Control-Flow and Dataflow Dependencies in And-Or Tree Rewriting Systems

Salvador Abreu   Luís Moniz Pereira

Departamento de Informática
Universidade Nova de Lisboa
Quinta da Torre, 2825 Monte de Caparica – PORTUGAL
+351-1-295-4464
`spa,lmp @fct.unl.pt`

Abstract: Logic Programming systems based on the Extended Andorra Model (EAM,) or on the Agents Kernel Language (AKL) – designated as And-Or Tree Rewriting Systems (AORSs) – propose a novel and interesting approach to dealing with non-determinism in a framework that is both more general than Prolog's and appears promising for the exploitation of parallelism. We propose a framework whereby the execution strategy of such systems may be improved through the use of information on dependencies among goals and constraints.

Keywords: Extended Andorra Model, Constraint Logic Programming, AKL, Concurrency, Intelligent Backtracking

Prolog systems explore a goal's solution space using a procedure derived from SLD resolution, which results in a run-time organization which is well established, relying on 3 basic data areas: the *heap*, the *stack* and the *trail*. This has led to the emergence of very efficient execution models for Prolog, usually based on Warren's Abstract Machine (the WAM.) Prolog's backtracking has been kept simply chronological (i.e. "dumb",) in order to satisfy the requirement that it be sufficiently lightweight for a general-purpose (logic) programming language to include it as a basic mechanism.

This has not deterred researchers from exploring the potential of "intelligent" (i.e. non-chronological) backtracking, and there are approaches, such as that of [11], which acheive reasonable speedups at little extra overhead for general programs.

With the advent of multiprocessor systems, logic programming has proven an adequate vehicle to exploit and express parallelism, as witness the Aurora [19], Muse [4], &-Prolog [16], DDAS [22] and Andorra-I [12, 13] systems on one hand, and the committed choice languages such as Parlog [7], the Concurrent Prolog family [21], GCH [23] and KL/1 [24] on the other. In between these two families lies AKL [17], which retains don't-know nondeterminism while providing don't-care nondeterminism as well, in a general constraints framework.

We present the *Hypothesis Engine* (HE), a new approach to the issue of making use of dependency information computed at run-time, allowing it to direct the execution of logic programs in various computational models, including that of AKL. The HE maintains an entity called the *Hypothesis Graph*, which contains all the information pertaining to the two forms of dependencies that may occur between goals in a program's execution:

that due to the flow of control (i.e. a representation of the goal-subgoal relation) and that related to the flow of data (constraint operations on variables). We claim that the HE can be useful to provide run-time speedups in combinatorial search problems, by providing an *a-posteriori* search-space pruning mechanism, which complements the *a-priori* pruning brought by constraint propagation and the Andorra Principle of sidetracking. The Hypothesis Engine comes as a development of the AKL/IP work presented in [1].

This article is structured as follows: in section 2 the notions of hypothesis graph, control flow and dataflow hypotheses are introduced and illustrated. Section 3 describes the Hypothesis Engine, a device that manipulates hypotheses. In section 4 we discuss how the hypothesis engine may behave in the occurrence of failure, and section 5 shows how the Hypothesis Engine may be specialized for both Prolog and AKL.

The techniques of Intelligent Backtracking (IB), examples of which are [8, 9, 10, 6, 20, 15], apply to resolution-based systems such as Prolog. The theory of IB is well developed and there are extensions thereof that cover variations on the Prolog model, such as Constraints and various forms of And-parallel execution as well as co-routining and suspension of goals.

An And-Or Tree Rewriting System such as AKL, which is a system with no backtracking, requires an altogether different approach to the issue of making use of the binding-dependency information as an aid in speeding up the search of the solution space. This, together with the need to accomodate a more general constraint system than unification over first-order terms, prompted us to distinguish two forms of dependency:

- Control Flow dependency. This conveys information pertaining to the execution of a goal, and reflects relations such as *ancestor* or the multiple alternatives for a given goal.
- Dataflow dependency. This kind of dependency provides information on producer/consumer relations between goals, and is used to reflect the functioning of constraint operations.

These two forms of dependencies may be represented with the notions of *Control Flow Hypotheses* (CFH) and *Dataflow Hypotheses* (DFH.) These may be characterized as follows:

- Control Flow Hypotheses represent execution *Control Points*. These are the various representations a goal may be associated with. For example, in Prolog these would be choice-points, in AKL they would be and-boxes and choice-boxes.
- Dataflow Hypotheses stand for the occurrence of a constraint operation on a *variable*.

Control Flow Hypotheses are intended to represent the relations that occur among goals occurring in a computation and are the constituents of the *Hypothesis Graph* (HG), which is defined through the dependency and support relations over H, the set of all CFHs, as follows:

- The *immediate dependency* relation $\succ$ and its inverse, the *immediate support* $\prec$.
- The *structural dependency* relation $>$ and its inverse, the *structural support* $<$. These are the transitive closures of $\succ$ and $\prec$ respectively.

These relations model the goal-subgoal relationship. Let $h \in H$, the following functions are also defined:

- The *alternatives* of a hypothesis h, written $[h]$. $[h] \subset H$ holds for all h. $[]$ is an equivalence relation over H.
- The *alternative representative* of a hypothesis h, written $|h|$. $|h| \in [h]$ is true for all h.

These functions reflect the calling goal vs. the (instantiated) goal in each matching clause.

Consider the example program of figure 1, in which the predicates p, q and r all have more than one clause. By the time execution of the program is within the body of r's first clause, the associated hypothesis graph will look like that of figure 2, which illustrates the various relations on CFHs:

- Each CFH is represented by a circle.
- The shaded areas stand for the *alternatives* relation, $[h]$. For example $[q_2] = \{q, q_1, q_2\}$.

$$\begin{array}{ll} p \leftarrow \cdots q \cdots & p \leftarrow \cdots \\ q \leftarrow \cdots r \cdots & q \leftarrow \cdots \\ r \leftarrow \cdots & r \leftarrow \cdots \end{array}$$

Figure 1: Program for Hypothesis Graph

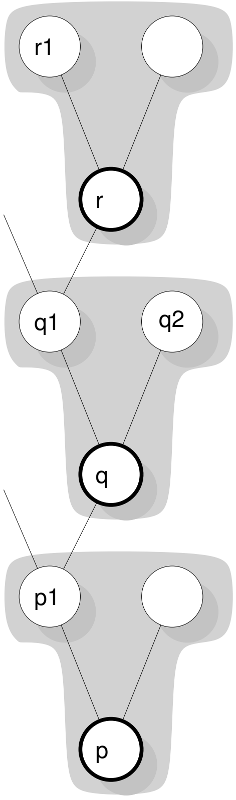


Figure 2: Hypothesis Graph

- The CFHs with a thicker outline represent the *alternative representatives* for every CFH h in their *alternatives* class. For example $|q_1| = q$.
- The lines represent the *immediate dependency* relation. For example $h_p \prec h_{p_1}$.

Notice the great similarity between this representation of the Hypothesis Graph and the structure of an AKL configuration (i.e. an and-or tree).

Dataflow Hypotheses represent the occurrence of a constraint operation on a variable. A DFH d is always associated with a specific control flow hypothesis h, designated as its (control flow) *hypothesis*: $\exists!\ h = \mathbf{hyp}(d)$. There are three types of DFHs:

- *producer* DFH ("**prod**"): these indicate that a new constraint was imposed on a variable, this comes about as the result of a Tell constraint operation.
- *consumer* DFH ("**cons**"): these indicate that a constraint on a variable was successfully tested, for instance by means of a Tell or Ask constraint operation.
- *failing* DFH ("**fail**"): these indicate that a constraint operation (either kind) failed.

A **prod** DFH must specify a variable, while **cons** and **fail** DFHs always refer to another (**prod**) DFH.

Refining a little bit the example of figure 1, and introducing variables in the predicates: assume that the first clauses for p, q and r are now those shown in figure 3. Goal p now produces a value[1] for X, q consumes

$$\begin{array}{l} p(X) \leftarrow \cdots X = a, q(X) \cdots \\ q(X) \leftarrow \cdots Y = f(X), r(Y) \cdots \\ r(Y) \leftarrow \cdots Y = f(b) \cdots \end{array}$$

Figure 3: Program with variables

this value and produces one for Y, while r consumes the value for Y and fails because of the value of X.

The updated hypothesis graph, including DFHs, is shown in figure 4, in which a CFH and its associated

[1] In the text, "a value for X" should be considered as equivalent to "a constraint on X."

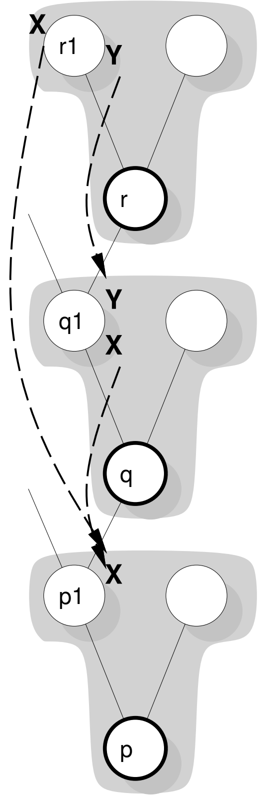
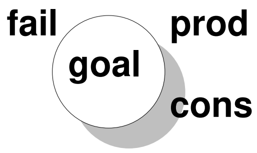


Figure 4: Hypothesis Graph with DFHs

DFHs are rendered as follows:



Where **prod**, **cons** and **fail** are all functions from the set of all CFHs H into the powerset of the set of DFHs, 2^D, and have the meanings previously described.

With respect to figure 4, the dashed lines represent the references from **cons** and **fail** DFHs to the corresponding **prod** CFHs. Notice that CFHs which are alternative representatives have no associated DFHs.

The issues of how to construct the hypothesis graph, and how to make use of it in a computation have not yet been addressed. Our goal is to provide a solution to these problems which can be easily adapted to different computational models, which provides sufficient generality to accomodate various constraint systems, while retaining a clean structure to allow for ease of experimentation and development.

Such is the purpose of the *Hypothesis Engine*, which we now describe.

In order to accomodate the view that Hypotheses (both Control Flow and Dataflow) are part of a logic programming system, the latter can be presented as being split in three components, as shown in figure 5. These

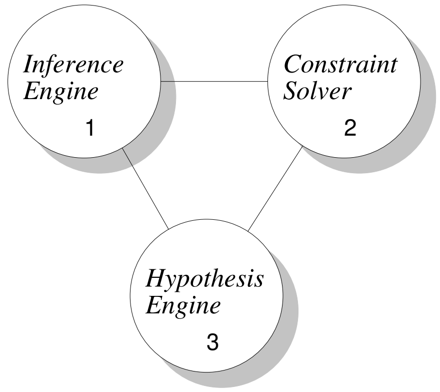


Figure 5: Structure of a LP System

components may be characterized by their function in the Logic Programming system, as follows:

1. The *Inference Engine* (IE) is responsible for the flow of control, by handling the creation and destruction of execution control points. It has access to the program code.

2. The *Constraint Solver* (CS) is responsible for the handling of variables and constraints over these. It has access to the constraint store.

3. The *Hypothesis Engine* (HE) is responsible for the maintenance of the Hypothesis Graph, and is the

entity that manipulates both Control Flow and Dataflow hypotheses.

Each of these components communicates with the other two.

The Hypothesis Engine (HE) is a separate entity, which operates on abstrations of:

- The computational model and the control strategy: it may be used with either resolution-based systems such as Prolog or with And-Or Tree Rewriting Systems such as AKL.
- The constraint domain: it may operate, for instance, with unification over first-order terms and finite domains.

It is responsible for the maintenance of the Hypothesis Graph (see the section 2), and is interfaces to both the Inference Engine and the Constraint Solver through an operational interface, which can be used both to add and extract information from the Hypothesis Graph.

The structure of the Hypothesis Engine is as follows:

There may be more than one instance of the HE, each one being known as a *Hypothesis Task* (HT.) Each HT is associated with exactly *one* instance of the Inference Engine (in a parallel Logic Programming system where there is more than one "worker," each one will have an associated Hypothesis Task.)

Besides manipulating the Hypothesis Graph, each HT has a state which is specified by the *current control flow hypothesis* CH, as well as a few auxiliary data structures.

Each HT executes a stream of operations, originating from the Inference Engine and the Constraint Solver.

The label L which is mentioned below is always a referent to an Inference Engine *control point*, e.g. a Prolog choice-point or an AKL and- or choice-box.

These operations are now listed:

CFH Construction Operations are intended to be invoked by the Inference Engine, and their purpose is to declare the relations between CFHs, thereby defining the structure of the hypothesis graph:

Operation	Description
new_h	Create a new empty hypothesis (and keep it in CH).
delete_h	Remove current hypothesis from HG.
select_h L	Sets CH to label L's hypothesis.
label L	Associate label L with current hypothesis.
asup L	Add hypothesis labeled L as a supporter for current hypothesis.
aalt L	Add current hypothesis to the one labeled L's alt.

CFH Retrieval Operations return the appropriate value to their caller:

Operation	Description
gcur	Get CH's label.
galt	Get alternatives of CH.
gdep	Get immediate dependencies of CH.
gsup	Get immediate supporter of CH.

DFH Construction Operations apply to the current control flow hypothesis, CH:

Operation	Description
aprod V	Produce constraint on variable V.
acons V	Consume constraint on variable V.
afail V	Fail constraint on variable V.

These operations are normally invoked by the Constraint Solver, as a result of executing a constraint operation.

Combined CFH and DFH Operations include:

Operation	Description
relax	Remove DFH's associated with CH.
gsusp	Compute the suspect set for CH.
gculp	Select a culprit from the suspect set (becomes CH).
gactions	Query the HE for a sequence of actions to perform.

These operations are invoked by the Inference Engine, after a sequence of Constraint operations which ended abnormally. By "abnormal" it is meant either a constraint failure (eg. after a series of **afail** operations, which will precede an action from the second group of the table) or because the execution suspended, in which case the operation will be **relax**.

The purpose of the Hypothesis Engine is to act as an advisor to the Inference Engine when an deviation from its normal flow of control occurs, for instance when there is a constraint failure. The purpose of the **gactions** operation previously mentioned, is to provide the Inference Engine with the appropriate action resulting from the Hypothesis Engine's analysis of the failure.

The outcome of the execution of the **gactions** operation is a sequence of actions to be performed by the Inference Engine, which include:

$$\begin{array}{lcl} \mathit{action} & \longrightarrow & \mathbf{remove}(L) \\ \mathit{action} & \longrightarrow & \mathbf{reorder}(L_1, L_2) \\ \mathit{action} & \longrightarrow & \mathbf{proceed}(L) \\ \mathit{action} & \longrightarrow & \mathbf{fail}(L) \end{array}$$

Where L, L_1 and L_2 are references to Inference Engine control points.

The determination of the most appropriate sequence of actions that should be performed in response to a specific failure has two aspects:

- The culprit[2] selection.
- The type of action to perform, based on the culprit.

Culprit Selection. In order to be able to say what the *culprit* is, a few definitions are required. A more detailed discussion of these issues may be found in [3].

The *relevant support* of a CFH h is informally defined to be the set of CFHs that can "collectively" be held accountable for the occurrence of h in the computation. This amounts to doing the transitive closure of the immediate support relation $\prec$, applied to the initial set $\mathbf{hyp}(\mathbf{cons}(h) \cup \mathbf{fail}(h))$.

The *suspect set*, intuitively, is the set of CFHs which may be responsible for a failure occurring in a CFH h. This is defined to be equal to the relevant support of h.

Finally, the *culprit* is defined to be one member of the *suspect set*, which is selected by the Hypothesis Engine as *the* CFH that can be "blamed" for the failure. There are various criteria that the Hypothesis Engine can adopt in order to select one element of the suspect set rather than another. Examples of these criteria require that the culprit be:

- Any member of the suspect set. This approach is suitable for And-Or Tree Rewriting Systems such as AKL, because it does not endanger the completeness of the search of the solution space. Selecting an arbitrary element of the suspect set may however change the order in which solutions are generated, w.r.t. the standard execution strategy.

- The maximal element of the suspect set (it always exists and is unique, by definition.) This approach is suitable for languages such as Prolog, in order to maintain completeness. Note that this selection strategy may lead to the same type of behavior that Intelligent Backtracking techniques such as that of [11] exhibit.

Expanding on the first criterion ("any member..."), and when using an AORS Inference Engine, the selection may be performed on the basis of a heuristic aimed at getting better performance. Let S be the suspect set, $h_f \in S$ be the CFH where the failure occurred and $h_c \in S$ be the culprit CFH:

C1) Select the CFH $h_c \in S$ which has the least number of alternatives (i.e. $[h_c]$ is minimal), and then either:

  - Proceed from h_c[3] without performing modifications to the configuration.
  - Proceed from h_c after reordering h_c's aliases[4] to the front of their siblings.

  This criterion amounts to selecting the culprit with the least expected cost, as it has less alternatives.

C2) Increment h_f's fail count, select the CFH $h_c \in S$ khown to have failed most often in other branches (i.e. with the highest fail count,) and then proceed as with C1. This approach leads to a behavior similar to the first-fail heuristic, as applied to AKL. This is described in [2].

[2] The intuition for the term "culprit" is that it is the control flow hypothesis (recall that CFHs are directly mapped to control points in the Inference Engine) which is to be held responsible for the failure.

[3] "Proceed from h" is a notational abuse because h is *not* an Inference Engine control point, however, it is clear that this means "proceed from the control point L such that $L = \mathbf{label}(h)$.

[4] The notion of "alias" is related to that of "re-incident node" in the notation of [2], and represents the set of configuration nodes which are copies of h_c occurring as the result of a nondeterminate promotion transformation.

As presented, the Hypothesis Engine is not specifically tied to any particular computational model, although it is well suited to AORSs such as AKL. However, the design of the Hypothesis Engine was done so as to allow for the basic model to be extended to fit specific computational models, such as AKL's or Prolog's.

AKL Specialization. The AKL Inference Engine has a few idiosyncrasies that must be dealt with. The previously introduced HE's operational interface is extended to constitute the AKL Hypothesis Engine, $\mathrm{HE}_{\mathrm{AKL}}$:

Operation	Description
alias L	Mark CH as a copy of L's CFH.
reloc L_1, L_2	L_2 is L_1's new name.

- The **alias** operation associates CH with another CFH, establishing that they're *aliases*, i.e. other instances of the *same* goal occurring in different branches of an or node (choice-box). This is similar to the *re-incident node* notion of [2].

- The **reloc** operation is used during garbage collections and in determinate promotions. It has to do with control points being moved in the IE's memory, and is a device necessary to keep the Hypothesis Engine separate.

Prolog Specialization. The computational model for Prolog (without suspension) brings about a restricted form of Hypothesis Graph because:

- There is only one active alternative for each disjunction at any time, i.e. only one clause is active for each goal.

- Execution within a conjunction is performed sequentially.

The result is that the Hypothesis Graph can be represented as a list of CFHs, each associated with a choice point, which are the control points of a Prolog inference engine.

The WAM's temporary (X) registers require special treatment: these are viewed as aliases for actual variables. This is necessary because terms in registers may be used in constraint operations.

To implement this, the HE provides a set of *alias registers*, corresponding to the WAM's registers, as well as operations to manipulate them. These are described in [3].

We briefly presented a framework whereby knowledge about the past execution of a program may be used to influence its further execution, relying on two distinct features: control flow information and dataflow information.

The approach was shown to be sufficiently general to accomodate both traditional implementations of Prolog and And-Or Tree Rewriting Systems such as AKL.

The granularity at which the Hypothesis Engine operates is entirely up to the other two components of the logic programming systems it is integrated in. This feature, which is made possible by the well-defined operational interface, allows for a more controlled use of the HE's features and consequent overhead.

A complete system based on SICS' AKL/PS [18, 17] is currently being worked on, in which the Hypothesis Engine component is implemented in a modular fashion, allowing it to be re-used in other logic programming systems.

It is in our plans to investigate the usefulness (or lack thereof) of the Hypothesis Engine in other settings, such as refining the detection of independence between variables as in &-Prolog's CGEs, or its use as a complementary device to static analysis techniques.

[1] Salvador Abreu and Luís Moniz Pereira. Design for AKL with Intelligent Pruning. In Roy Dyckhoff, editor, *Extensions of Logic Programming*, pages 3–10, Heidelberg, March/April 1993. University of St. Andrews, Scotland, Springer-Verlag.

[2] Salvador Abreu, Luís Moniz Pereira, and Philippe Codognet. Improving backward execution in the andorra family of languages. In Apt [5], pages 384–398.

[3] Salvador Pinto Abreu. *Improving the Parallel Execution of Logic Programs*. PhD thesis, Universidade Nova de Lisboa, June 1994.

[4] Khayri A. M. Ali and Roland Karlsson. The muse Or-Parallel Prolog model and its performance. In Saumya Debray and Manuel Hermenegildo, editors, *Proceedings of the 1990 North American Conference on Logic Programming*, pages 757–776, Austin, 1990. ALP, MIT Press.

[5] Krzysztof Apt, editor. *Proceedings of the Joint International Conference and Symposium on Logic Programming*, Washington, USA, 1992. The MIT Press.

[6] M. Bruynooghe and L. M. Pereira. Deduction revision by intelligent backtracking. In J.A. Campbell, editor, *Implementations of Prolog*. Ellis-Horwood, 1984.

[7] K. Clark and S. Gregory. PARLOG : Parallel programming in logic. *ACM TOPLAS*, 8 (1), January 1986.

[8] Christian Codognet, Philippe Codognet, and Gilberto File. Yet another intelligent backtracking method. In Robert A. Kowalski and Kenneth A. Bowen, editors, *Proceedings of the Fifth International Conference and Symposium on Logic Programming*, pages 447–465, Seatle, 1988. ALP, IEEE, The MIT Press.

[9] Chritian Codognet. *Backtracking intelligent en Programmation Logique–une étude théorique.* PhD thesis, Université de Paris VII, 1988.

[10] Philippe Codognet. *Backtracking intelligent en Programmation Logique–de la théorie à l'implémentation et à l'application au parallélisme.* PhD thesis, Université de Bordeaux I, 1988.

[11] Philippe Codognet and Thierry Sola. Extending the WAM for intelligent backtracking. In Furukawa [14], pages 127–141.

[12] Vítor Santos Costa, David H. D. Warren, and Rong Yang. The Andorra-I engine: A parallel implementation of the basic andorra model. In Furukawa [14], pages 825–839.

[13] Vítor Santos Costa, David H. D. Warren, and Rong Yang. The Andorra-I preprocessor: Supporting full Prolog on the basic andorra model. In Furukawa [14], pages 443–456.

[14] Koichi Furukawa, editor. *Proceedings of the Eighth International Conference on Logic Programming*, Paris, France, 1991. The MIT Press.

[15] William S. Havens. Dataflow Dependency Backtracking in a New CLP Language. In *AAAI Symposium on Constraint-Based Reasoning*, Stanford University, March 1991.

[16] M. V. Hermenegildo and K. J. Greene. &-Prolog and its performance: Exploiting independent And-Parallelism. In David H. D. Warren and Peter Szeredi, editors, *Proceedings of the Seventh International Conference on Logic Programming*, pages 253–268, Jerusalem, 1990. The MIT Press.

[17] Sverker Janson. *AKL – A Multi-Paradigm Programming Language.* PhD thesis, Uppsala University, 1994.

[18] Sverker Janson and Johan Montelius. Design of a sequential prototype implementation of the andorra kernel language. Technical report, SICS, September 1991. (draft).

[19] Ewing Lusk, David H. D. Warren, Seif Haridi, et al. The Aurora OR-parallel Prolog System. Technical report, Gigalips Project, 1988.

[20] Stanislaw Matwin and Tomasz Pietrzykowski. Intelligent Backtracking in Plan-based Deduction. Technical Report TR-82-08, University of Ottawa, 1982.

[21] E. Shapiro. The family of concurrent logic programming languages. *ACM computing surveys*, 21(3), september 1989.

[22] Kish Shen. Exploiting dependent and-parallelism in Prolog. the dynamic dependent and-parallel scheme (DDAS). In Apt [5], pages 717–731.

[23] K. Ueda. Guarded horn clauses. Technical report, ICOT, 1985. TR-103.

[24] K. Ueda and T. Chikayama. Design of the Kernel Language for the Parallel Inference Machine. *Computer Journal*, 33 (6), 1990.