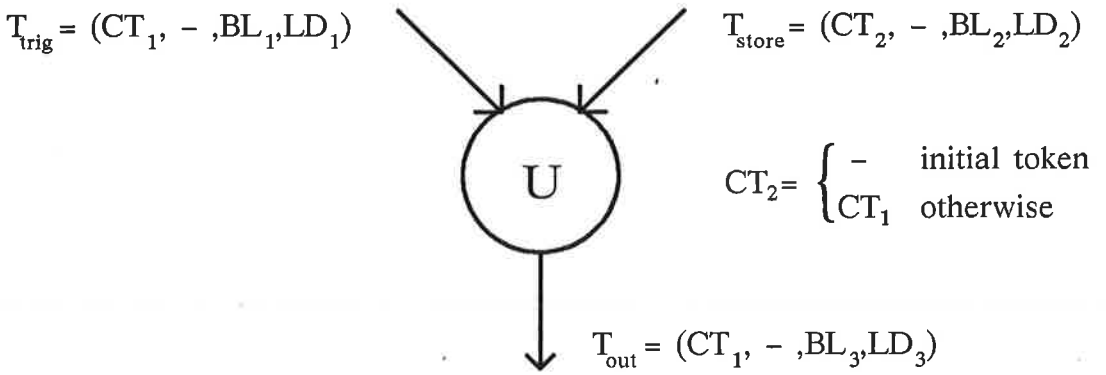
The most important and expensive among these are the E– and the A–operation. Each activation of a graph is started by a termwise unification (E) between a (sub)goal and the head of a clause. Unification is the fundamental operation in each resolution proof. In our data flow model, the execution of each subgoal is performed by a single process. Apply (A) is responsible for the administration of such a process. In our model, the A–operation organizes OR–parallelism and intelligent backtracking.
To communicate results between subgoals, the U–operation is used. While doing this, the subgoals are synchronized. The C–operation copies tokens for distributed use. To return results from a graph, our data flow model uses the R–operation. Finally, the kinds of dependencies studied in 2.1. are exemined by two tests (G,I). The semantics of the basic operations are given next using colored tokens. To simplify matters, the DST–fields are not listed explicitly.

(1) Entry–Unification



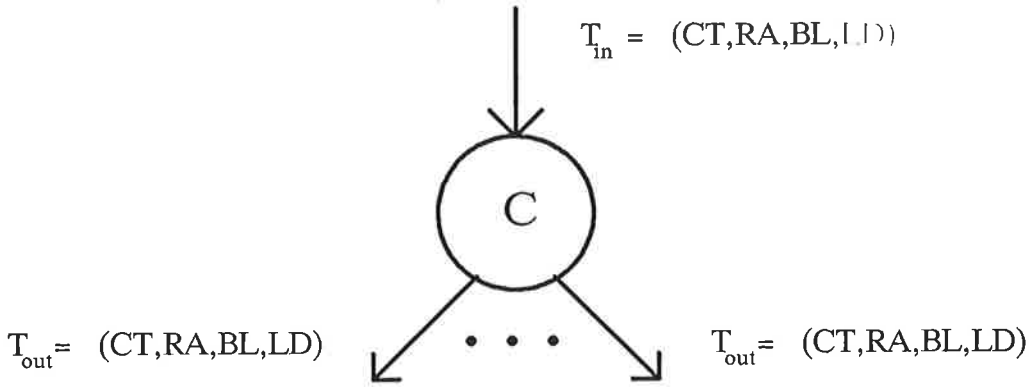
An E–operation performs a termwise unification between a trigger–token T_{trig}, representing the binding environment for a subgoal to be solved, and the head of a clause which resides permanently in a token T_{head} on the right input. Bindings determined during unification are given to the environments represented by the input tokens to produce the output tokens. If unification does not succeed, no output tokens are created and a token T_{fail} is sent to the return address.

(2) Update



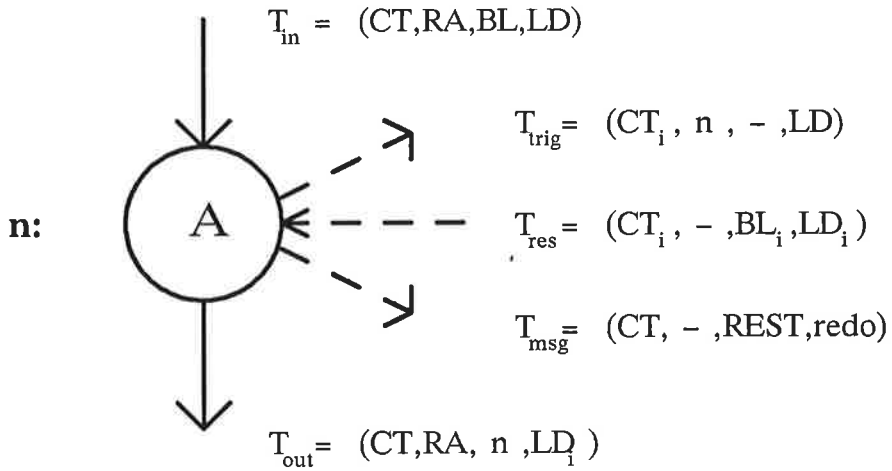
Update is used to exchange bindings of variables between subgoals. While doing so, the backtracking lists are constructed. The U–operation has two input ports, which can be classified as follows:
Tokens arriving at the *store port* represent a binding environment for a subgoal. They are stored on that port.
A token arriving at the *trigger port* contains variable bindings representing a result for a subgoal executed previously (or of the unification with the head of the clause). These bindings are added to a copy of the LD–field contained in the stored token with the same context (or in an initial token). The result is LD_3, which is given to the output token. T_{out} also contains BL_3, which is constructed by pushing BL_1 (a pointer to the subgoal that was executed previously) to the front of list BL_2.

(3) Copy



The C-operation has one input and any number of output ports. A token arriving at the input port is copied to each output port.

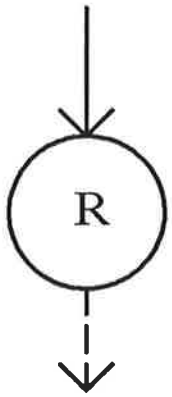
(4) Apply



The A-operation is used to manage the execution of a subgoal. A token T_{in} arriving at its input port is stored in a process control block (PCB) together with the state WAITING. The graph of each clause that can be used to persue the subgoal is triggered by a copy of T_{in}, each processing its own context and an identification of the A-operation as return address.
When the first result is returned to the A-operation, it is stored in the PCB and the state changed to EXECUTING. Furthermore, a redo message is sent to the first address in BL_i, which triggers a search for alternative solutions in the graph returned from. With LD_i and the backtracking address n, an output token is constructed to be sent to dependent subgoals. If a result T_{res} arrives at the A-operation and the according PCB is in state EXECUTING, this token is simply stored in the PCB without producing an output token. If each graph being activated by an A-operation has returned a fail-token, all alternative solutions have been calculated. Therefore, the state in the PCB is changed to SOLVED.

If the state of the PCB is WAITING and each graph activated by an A-operation has sent a fail token, backtracking has to be performed. Therefore a redo token T_{msg} is sent to the first address in BL. If BL has the form [k | REST], T_{msg} contains the list REST to be merged with the backtracking list stored at the A-operation k. Finally, the PCB managed in the A-operation n is deleted. For more detailed information about realization of backtracking in RAPiD see [Sch88].

(5) Return



T_{in} = (CT,RA,BL,LD)

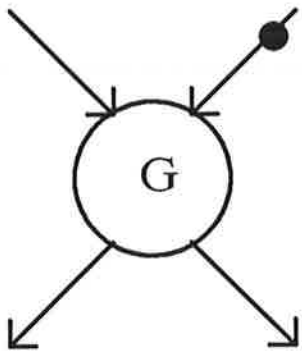
T_{res} = (CT, - ,BL,LD)

Return is the final operation in each graph activation. A token arriving at the input port is sent to the return address RA in the calling graph. On the highest level, where no return address exists, a solution for the initial goal has been constructed.

(6) Ground-Test

T_{trig} = (CT,RA,BL,LD)

T_{store} = (- , - , - ,L)

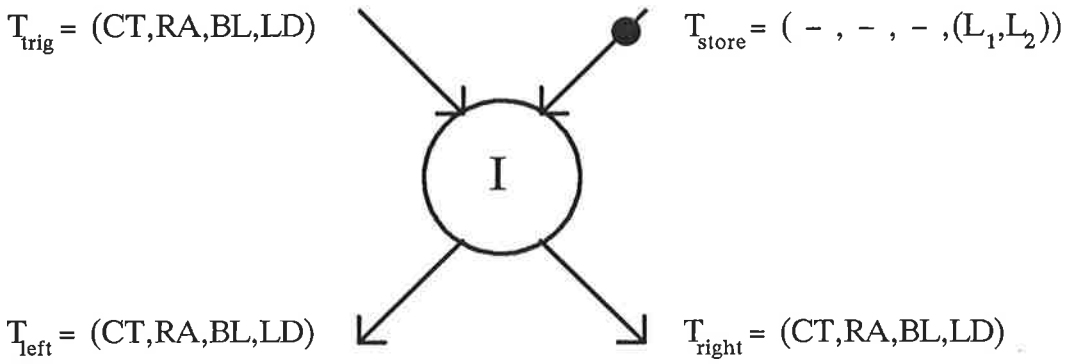


T_{left} = (CT,RA,BL,LD)

T_{right} = (CT,RA,BL,LD)

The G-operation has two input and two output ports. For a token arriving at the trigger port a test is done if all variables of list L are bound to ground values (see [DeG84]). In this case, T_{trig} is sent to the right output port, otherwise to the left. T_{store} resides permanently on the right input port.

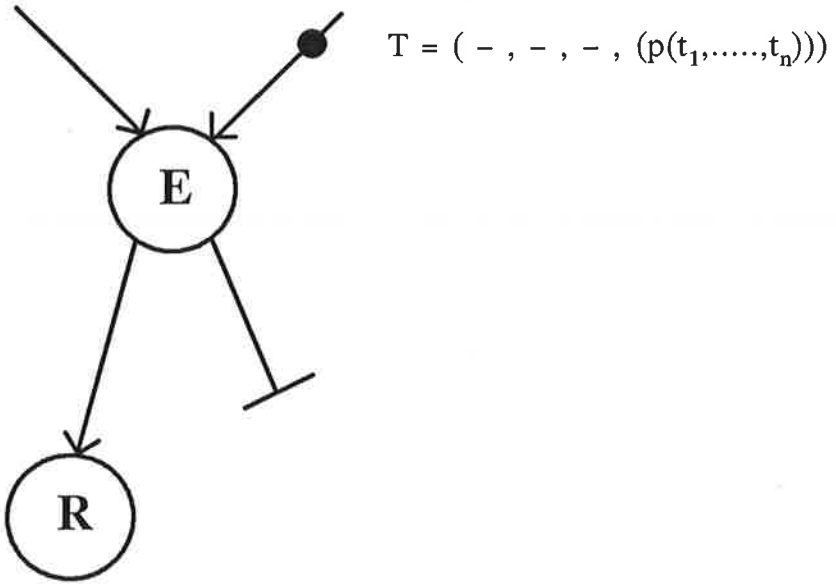
(7) Independence–Test



The I–operation has two input and two output ports. For a token arriving at the trigger port a test is performed to check, if a variable of list L_1 is bound to a variable of list L_2 (see [DeG84]). The lists L_1 and L_2 reside permanently in a token on the right input port. If a binding exists, the input token is sent to the left output port, otherwise to the right output port.

3.3. Data Flow Graph Representation of Facts and Simple Rules

Given the specification of the semantics for the basic operations, we now concentrate on the graphs to be constructed. A fact ($p(t_1,...,t_n)$.) is represented in RAPiD by the following graph:

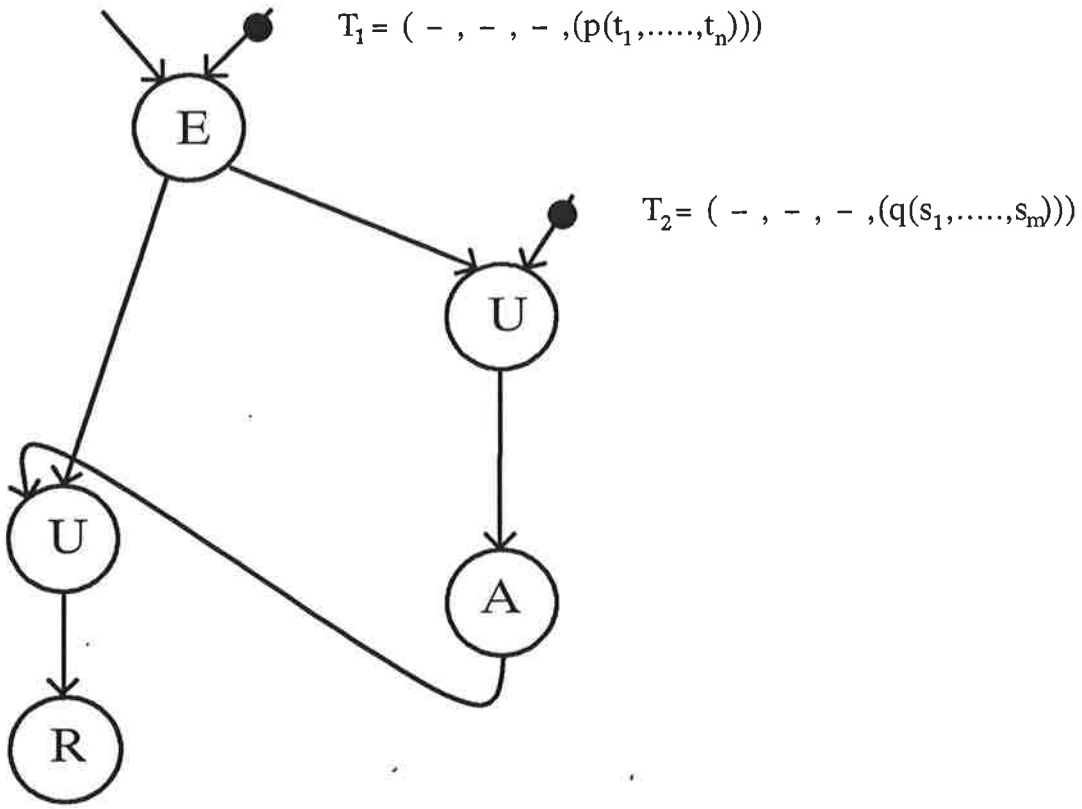


The execution of a fact starts with a unification of a token representing a (sub)goal and the token T. If the unification succeeds, the result is returned to the calling graph.

The data flow graph representing the rule

$p(t_1,\ldots,t_n)$:- $q(s_1,\ldots,s_m)$.

is as follows:



Compared with the graph of a fact, it contains an additional U-operation to obtain the required bindings constructed during unification to the binding environment for the subgoal to be derived. This U-operation is followed by an A-operation that manages the execution of the subgoal. Variable bindings constructed during this process are passed to another U-operation (on the left hand side of the pictured graph) to complete the binding environment of the token that triggered the graph.

4. Construction of Data Flow Graphs for Rules

The simple graphs presented in the previous section do not exploit any AND-parallelism. In this section, we describe the construction of data flow graphs for rules containing more than one subgoal. To detect AND-parallelism in our model, a RAP-like dependency analysis between the subgoals of a clause is used. First, we give a complete classification of possible dependency situations between two subgoals [ScB86]. Next, we describe a simple algorithm that constructs a data flow graph for a given clause. Finally, we show that our graph representation allows a more efficient execution than the conditional graph expressions (CGEs) used by DeGroot [DeG84] and Hermenegildo [Her86].

4.1. Classification of Dependency Situations

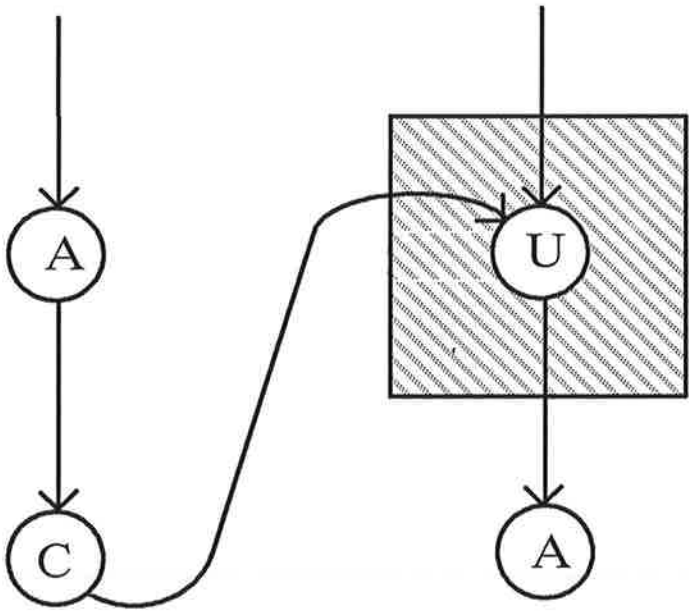
The classification given below is based upon the two kinds of dependencies studied in 2.1. In our model, independencies of subgoals can be detected by the two test-operations (G, I) introduced in the previous section. While constructing a graph for a given rule, pairs of subgoals are examined (from left to right), where the following five cases are distinguished:

(a) **Unconditioned Dependency**
This can be demonstrated by the clause

grandfather(X,Y) :- *father(X,Z)* , *parent(Z,Y)*.

In those cases, the two subgoals to be examined have a new temporary* in common. Given such a situation, the 'second' subgoal needs the binding initiated by the 'first'. No dependency test is necessary.

The following subgraph realizes the dependency between the two subgoals:



(b) **Ground-Test Dependency**
The following example demonstrates this situation:

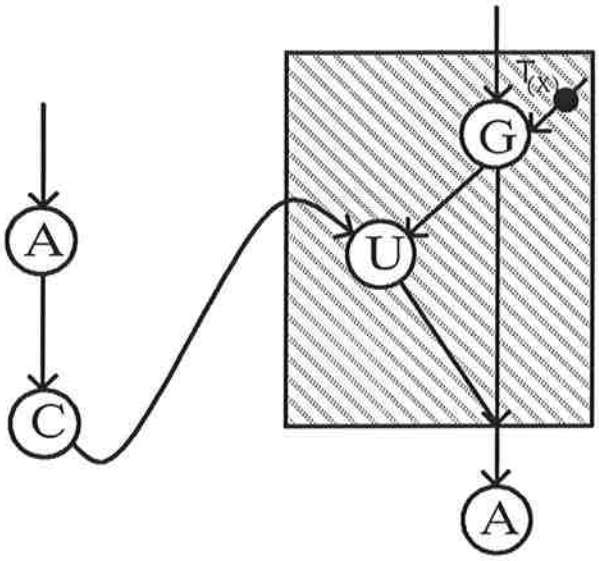
father(X,Y) :- *parent(X,Y)* , *male(X)*.

Here, case (a) does not apply and
common variables exist but
the term lists of the subgoals do not both contain other variables.
Therefore a ground-test for the common variables has to be performed to detect the dependency.

---

* A temporary is a variable not appearing in the head of the rule.

In this case, the following subgraph is used:

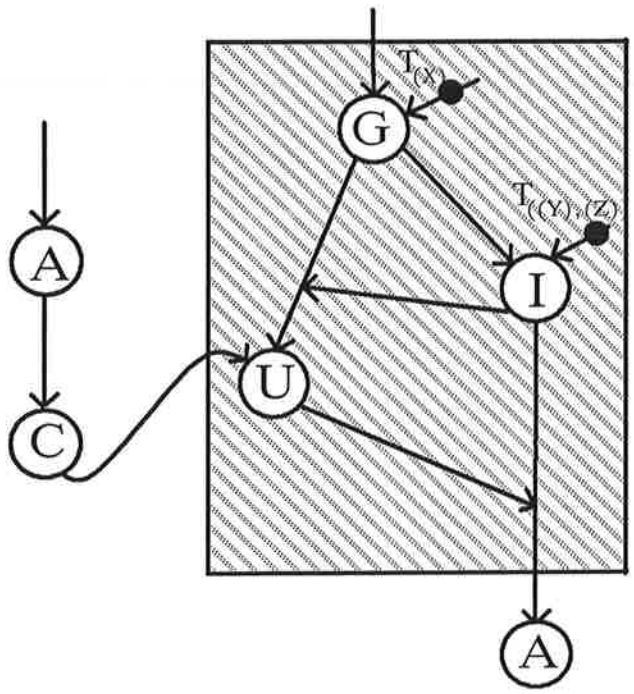


(c) **Ground/Independence-Test Dependency**
This case can be exemplified by the clause:

parents(X,Y,Z) :- *father(Y,X)* , *mother(Z,X)*.

Here, case (a) does not apply and
common variables exist and
both termlists contain other variables.
Therefore a test must be performed to see, whether the common variables have ground values before the execution of the first subgoal is started. If this is true, a binding between other variables leads to a dependency between the subgoals.

This kind of dependency is realized by the following subgraph:

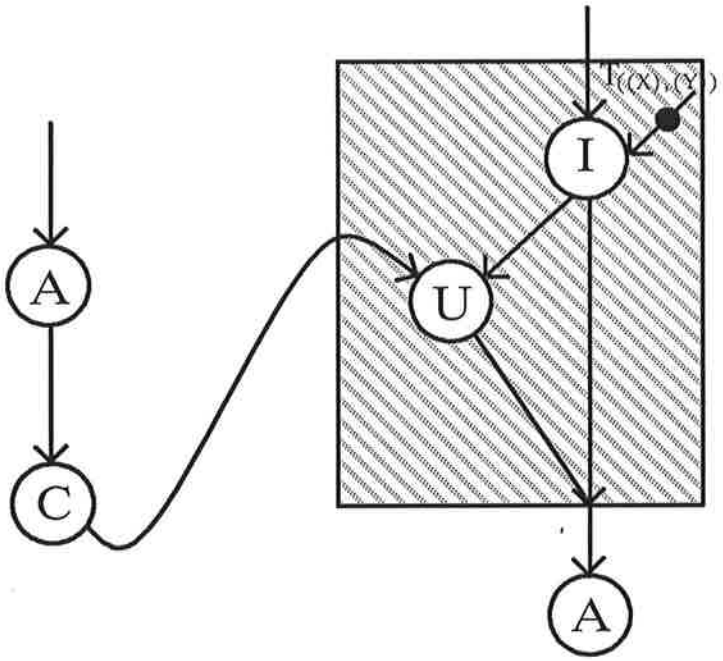


(d) **Independence-Test Dependency**
This situation can be demonstrated by the rule:

can_marry(X,Y) :- *man(X)* , *woman(Y)*.

If two subgoals have no common variable but
variables (besides new temporaries) appear in both termlists
then an independence-test is done to state whether the execution of the second subgoal can be started earlier.

In a situation like that, a subgraph of the following form is used:



(e) **Unconditioned Independency**
The first and second subgoal of the following rule demonstrate this case:

orphan(X) :- *dead(Y)* , *dead(Z)* , *parents(X,Y,Z)*.

This situation occurs
if two subgoals have no common variables nor
do both of them contain other variables (apart from new temporaries).
These subgoals are independent and may be executed concurrently without any further test.

It can easily be verified [Sch88] that the five cases given above are disjoint and cover all posibilities.

4.2. Construction of Data Flow Graphs (DFGs) in RAPiD

The classification introduced above is used by a simple algorithm to compile clauses into data flow graphs. The algorithm works through a clause from left to right, constructing a graph in which dependent subgoals are executed in that order.

For a given clause the **algorithm** performs the actions given below:

1. Construct an E–node for the head of the clause.

2. For each subgoal in the body of the clause perform the following actions:

   2.1. Create a U–node to initialize the binding environment for the subgoal. Connect its left input with the port that distributes the right output of the E–node. (Usually this is a C–node constructed when the second subgoal is handled.)

   2.2. With each subgoal on the left of the actual subgoal, perform a dependency classification. While doing this, construct a sequence of the appropriate subgraphs (marked |\\\\\| in 4.1.).

   2.3. Construct an A–node that manages the execution of the subgoal, followed by a C–node to distribute the results. Finally, an U–node (left input) is used to assemble the results in the environment of the (sub)goal that triggered the clause. Connect each of these U–nodes (right input) with the output of the appropriate U–node constructed for the previous subgoal. (The U–node for the first subgoal is connected to the left output of the E–node.)

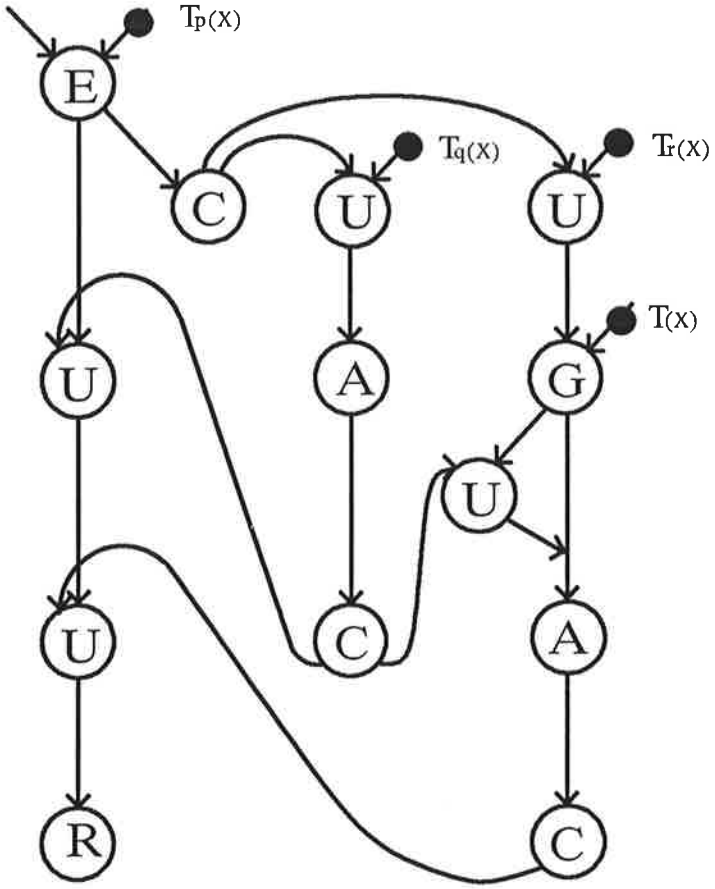
3. Construct a final R–node to be connected with the last U–node produced in 2.3.

To illustrate the process of constructing a graph, look at the data flow graph produced for the rule

$$p(X) \;:-\; q(X)\;,\; r(X).$$

which is shown below.

To get better performance, the graphs produced by this simple algorithm can be optimized. For further details about this optimization see [Sch88].



4.3. Comparison: CGEs versus DFGs

The example given above can also be used to demonstrate advantages of DFGs over CGEs. The CGE representation of the given clause is

p(X) :- (ground(X) | q(X) & r(X))

In this expression, the ground-test is performed before the subgoals are executed. If the two subgoals are dependent, the execution of the rule will be slower than in a sequential execution model. On the other hand, our DFGs allow the execution of the first subgoal to be started immediately. The ground-test can be performed concurrently. Depending on its result, the second subgoal may be started earlier.

Furthermore, optimized DFGs can be constructed automatically. Graphs are a more natural way of representing dependencies than nested, paranthesized expressions. Thus, it is no surprise that for some optimized DFGs no equivalent CGEs exist (see [Sch88]). Therefore, the loss of AND-parallelism (and a higher number of useless recalculations) caused by this kind of representation [DeG84] can be avoided if DFGs are used. The differences are evidenced by results of tests listed in section 6.