The Eventflow Model - A Concept for Real-time Control of Intelligent Autonomous Systems

A. Burg
Fakultät für Mathematik und Informatik
Universität Passau, 94030 Passau, F. R. Germany

MIP 9713

Abstract

Autonomous systems need control that on the one hand is fast enough to satisfy real-time conditions and that on the other hand is flexible enough to navigate according to selective mission goals in a variety of environments. The only way to guarantee this is to combine the calculation of mathematical formulas that are responsible for the basic behaviour with the evaluation of rules that represent the knowledge-based (intelligent) decision part of the control system in a way that parallel processing can be employed most efficiently. Prolog, a generally used tool for specifying knowledge by a set of rules, however, lacks, even in parallel forms, of performance.

Therefore, this paper presents a new data-flow model to make knowledge-based control suited for execution on a highly parallel dataflow architecture.

The model does not restrict evaluation parallism in any way, i.e., the same amount of parallelism is exploited as by the (restricted) AND/OR-evaluation models of logic programming. Control efficiency is increased by the model's purely data-driven evaluation scheme that saves any distribution of demands-for-evaluation. Furthermore, activity-driven evaluation allows for a much higher complexity of rules.

The key ideas of this report will be found in

- *"Dataflow Computing and Embedded Real-World-Simulation for High-Performance Control of Autonomous Systems", Proc. of the European Simulation Symposium, ESS'97, Passau, Germany, Oct. 1997.*
- *"Activity-Directed Dataflow Computing for Real-time Control of Intelligent Autono mous Systems", accepted for the Proc. of Int. Conf. on Computational Intelligence and Multimedia Applications, ICCIMA'98, Churchill, Australia, Feb. 1998.*
- *"Activity-Directed Dataflow Computing for High-Performance Knowledge-Based Control of Autonomous Systems", accepted for the Proc. of Int. Computer Science Conventions, ICSC'98, Tenerife, Spain, Feb. 1998.*

Contents

1 Introduction

Time frame after time frame, it is the general task of control for highly autonomous systems to sample sensor data at the beginning and to generate data for output to the controlled system at the end of each time frame.

To handle their complexity, these control systems should be organized hierarchically due to their different demands [4]: Lowest-level control must be fast and calculate exactly to guarantee rapid, reflex-like reactions when necessary. Therefore basic behaviour is mainly calculated by mathematical formulas. Higher control levels more and more coordinate fast reaction with planning according to a mission goal. Figure 1 depicts this hierarchical ordering.

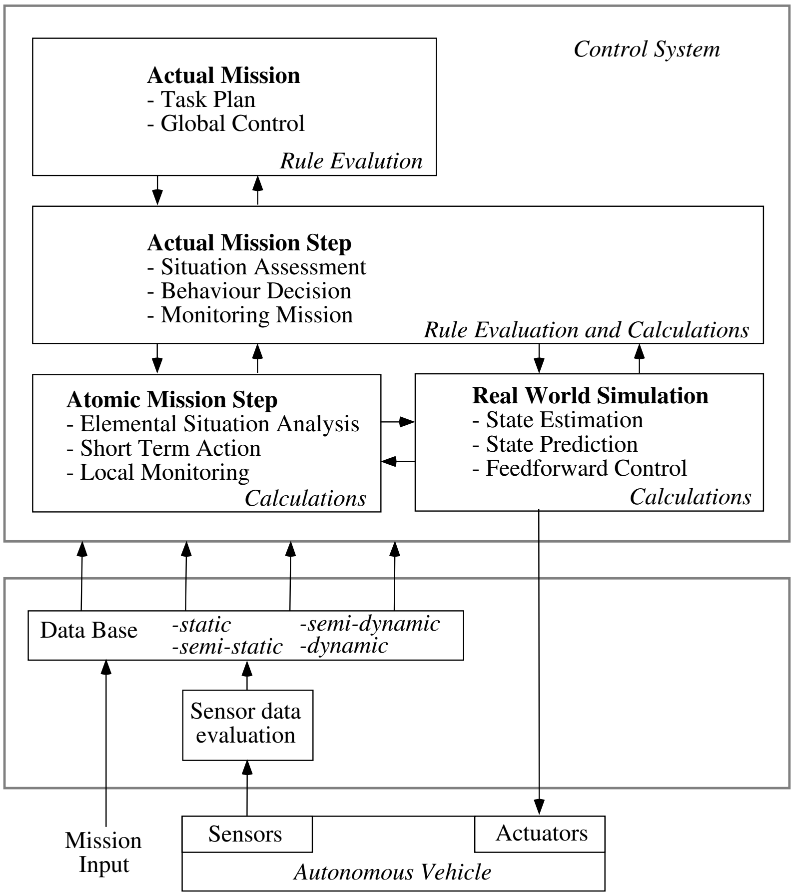


Figure 1: Hierarchical control system

Complex high-level control needs the ability for planning and deciding intelligently. Thus, an expert system must be part of high-level control. An expert system usually consists of a complex set of rules over an extensive database. For implementations, Prolog, a high level language for deductive reasoning of rules [7], is commonly used. The disadvantage of Prolog, however, is that evaluation is slow, due to its depth-first search algorithm and backtracking.

Although evaluation time decreases with parallel forms of Prolog implementations [9, 21] performance gain still does not allow for control of autonomous systems in real-time. Therefore, a strong need for increasing the performance by multi-processing under well-balanced load distribution remains.

The dataflow architecture, on the other hand, is a parallel architecture that supports high-performance execution of high-level programs compiled into semantically equivalent graphs [23, 26]. Therefore, the basic idea of this approach is to transform a Prolog-program into a dataflow graph.

In contrast to general purpose expert systems it is a special feature of knowledge-based control systems to impose a fixed set of queries on repeatedly sampled input (sensor) data. This fact makes it possible to compile a Prolog-program into a dataflow graph, where the performance potential of dataflow architectures allows a high repetition rate of the time frames. Furthermore, the approach allows a refinement to activity-directed computation to shorten the time frames or extending the knowledge-base for a more accurate ouput, by reducing the necessary amount of evaluations.

The paper starts with an overview of the state of the art concerning Prolog and dataflow architectures. This is followed by an outline of the conditions a knowledge-based control system must obey in contrast to a general-purpose Prolog-program, to illustrate how a new dataflow model, resulting in a more efficient type of evaluation, is possible.

The paper continues with a detailed description of the dataflow model for knowledge-based control systems. In the first step the structure of the original Prolog-program is mapped onto a structure graph, from which static data-dependencies can be obtained in a second step. Based on this information the dataflow graph can easily be developed in a third step.

Thereafter, the analysis of the input data for the control system allows to refine the dataflow model to the more efficient activity-directed model of computation.

Next, the proposed model is extended to make it suitable for calculations as well and hence for integrating low-level control within the same computation model.

Finally, the advantages of the model will be discussed.

2 State of the Art

The base concept of Prolog is logic programming, derived from a mathematical formalism for certain classes of problems that require deductive reasoning [14]. It is a high-level, non-procedural programming language requiring no specialized computer architecture, i.e., not absolutely the von Neumann architecture.

Parallelism in Prolog can easily be found at the level of rules (OR-parallelism) as well as in the rules themselves (AND-parallelism) [21], i.e., at the level of the literals of the rule, while it is not that simple to gain parallelism by splitting a Prolog-program into different processes.

By the dataflow architecture, in contrast to a von Neumann multiprocessor architecture, fine-grained parallelism in a program is immediately exploitable: A dataflow graph, the low level language for dataflow architectures, is given by a set of nodes, that are connected by directed arcs. Each node represents a separate operation element, capable of accepting, processing and distributing tokens. The arcs define paths over which tokens carrying data are asynchronously passed from one node to another [11]. Enabling a node for execution requires that all needed data is present on the input arcs. Executing a node means applying the node's function on this data. After execution, input tokens are deleted and new tokens carrying the calculation results are placed on the output arcs.

The processing of nodes is driven by the availability of the required data. Therefore, no centralized control is necessary and thus nodes are permitted to be evaluated in parallel, without explicitly specifiying the parallelism. Since each node of the graph generally represents an atomic element of the high level program from which the graph is derived, the resulting parallelism is fine-grained.

With regard to this advantage, various models for evaluating Prolog according to the dataflow model have been developed [5, 15, 17, 18, 25, 27]. However, all this work is related to general-purpose Prolog-programs, i.e., to formulate rules for answering changing queries over a mainly static set of facts. The outcome, however, usually is not really efficient, since either the graph evaluation is still demand-driven, or the resulting graph is very comprehensive.

Knowledge-based control systems work in a completely different way. In a sequence of time frames they always impose the same static set of queries, but the facts, e.g., sensor data, change dynamically.

3 Prolog

Prolog is a high level language for logic programming, that is derived from the mathematical formalism of logic [14].

A program in Prolog consists of

- a data base, in which definition are made in form of facts,

- a rule system, that derives logical consequences out of the definitions and

- queries that the program has to solve.

3.1 Logic programming

First, some introducing definitions have to be made.

Definition 3.1: ***Terms*** A term t is

i) a constant a,

ii) a variable X, or

iii) if t_1 and t_2 are terms and $\rho \in I_O$ is a operation on terms, then $t_1 \rho t_2$ is a term again.

Definition 3.2: ***Substitution, Unification***

i) $\Phi = \{X_1/t_1, ..., X_n/t_n\}$ is called substitution, if $X_1, ..., X_n$ are pairwise different variables, i.e., $X_i \neq X_j \ \forall i = 1, ..., n \ \forall j = 1, ..., n$ with $i \neq j$ and $t_1, ..., t_n$ are terms with $X_i \neq t_i \ \forall i = 1, ..., n$.

ii) Two terms are unifiable, if there exists a substitution for the variables in the terms, so that the terms get syntactically identical.

Now a logic program, that forms the base for a Prolog-program, can be defined.

Definition 3.3: ***Logic Program***

A logic program $\mathcal{L}$ can be described by a tuple $(C, L_{\mathcal{L}}, P, T_{\mathcal{L}})$, where C is a set of clauses, $L_{\mathcal{L}}$ is a set of literals, P is a set of predicates, $T_{\mathcal{L}}$ is a set of parameters, $n, h \in \mathbb{N}$ and $\forall i \in \{1, ..., n\} : m_i \in \mathbb{N}$.

i) C is defined as:
$C := \{c_s \mid s \in \{1,...,h\},\ h \geq 1,\ c_s = p^0_{i,k} \leftarrow p^1_{i,k},...,p^{l_{i,k}}_{i,k}. \vee c_s = p_{i,k}.\ i \in \{1,...,n\},\ k \in \{1,...,m_i\},\ l_{i,k} \geq 1\}$

ii) Each $p^r_{i,k} \in L_{\mathcal{L}},\ i \in \{1,...,n\},\ k \in \{1,...,m_i\},\ r \in \{0,...,l_{i,k}\}$ is called literal and is of the form $\nu(t_1,...,t_u),\ u \geq 1$, or $\nu()$.

- with $\nu \in P$ is a predicate symbol and

- $t_1,...,t_u \in T_{\mathcal{L}}$ are constants, variables or terms.

iii) The set of the literals is defined as: $L := \bigcup_{i \in \{1,...,n\}} p_i$, with p_i for $p^0_{i,1}$. Therefore,

i) by the condition $k \geq 1$ the existence of $p^0_{i,1}$ is guaranteed,

ii) $\forall i \in \{1,...,n\}\ \forall k \in \{1,...,m_i\}:\ p^0_{i,1} \equiv_L p^0_{i,k}$.
where $\equiv_L$ stands for

- the same predicate symbol,

- the same amount and type of parameters.

Furthermore it holds:
$\forall i \in \{1,...,n\}\ \forall k \in \{1,...,m_i\}\ \forall r \in \{1,...,l_{i,k}\}\ \exists p \in L:\ p^r_{i,k} \simeq p$,
with $p \simeq q$ meaning that a substitution Φ exists with $p = \Phi(q)$.

iii) $p^0_{i,k}$ is called head of the clause, $p^1_{i,k},...,p^{l_{i,k}}_{i,k}$ forms the body.

iv) A clause $p_{i,k}.$ with an empty body is called a fact.

Additional remarks:

- h is the amount of clauses in C.

- n is the amount of literals in L.

- m_i is the amount of occurances of the literal p_i as head of a clause.
Thus $h = m_1 + ... + m_n$.

- $l_{i,k}$ is the amount of literals of a clause with the head $p_{i,k}$.

Definition 3.4: ***Query to a program***

A query to a logic program $\mathcal{L}$ is a tuple (q, $\mathcal{L}$), where $\mathcal{L} = (C, L_{\mathcal{L}}, P, T_{\mathcal{L}})$ is a logic program and the query q is defined as: $\leftarrow q^1,...,q^l.$, with $l \geq 1$ and $\forall r \in \{1,...,l\}\ \exists p \in L_{\mathcal{L}}:\ q^r \simeq p$.

3.2 Evaluation Mechanism of Logic Programs

The operational semantics of

$p^0 \leftarrow p^1, ..., p^n.$

is:

if p^1 **and ... and** p^n **then** p^0 .

Informally spoken, the results calculated by the literals in the body of a rule are combined conjunctively.

If there is more than one rule with the same head, i.e., with the same predicate and the same number of parameters in the head, the rules realize alternatives, e.g.,

$p^0 \leftarrow p^1, ..., p^k.$
$p^0 \leftarrow p^{k+1}, ..., p^n..$

They represent a disjunctive combination of the rule bodies, i.e., of the results calculated by every rule. The operational semantics is:

if $(p^1$ **and ... and** $p^k)$ **or** $(p^{k+1}$ **and ... and** $p^n)$ **then** p^0..

The evaluation mechanism of a logic program works in the following way:

- The evaluation starts with the goal.
- If the rule $p^0 \leftarrow p^1, ..., p^n.$ is called, the literals in the body of the rule are applied, i.e., the rules with the heads $p^1, ..., p^n.$ are called. Their results are combined conjunctively.
- If there is only one rule with the head p^0, only this rule is called when the literal p^0 is applied.
- If there is more than one rule with the same head p^0, each of the rules is called when the literal p^0 is applied. Their results are combined disjunctively.
- A rule can only be called by a literal if the parameters of the literal can be unified with the parameters of the head of the rule, i.e., if there exists for every two corresponding parameters a substitution to unify both parameters. If there is no rule with unifyable parameters, the literal fails.
- If the rule that is called fails, the literal fails.
- If a literal in the body of a rule fails, the rule with the literal in its body fails also. Otherwise the evaluation is successful.
- The binding of a variable to a value is done by unification.

In theory, the ordering of rules does not affect the selection of the next rule to be evaluated. Rather, the selection is done non-deterministical. E.g., if there is more than one rule with the head p^0 these can be called in arbitrary ordering. In the rule

$p^0 \leftarrow p^1, ..., p^n$. the literals p^1 to p^n are also applied non-deterministically. In implementations, rules are selected due to their syntactical ordering in the source program for efficency reasons, i.e., the ordering can be decided by the programmer. The ordering is generally done first to last and the evaluation is generally done depth-first.

Furthermore, logic programs may have different execution schemes: One possibility is, that a rule that is called, tries to find all solutions at once, i.e, the program finally produces all possible solutions. Another possiblity is, that a rule produces only the first possible solution and only, if this solution leads to a failure later on, the rule is called again and produces another solution, until a solution leads to final success, or there is no other solution possible. This evaluation mechanism is called backtracking.

3.3 Prolog

Beyond the concept of logic programming, Prolog has some extentions, that will be explained in the following.

Again, first some introducing definitions have to be made.

Definition 3.5: ***Operation symbols***

Let $I := I_O \cup I_M$ be a set of infix operations with

i) $I_O := \{+, -, *, /, \%\}$ with $+, -, *, \backslash, \%$ for arithmetic operations

ii) $I_M := \{is, =_a, \neq_a, <_a, \leq_a, >_a, \geq_a, \equiv_u, \neq_u, <_u, \leq_u, >_u, \geq_u\}$, where

- *is* stands for assignment,
- $=_a, \neq_a, <_a, \leq_a, >_a, \geq_a$ are for arithmetical comparison.
- $\equiv_u, \neq_u, <_u, \leq_u, >_u, \geq_u$ are for comparison of term structures, i.e., unifiablity of terms.

Now a Prolog-program can be defined.

Definition 3.6: ***A Prolog-Program***

A Prolog-program $\mathcal{P}$ is a logic program $(C, L_{\mathcal{P}}, P, T_e \cup T_{\mathcal{P}} \cup Z_{\mathcal{P}})$ with the following extentions:

1. A simple term $t_e \in T_e$ is

i) a constant a, or

ii) a variable X.

2. An extended term $t \in T_{\mathcal{P}}$ is

i) a simple term $t_e \in T_e$,

ii) if t_1 and t_2 are terms and $\rho \in I_O$, then $t_1 \rho\, t_2 \in T_{\mathcal{P}}$, or

iii) if $t_1, ..., t_m$, $m \geq 1$ are terms and $\varrho \in L_{\mathcal{P}}$, then $\varrho(t_1, ..., t_m) \in T_{\mathcal{P}}$.

3. A list $l \in T_e \cup Z_{\mathcal{P}}$ is

i) a simple term $t_e \in T_e$, or

ii) if $l_1, ..., l_m$, $m \geq 1$ are lists of terms, then $[l_1|...|l_m]$ are lists of terms.

4. In the literal $\nu(t_1, ..., t_u)$, $u \geq 1:$ $t_1, ..., t_u$ are simple terms, i.e., $t_1, ..., t_u \in T_e$.

5. The set of literals $L_{\mathcal{P}}$ is extended to a set $L_{\mathcal{P}}$: $L_{\mathcal{P}} = L_{\mathcal{P}} \cup \mathcal{F} \cup \{!, \texttt{fail}, \texttt{succ}\} \cup \mathcal{M} \cup \mathcal{N}$.

a) $\mathcal{F}$ is a set of predefined functions. Literals of that kind are also called metapredicates.

b) $L_{\mathcal{P}}$ consists of three special elements:

i) The cut, "!", realizes some sort of control flow in the body of a rule.

ii) "`fail`" and "`succ`" are also needed for control flow.

c) $\mathcal{M}$ is a set of literals of the form

i) $a\tau t$, with a is a constant, t a term and $\tau \in I_M \backslash \{is\}$.

ii) $X\tau t$, with X is a variable, t a term and $\tau \in I_M$.

$\mathcal{M} := \{a\tau t \mid a \text{ is a constant}, t \in T_e \cup T_{\mathcal{P}}, \tau \in I_M \backslash \{is\}\} \cup \{X\tau t \mid X \text{ is a variable}, t \in T_e \cup T_{\mathcal{P}}, \tau \in I_M\}$.

d) $\mathcal{N}$ is a set of literals of the form

i) $a\tau l$, with a is a constant, l a list and $\tau \in I_M \backslash \{is\}$.

ii) $X\tau l$, with X is a variable, l a list and $\tau \in I_M$.

$\mathcal{N} := \{a\tau l \mid a \text{ is a constant}, l \in T_e \cup Z_{\mathcal{P}}, \tau \in I_M \backslash \{is\}\} \cup \{X\tau l \mid X \text{ is a variable}, l \in T_e \cup Z_{\mathcal{P}}, \tau \in I_M\}$.

Additional remarks [22]:

1. For arithmetic operations, assignments, and comparisons infix functors are used in general. The construction of lists can be done by $[X|Xs]$, where X is an atomic symbol, i.e., a variable or a constant, and Xs may be a symbol or again a list.

2. Literals may now also be operations with infix functors, or list constructions.

3. Literals may now also specify predefined functions.

4. Generally, in Prolog, there may be assignments (e.g., $p(X-1,Y)$) or even list constructions (e.g., $p([X|Xs],X,Xs)$) in the list of parameters of the head of a rule or of a literal in the body. In this model, on the one hand, a new uniquely named variable is created to replace the expression, and on the other hand a new literal evaluating the expression is inserted in the rule.
For example, if X and Y are output variables,
$p(X+1,Y*2) \leftarrow q(X,Y).$
is translated into:
$p(X^1,Y^1) \leftarrow q(X,Y), X^1 \ is \ X+1, Y^1 isY*2.$
If X would be of type input, the translation would be:
$p(X^1,Y^1) \leftarrow X \ is \ X^1-1, q(X,Y), Y^1 isY*2.,$
i.e., a reversive expression for $X^1 \ is \ X+1$ has to be created. While this is possible for simple arithmetic operations as $+$, it is not necessarily solvable for complex arithmetic operations, e.g., (X^2+Y^4).
The solution for this problem is, that operations in the list of parameters of the head of a rule are restricted to those of type output, while operations in the list of parameters or a literal in the body of a rule are restricted to those of type input.
The problem does not exist for lists: The reversive operations to the list construction made by $X^1 \ is \ [X|Xs]$ are $X \ is \ head(X^1), Xs \ is \ rest(X^1)$.
The explained replacements allow for the definition of Prolog as made above.

5. If a fact contains an expression like the one above the same replacement will be done. Thus, this fact actually becomes a rule.

6. The cut, "!", is a special syntax element appearing in the body of a rule to realize a simple form of control.
It reduces the search-space by preventing the evaluation from following fruitless computation paths, known to the programmer.
Backtracking over the cut is not allowed; in
$$p^0 \leftarrow p^1, ..., p^k, !, p^{k+1}, ..., p^n.$$
the rule fails if one of the literals $p^{k+1}, ..., p^n$ fails in a way that would make backtracking to $p^1, ..., p^k$ necessery.

7. Negation by failure is included by the general implementation of the metapredicate **not** is, while X has either to be a constant or a variable instantiated with a value:
   $not(g(X)) \leftarrow g(X), !, fail.$
   $not(g(X)).$

8. By convention, a single capital and words beginning with a capital denote variables, while a lower case letter or words beginning with a lower case letter denote constants and literals.

3.4 Parallel Forms of Prolog

Parallelism in Prolog can be found at the level of rule, e.g., if there is more than one rule with the same head p^0, each of the rules can be called independently and thus in parallel. Or at the level of the literals in the body of a rule, that can also be applied in parallel (with some restrictions).

Evaluating a Prolog-program in parallel without support of special constructs has proved to be very inefficient and bears in particular problems concerning data binding environments and non-terminating recursions. Thus, numerous implementations of parallel Prolog have been tried.

The various concepts for parallizing Prolog can be classified as follows:

1 *OR-parallism*
The search for rules to be applied for a goal is parallelized and, if more than one goal is found, the goals are executed in parallel.

2a *Full AND-parallelism*
All goals in the body of a rule are evaluated in parallel. However, since each evaluation may produce contradicting bindings for shared variables there can be an enormous overhead in calculation. This overhead is avoided by alternative forms of AND-parallelism.

2b *Restricted AND-parallelism*
Only goals with non-shared variables are evaluated in parallel.

2c *Stream AND-parallelism*
Stream AND-parallelism is an extension of restricted AND-parallelism. The bindings for any shared variables are passed inside the body of a rule from one goal to the next goal in streams. This way, as soon as a binding is calculated in one goal, the next goal can start its evaluation without waiting for all possible bindings.

4 Dataflow Architecture Principles

4.1 Data-Driven Computation

A dataflow graph consists of nodes connected by directed arcs. Each node represents a separate operation element, capable of accepting, processing and distributing data. The arcs define paths over which data are asynchronously passed from one node to another by tokens [11].

Enabling a node for execution does not require any centralized control. Rather, a node may be executed, when all needed data are present on the input arcs by input tokens. Executing a node means applying the node's function on these data. After execution, input tokens are deleted and new tokens carrying the calculation results are placed on the output arcs. Nodes with all needed input data present may be executed independently in parallel or in any order, while a node that depends on the data produced by another node is executed in sequence to this node.

This fundamental principle permits nodes to be evaluated in a parallel way, without explicitly specifiying the parallelism.

4.2 Static and Dynamic Dataflow Architectures

If a node receives tokens on different input arcs, it has to be clear, that data in the tokens belong to the same evaluation cycle. In a linear program part every node gets a token by each input arc and sends a token on each output arc once a graph evaluation. Within an iteration, though, a node gets more than one token and sends out more than one token during the graph evaluation.

To specify corresponding tokens, different concepts of dataflow architectures have been introduced: In static dataflow architectures only one token is allowed on each arc, i.e., the execution of a node has to be delayed until there are no more tokens on its outgoing arcs. Thus, tokens of different evalation cycles cannot be mixed up.

Dynamic dataflow architectures overcome this restriction by tagging the tokens, but the drawback is on the one hand token-matching and on the other hand the problem of tagging the tokens uniquely without relying on a dynamic tag size. Monsoon [20], a special dynamic dataflow architecture, overcame the difficulty of comparing tags with large tag sizes by implementing a *explicitly token store*, but this is still dynamically organized.

5 Knowledge-based Control of Intelligent Automous Systems

Compared with general-purpose Prolog-programs, a knowledge-based control system, formulated in Prolog, works in a completely different way:

- *Static queries*

  A general-purpose Prolog-program has to answer a variety of different questions, whereas a control system has to produce data for a fixed kind and number of control outputs that do not change during run-time.

  In the case of an autonomous mobile e.g., this outputs may be values for acceleration, brake and wheel.

- *Partially dynamic database*

  The database of a general-purpose Prolog-program is mostly static. For a control system, the database contains information, e.g., about the system, the mission goal, the environment and the actually sensored situation. This means, at least some of the data (e.g., visual sensor data) change from time frame to time frame (e.g., due to every new evaluated picture from the video camera).

- *Frequently repeated evaluation*

  Due to real-time conditions, calculation not only has to be fast, it also has to be repeated frequently in correspondence to the changes of the database. I.e., when a change in the environment is sensored, a new control output is necessary as soon as possible to react correctly.

Thus, in a sequence of time frames knowledge-based control systems always impose the same static set of queries, while part of the facts change dynamically.

The differences of knowledge-based control to a general-purpose program now allow for a new dataflow model.

6 The Dataflow Model for Knowledge-Based Systems

The above mentioned conditions for knowledge-based control systems affect the evaluation method strongly: Since the queries are static, they can be compiled into the graph. Thus, they no longer have to cause demands-for-evaluation flowing along the graph from the queries to the facts. Rather, evaluation can immediately start at the facts, causing results flowing bottom-up from the facts to the static set of queries. Side-effects caused by changing database items are eliminated by the condition, that updates, occuring after an evaluation has started, only affect the evaluation of the next time frame.

The compilation of a knowledge-based control system in Prolog into a dataflow graph comprises three steps:

As a first step, the structure of the Prolog-program, including the static queries, is mapped onto a structure graph. Since the queries that drive the control outputs are static, all data dependencies, i.e., input and output types of parameters of nodes, are fixed and can be analyzed at compile time in a second step. In the final step, the dataflow graph is generated by substituting data dependencies by dataflow edges.

6.1 Deriving the Structure Graph

For the definition of a structure graph, first some basic definitions concerning graphs have to be made.

Definition 6.1: ***Graph***

A graph $\mathcal{G}$ is a tuple (V, E), with

i) V is a set of nodes.

ii) $E \subset V \times V$ is a set of edges between the nodes.

Definition 6.2: ***Directed Graph***

A directed graph $\mathcal{G}_d$ is a tuple (V, E_d), with:

i) V is a set of nodes.

ii) $E_d \subset V \times V$ is a set of directed edges,

with $e = e_{u,v}$ for $e \in E_d$ connecting $u, v \in V$ with direction from u to v.

iii) A path in the graph is a tuple $(e^1, ..., e^n)$ with $e^1, ..., e^n \in E_d$, $n \geq 1$ and $\exists v_1, ..., v_n, v_{n+1} \in V: \; e^i = e^i_{v_i, v_{i+1}} \; \forall i \in \{1, ..., n\}$.

iv) A path without cycles is a tuple $(e^1, ..., e^n)$, $n \geq 1$, with $e^i = e^i_{v_i, v_{i+1}} \in E_d \; \forall i \in \{1, ..., n-1\} \; \wedge \; v_i \neq v_j \; \forall i \neq j \in \{1, ..., n-1\}$.

Definition 6.3: ***Tree***

A tree $\mathcal{T}$ is a directed graph (V, E_d).

i) The root w of the tree is a node $w \in V$:
$\exists v \in V \; \exists e_{w,v} \in E_d \; \wedge \; \forall v \in V \; \nexists e_{v,w} \in E_d$ and
$\forall w' \in V \; (\exists v \in V \; \exists e_{w',v} \in E_d \; \wedge \; \forall v \in V \; \nexists e_{v,w'} \in E_d \Rightarrow w = w')$.

Thus, the root is a node, that has only outgoing edges. In general a tree has only one root.

ii) A branch in a tree is a path without cycles.

iii) A leave v of a tree is a node $v \in V$:
$\exists u \in V \; \exists e_{u,v} \in E_d \; \wedge \; \forall u' \in V \; \nexists e_{v,u'} \in E_d$.

Thus, a leave is a node without outgoing edges.

Now the structure graph can be defined.

Definition 6.4: ***Structure Graph***

A structure graph SG to a Prolog program $\mathcal{P}^e = (C, L'_{\mathcal{P}}, P, T_{\mathcal{P}} \cup Z_{\mathcal{P}})$ is defined as $SG = (\mathcal{P}, V, E_d, F, attr, name)$, with

i) V is a set of nodes.

$V := V^+ \cup V_p \cup V_{op} \cup V_q$, with

- $V^+ := \{v_i^+ \mid i \in \{1, ..., m\} \wedge m_i \geq 2\}$
- $V_p := \{v^0_{i,k} \mid i \in \{1, ..., n\}, k \in \{1, ..., m_i\}\}$
- $V_q := \{v_q\}$
- $V_{op} := \{v^r_{i,k} \mid i \in \{1, ..., n\}, k \in \{1, ..., m_i\} \wedge \exists r \geq 1 : p^r_{i,k} \in \mathcal{F} \cup \{!, \texttt{fail}, \texttt{succ}\} \cup \mathcal{M} \cup \mathcal{N}\} \cup \{v^r_q \mid \exists r \geq 1 : q^r \in \mathcal{F} \cup \{!, \texttt{fail}, \texttt{succ}\} \cup \mathcal{M} \cup \mathcal{N}\}$

ii) $E_d \subset V \times V$ is a set of directed edges:

$E_d := E^+ \cup E_p \cup E_q$, with

- $E^+ := \{e_{v_i^+, v_{i,k}^0} \mid i \in \{1, ..., n\}\ k \in \{1, ..., m_i\} \wedge m_i \geq 2\}$

- $E_p :=$
$\{e_{v_{i,k}^0, v_{i'}^+} \mid i, i' \in \{1, ..., n\}, k \in \{1, ..., m_i\} \wedge m_{i'} \geq 2 \wedge \exists r \geq 1 : p_{i,k}^r \simeq p_{i',1}^0\} \cup$
$\{e_{v_{i,k}^0, v_{i',1}^0} \mid i, i' \in \{1, ..., n\}, k \in \{1, ..., m_i\} \wedge m_{i'} = 1 \wedge \exists r \geq 1 : p_{i,k}^r \simeq p_{i',1}^0\} \cup$
$\{e_{v_{i,k}^0, v_{i,k}^r} \mid i \in \{1, ..., n\}, k \in \{1, ..., m_i\} \wedge \exists r \geq 1 : p_{i,k}^r \in \mathcal{F} \cup \{!, \texttt{fail}, \texttt{succ}\} \cup \mathcal{M} \cup \mathcal{N}\}$

- $E_q :=$
$\{e_{v_q, v_i^+} \mid i \in \{1, ..., n\}, k \in \{1, ..., m_i\} \wedge m_i \geq 2 \wedge \exists r \geq 1 : q^r \simeq p_{i,1}^0\} \cup$
$\{e_{v_q, v_{i,1}^0} \mid i \in \{1, ..., n\} \wedge m_i = 1 \wedge \exists r \geq 1 : q^r \simeq p_{i,1}^0\} \cup$
$\{e_{v_q, v_q^r} \mid \exists r \geq 1 : q^r \in \mathcal{F} \cup \{!, \texttt{fail}, \texttt{succ}\} \cup \mathcal{M} \cup \mathcal{N}\}$

iii) F is a set of attributes:
$F = \{\text{AND, OR, FACT, CALC, FUNC, CUT, FAIL, SUCC, QUERY}\}$

iv) $attr : V \to F$ with $v.attr$ is the attribute for the node v.

- $$v_{i,k}^r.attr = \begin{cases} \text{AND} & \text{if } r = 0 \text{ and } p_{i,k}^r \text{ist Kopf einer Regel} \\ & \quad p_{i,k}^0 \leftarrow p_{i,k}^1, ..., p_{i,k}^{l_{i,k}}. \\ \text{FACT} & \text{if } r = 0 \text{ and } p_{i,k}^r \text{ ist Kopf einer Regel } p_{i,k}^0. \\ \text{SUCC} & \text{if } r \geq 1 \text{ and } p_{i,k}^r = \texttt{succ} \\ \text{FAIL} & \text{if } r \geq 1 \text{ and } p_{i,k}^r = \texttt{fail} \\ \text{CUT} & \text{if } r \geq 1 \text{ and } p_{i,k}^r = ! \\ \text{FUNC} & \text{if } r \geq 1 \text{ and } p_{i,k}^r \in O \\ \text{CALC} & \text{if } r \geq 1 \text{ and } p_{i,k}^r \in M^e \cup N^e \end{cases}$$

- $v_i^+.attr = \text{OR}$

- $v_q.attr = \text{QUERY}$

v) $name : V \to L$ with $v.name$ is the literal that is assigned to the node v.

- $v_{i,k}^r.name = p_{i,k}^r,\ r \geq 0$

- $v_i^+.name = p_{i,1}^0$

- $v_q.name = q^0$

By this definition an assignment is made between nodes and literals.

The structure graph is a graph where every literal of the rule is represented by a node and the nodes are connected by edges due to the structure of the original program: Every head of a rule becomes an AND-node. Heads of alternative rules, i.e., rules with the same head and the same set of parameters, are connected by an OR-node. E.g., when a rule $p^0 \leftarrow p^1, ..., p^k$. occurs, edges going from p^0 to p^1, ..., from p^0 to p^n are created. p^1 to p^n are either an OR-node, if an OR-node exists for one the literal, or, if no OR-node exists, the AND-node itself.

The result is a tree-like graph that may contain cycles. Each path of the graph is a sequence of AND-OR-AND- as well as AND-AND-structures in any arbitrary combination. The leaves of the graph are nodes representing the other elements of a Prolog-program, i.e., facts, assignments, comparisons, and metapredicates.

All other elements of a Prolog-program, i.e., facts, assignments, comparisons, and metapredicates, become leaves of the graph. Concerning their complexity, it stands that facts and metapredicates are represented by atomic nodes, but assignments (e.g., X *is* $Y * Z$) and comparisons (e.g., $X \leq 0$), are represented by macronodes. Behind these again a dataflow graph structure stands for evaluating the expression.

An example is given in Fig. 2. Facts are represented by FACT-nodes, macronodes by CALC-nodes.

Example:

```
func(X,Y,Z) :- ddb(X), calc(X,Y,Z);
calc(X,Y,Z) :- quad(X,Y), inv(X,Z);
mul(X,Y) :- Y is X * X;
inv (X,Z) :- Z is 1 / X;
ddb(2);
ddb(4);
```

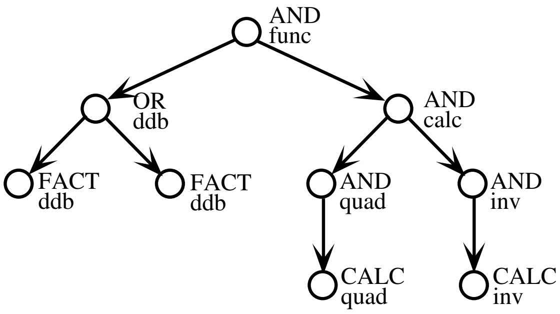


Figure 2: Prolog-example and corresponding structure graph

6.2 Analyzing the Data Dependencies

In the second step, as a new approach, data-dependencies are analyzed and edges are changed according to these dependencies.

Analyzing the data-dependencies means analyzing which parameters of a node are of type input and which ones are of type output. If a parameter is of type input, its value must be known before evaluation takes place in the node or in the subtree of which the node is the root. Otherwise, if a parameter is of type output, the value of the parameter is produced in the node or in its subtree.

Remark: In all following definitions the QUERY-node is handled the same way as the AND-node and not mentioned explicitly.

The syntax of $v.name$ is not uniform, due to the fact, that the node can be an assigment as well as a metapredicate. Thus, a general form of a name has to be defined to simplify following definitions, where the parameters of a node are needed.

Definition 6.5: ***Simple terms in a term or a list***

Let M_t be the set of all simple terms of t and $\#t \in \mathbb{N}$ the amount of all simple terms in t.

- If $t \in T_e$, then $M_t = \{t\}$ and $\#t = 1$
- If $t \in T_\mathcal{P}$ and $t = t_1 \rho t_2 \in T_\mathcal{P}$, $t_1, t_2 \in T_\mathcal{P}$, $\rho \in I_O$, then $M_t = M_{t_1} \cup M_{t_2}$ and $\#t = \#t_1 + \#t_2$.
- If $t \in T_\mathcal{P}$ and $t = \varrho(t_1, ..., t_m), m \geq 1$, $t_1, ..., t_m \in T_\mathcal{P}, \varrho \in L_\mathcal{P}$, then $M_t = M_{t_1} \cup ... \cup M_{t_m}$ and $\#t = \#t_1 + ... + \#t_m$.
- If $l \in Z_\mathcal{P}$ and $l = [l_1|...|l_m], m \geq 1$, $l_1, ..., l_m \in Z_\mathcal{P}$ then $M_l = M_{l_1} \cup ... \cup M_{l_m}$ and $\#l = \#l_1 + ... + \#l_m$.

Definition 6.6: ***Most general form of a parameterlist of a node***

- $v.name = \nu()$ is the most general form of a parameterlist of a node without parameters.
- $v.name = \nu()$ or $v.name = \nu(t_1, ..., t_u)$ with $\nu \in P$, $t_1, ..., t_u \in T_e$ is the most general form of a parameterlist of a node with parameters.
- If $v.name = X \tau t, X$ is a simple term, $t \in T_\mathcal{P}, \tau \in I_M$, then the most general form of a parameterlist is $\tau_p(X, t_1, ..., t_u)$, with τ_p as prefix notation for τ and $M_t = \{t_1, ..., t_u\} \in T_e$, $u = \#t$.
- If $v.name = X \tau l, X$ is a simple term, $l \in Z_\mathcal{P}, \tau \in I_M$, then the most general form of a parameterlist is $\tau_p(X, t_1, ..., t_u)$, with τ_p as prefix notation for τ and $M_t = \{t_1, ..., t_u\} \in T_e$, $u = \#l$.

Now it is possible to define a parameter of a node.

Definition 6.7: ***Parameter of Nodes***

The set of parameters of a node is a tuple $KP = (T_e, I, IOT$, *name, uname, history, type*) with $I \subset \mathbb{N}$, $IOT = \{\texttt{undef}, \texttt{in}, \texttt{out}, \texttt{const}\}$.

A parameter $\delta_i \in KP$, $i \in \{1, ..., m\}$, $m \geq 1$ of a node v with $v.name = \nu(t_1, ..., t_m)$, in most general form is defined as follows:

- $\delta_i.name$ is the parameter, i.e., the simple term the parameter represents.
  $name : KP \rightarrow T_e$, with
  - $\delta_i.name = a$, if $t_i = a$.
  - $\delta_i.name = X$, if $t_i = X$.
- $\delta.uname$ is a list of unique names for the parmeter t.
  $uname : KP \rightarrow I$.
- $\delta.history$ is the history of the parameter.
  $history : KP \rightarrow IOT$.
- $\delta.type$ is the input/output type of the parameter.
  $type : KP \rightarrow IOT$.

An algorithm will be given later on how to specify the attributes $uname, history$ and $type$.

For the reason of simplifying following definitions, some more terms are introduced.

Definition 6.8: ***Subnode, Subgraph***

$\mathcal{G}_d = (V, E_d)$ is a directed graph.

- A node $w \in V$ is a *subnode* of a node $v \in V$, if $e_{v,w} \in E_d$.
- A *subgraph* $\mathcal{G}(v)_d = (V^v, E_d^v)$ with root $v \in V$ is a directed graph with:
  $v \in V^v$ and
  $\forall w \in V$ for which a path $(v, ..., w) \in \mathcal{G}_d$ exists, is $w \in V^v$.
  $\forall w, w' \in V^v$ with $e_{w,w'} \in E_d$ is $e_{w,w'} \in E_d^v$
- A node $w \in V$ is *beneath* a node $v \in V$, if $w \in \mathcal{G}(v)_d$, i.e., if w is in the subgraph of v.

Furthermore a linear ordering on nodes has to be made.

Definition 6.9: ***Depth-First Left-Right Ordering of Nodes***

By the definition of the set $E_d := E^+ \cup E_p \cup E_q$ a ordering of nodes $v, v' \in V$ is given:

i) A node $v \in V$ has greater depth than a node $v' \in V$, if
   - $e_{v',v} \in E_d$, or
   - $\exists v'' \in V : e_{v'',v} \in E_d$ and v'' has greater depth than v'.

ii) A node $v \in V$ is locally left to a node $v' \in V$, if

- $\exists w \in V_q$ and $\exists e_{w,v}, e_{w,v'} \in E_d \ \wedge \ v.name \simeq q^r, v'.name \simeq q^{r'}$, $r, r' \in \{1, ..., l\},\ r > r'$.
- $\exists w \in V^+$ and $\exists e_{w,v}, e_{w,v'} \in E_d \ \wedge \ v.name = p^0_{i,k}, v'.name = p^0_{i,k'}$, $i \in \{1, ..., n\},\ k, k' \in \{1, ..., m_i\},\ k < k'$.
- $\exists w \in V_p$ and $\exists e_{w,v}, e_{w,v'} \in E_d \ \wedge \ v.attr \simeq p^r_{i,k}, v'.attr \simeq p^{r'}_{i,k}$, $i \in \{1, ..., n\},\ k \in \{1, ..., m_i\},\ r, r' \in \{1, ..., l_{i,k}\},\ r < r'$.

This "left-right" ordering of nodes $v, v' \in V$ can only be made locally to a node $w \in V$ with $e_{w,v}, e_{w,v'} \in E_d$, because in SG there may be a node $w' \in V$, for which v' is left of v according to the above definition.

Therefore, the definition has to be extended to a global "left-right" ordering.

iii) A node $v \in V$ is locally left to a node $v' \in V$, if $\exists x, x' \in V$, $v \in V^x, v' \in V^{x'}$ and x is locally left to x'.

iv) A node $v \in V$ is globally left to a node $v' \in V$, if

- v is locally left to v',
- $\exists x, x' \in V$, $v \in V^x, v' \in V^{x'}$ and x is locally left to x' and $\nexists y, y' \in V$, $V^y, V^{y'}$ with $v \in V^y, v' \in V^{y'}$ and y' is locally left to y, or
- $\exists x, x' \in V$, $v \in V^x, v' \in V^{x'}$ and x is locally left to x' and $\exists y, y' \in V$, $V^y, V^{y'}$ with $v \in V^y, v' \in V^{y'}$ and y' is locally left to y, but $\nexists z, z' \in V$, $V^z, V^{z'}$ with $x \in V^z, y \in V^{z'}$ and z' is globally left to z.

$v \in V^x$ does not necessarily require $v \neq x$.

Thus rules for each type of node to analyze the input and output types of its parameters are in general, when handling

- *Facts*:
  All parameters of a fact are of type output, since facts realize an access to the database to input data into the program.
- *Metapredicates:*
  The type of every parameter is given by the declaration part of the Prolog built-in-funtion (e.g., *write(X)* has the input parameter X, while *read(X)* produces an output for X).
- *Calculations:*
  For comparison each Variable has to be of type input, otherwise the comparison could not be evaluated. For assignment it holds that if the variable appears left of

the assignment symbol it is of type output, otherwise it is of type input (e.g., the assignement X *is* $(Y^2+1)/Z$ needs the two parameters Y and Z for input, while the result is assigned to X).

- *AND-nodes:*
  The type of a parameter is determined by that child-node among all child-nodes that uses the parameter first. If this first appearence enforces the type output, the parameter may be of type input or output at the other child-nodes, but is produced by the AND-node. On the other hand, if the type input is enforced, the parameter has to be of type input at the AND-node, because there is no other possiblity for calculating any input value.
- *OR-nodes:*
  The type of a parameter results from the type of the parameter of the AND-child-nodes, since every OR-node has the same set of parameters as the AND-nodes which the OR-node connects.

Furthermore, each node produces a result *success* or *failure*, that is, handled as a normal parameter, generally of type output.

However, there are special cases, where a parameter is used in a way at a node, that the above rules are not sufficient for a correct analysis. For these cases not only the usage of the parameter at the node or the type of the parameter at the node's child-nodes has to be considererd, but the type of the parameter in other parts of the graph.

By applying a rule, parameters get renamed most of the time, though it still is the same parameter, calculating values for or consuming values from other occurances of the same parameter in other parts of the graph, e.g., if the literal $g(X,Y)$ calls a rule with the head $h(X,Z)$ the parameter Y is renamed to Z. Thus, to examine the type of a parameter in other parts of the graph, each parameter must have a unique name for identification.

Unique names can easily be attached to each occurance of a parameter by starting at the root of the graph, and considering all renamings that take place along each path in the graph from the root to a leaf. This is done by the following algorithm.

Definition 6.10: *Unique Names for Parameters*

uname: $KP \rightarrow (\mathbb{N} \times ... \times \mathbb{N})$

For all nodes $v \in SG$ with $v.name = \nu(t_1, ..., t_u)$, $u \geq 1$ and parameters δ_i with $i \in \{1, ..., u\}$: $\delta_i.uname := [\,]$, i.e., the empty list.

$n := 1$

1. Start with the QUERY-node $v_q \in SG$: $v := v_q$
2. For node v do

   a) if $v.name = \nu()$ do nothing.

   b) for node v with $v.name = \nu(t_1, ..., t_u)$, $u \geq 1$ do for all parameters δ_i with $i \in \{1, ..., u\}$:

      i) If $\delta_i.uname = []$, then give a new unique name $\delta_i.uname := [n]$. $n := n + 1$.

      ii) For all nodes $w \in SG$ with $e_{v,w} \in E_d$ with $w.name = \nu(l_1, ..., l_m)$, $m \geq 1$ and parameters $\theta_1, ..., \theta_m$ do:
      if $\exists j \in \{1, ..., m\}$ with $\theta_j.name = l_j = \delta_i.name$,
      then $\theta_j.uname := [\delta_i.uname|\theta_j.uname]$, i.e., $\delta_i.uname$ is inserted in the list of unique names, θ_j already has.

   c) For all nodes $w \in SG$ with $e_{v,w} \in E_d$ call 2 with $v := w$.

There may be different pathes from the root to a node in the graph and therefore the parameters at this node and corresponding parameters further on the path to the leaves may get more then one unique name.

Example:
$f(X, Y) \leftarrow g(X, Z), g(Z, Y).$
$g(X, Y) \leftarrow h(X, Y).$
$h(X, Y).$

Assume that, when the rule $f(X, Y)$ ist checked, X has the unique name 4 and Y the unique name 5. Z is a new parameter in the rule $f(X, Y)$ and thus gets a new, unique name, e.g., 8. When the rule $g(X, Y)$ is called by $g(X, Z)$, the parameter X in the rule $g(X, Y)$ gets the unique name 4, the parameter Y gets the same unique name as Z, 8. When the rule $g(X, Y)$ is called a second time, X gets the unique name 8, Y the unique name 4. Thus, each parameter may have a list of unique names. When the rule $h(X, Y)$ is called by $h(X, Y)$, then X gets the list of unique names [4|8], Y gets [8|5].

However, each unique name of the list depicts to which parameters in the graph it corresponds.

Now the analysis can be extended for the special cases in a way that other occurances of the same parameter in the graph can be considered as well as the local occurances.

i) In the list of parameters of a FACT- or an AND-node, the same parameter may occur twice or even more often.

For a FACT-node it holds, that if a parameter occurs only once in the parameterlist, it is an output of the database. But if a parameter occurs more than once, it could possibly be of type input once.

Example: (Calculating $Y = X * 2^I$)
$f(0, X, X).$
$f(I, X, Y) \leftarrow J \text{ } is \text{ } I - 1, Z \text{ } is \text{ } X * 2, f(J, Z, Y).$

It is not possible to analyze the type of the parameters of the fact locally at the fact only.

Instead, the analysis has to look whether one of the parameters with the same name was used somewhere before in the graph, i.e., whether there was a value calculated for the parameter. If this is true, the type of the parameter is of type input.

For an AND-node, if a parameter does not occur at a child-node, the parameter cannot be of type output. Especially, if there is more than one parameter with the same name, there has to be at least one parameter of type input. Again, the analysis has to look whether one of the parameters with the same name was used somewhere before in the graph.

ii) If an AND-node has a recursive call as a child node, and a parameter of the AND-node is only or first used in the recursive call, the type of the parameter cannot be decided:

Example: (ancestor-relationship)
$\texttt{ancestor}(X, Y) \leftarrow \texttt{father}(X, Y).$
$\texttt{ancestor}(X, Y) \leftarrow \texttt{ancestor}(X, Z), \texttt{father}(Z, Y).$

Since each correct recursion in Prolog consists of at least a rule for terminating the recursion and of the rule with the recursive call, these are connected in the structure graph via an OR-node. The OR-node takes the type for its parameter only from the termination rule. Now, the type of the parameter in the rule with the recursive call is given, too.

iii) There may be cases, when the type of a parameter of the AND-child-nodes of an OR-node are different, e.g., one type is output and one type is input. In this case, the parameter of the OR-node and the corresponding parameters of all AND-childnodes become type input, due to the fact, that an input is needed and no output produced by at least one AND-node, which counts more than the fact that an output may be produced by another node.

iv) Prolog allows in general for operations and list-constructions in the list of parameters of the head of a rule. Since this would need a functionality of an AND-node that includes calculations, the structure graph is constructed in a way that all operations and list-constructions get extracted from the list of parameters and a new child-node is created for the AND-node. But it is not possible to say, whether the expression was used for input or for output.

Example: (Appending a list at another list)

$f([], Ys, Ys).$

$f([X|Xs], Ys, [X|Zs]) \leftarrow f(Xs, Ys, Zs).$

is for the construction of the structure graph transformed to:

$f([], Ys, Ys).$

$f(X^1, Ys, X^2) \leftarrow f(Xs, Ys, Zs), X^1 \textit{ is } [X|Xs], X^2 \textit{ is } [X|Zs].$

It is not possible by just analyzing the rule or the fact to derive the input-/output types of the parameters. Therefore, again, the analysis has to look whether one of the parameters with the same name was used somewhere before in the graph, i.e., whether there was a value calculated for the parameter. If this is true, the type of the parameter is of type input.

In the example above, another change has to be made after the types of the parameters are known. Assuming that $X1$ is of type input, while $X2$ is of type output, the rules becomes:

$f([], Ys, Ys).$

$f(X^1, Ys, X^2) \leftarrow X \textit{ is } head(X^1), Xs \textit{ is } rest(X^1), f(Xs, Ys, Zs), X^2 \textit{ is } [X|Zs].$

The above algorithm is formally:

Definition 6.11: *History of a Parameter*

history: $KP \rightarrow IOT$

If $w \in SG$ with $w.name = \nu(l_1, ..., l_m)$, $m \geq 1$ and parameters $\theta_1, ..., \theta_m$ and $\exists v \in SG$ with $v.name = \nu(t_1, ..., t_u)$, $u \geq 1$, parameters $\delta_1, ..., \delta_u$ and $\exists \epsilon \in \delta_i.uname$ and $\epsilon \in \theta_j.uname$ and $\delta_i.type = \texttt{out}$,
then $\theta_j.history := \texttt{in}$,
otherwise $\theta_j.history := \texttt{out}$.

Definition 6.12: *Input/Output Types of Parameters*

IO is a set of attributes for parameters $l_i \in T_{\mathcal{P}} \cup Z_{\mathcal{P}}, i \geq 1$ of literals $p \in L_{\mathcal{P}}$ of a node $v \in V : IO := \{\texttt{undef, in, out, const}\}$.

$type : t_e \to IO$, where $l_i.type$ refers to the type of the parameter. $l_i, i \geq 1$ in p.

i) If $v.attr$ = CUT or $v.attr$ = FAIL or $v.attr$ = SUCC,
then no parameters exist.

ii) If $v.attr$ = FACT

if $v.name = \nu()$, then no parameters exist.

if $v.name = \nu(t_1, ..., t_u)$, $u \geq 1$ with parameters δ_1 ,..., δ_u, then for δ_i, $i \in \{1, ..., u\}$, $u \geq 1$:

- if $\delta_i.name = a$, a constant, then $\delta_i.type :=$ `const`,
- if $\delta_i.name = X$, X variable
  - if $\nexists \delta_j.name = \delta_i.name$, $j \in \{1, ..., u\}$, $j \neq i$, then $\delta_i.type :=$ `out`,
  - if $\exists \delta_j.name = \delta_i.name$ and $i < j \leq u$, then $\delta_i.type := \delta_i.history$,
  - if $\exists \delta_j.name = \delta_i.name$ and $1 \leq j < i$, then
    - if $\delta_j.type =$ `out` and $\delta_i.history =$ `in`, then $\delta_i.type :=$ `in`,
    - else $\delta_i.type :=$ `out`.

iii) If $v.attr$ = FUNC

if $v.name = \nu()$ then no parameters exist.

if $v.name = \nu(t_1, ..., t_u)$, $u \geq 1$ with parameters δ_1 ,..., δ_u, then the type of δ_i, $i \in \{1, ..., u\}$ is given by the definition of the metapredicate $\nu(l_1, ..., l_u)$ with parameters $\theta_1, ..., \theta_u$:

- if $\delta_i.name = a$, a constant and $\theta_i.type =$ `in`, then $\delta_i.type :=$ `const`,
- if $\delta_i.name = a$, a constant and $\theta_i.type =$ `out`, then $\delta_i.type :=$ `undef`,
- otherwise $\delta_i.type := \theta_i.type$.

iv) If $v.attr$ = CALC

if $v.name = \tau_p(t_1, ..., t_u)$, $u \geq 2$ with parameters $\delta_1, ..., \delta_u$:

- if $\delta_1.name = a$, a constant
  - if $\tau_p = is\ _p$ (prefix notation of *is*), $\delta_1.type :=$ `undef`, i.e., then this is an error,
  - if $\tau_p \in I_M \backslash \{\ is\ \}$ then $\delta_1.type :=$ `const`,
- if $\delta_1.name = X$, X variable
  - if $\tau_p = is\ _p$ then $\delta_1.type :=$ `out`,

· if $\tau_p \in I_M \backslash \{ \ is \ \}$ then $\delta_1.type := \texttt{in}$,

- if $\delta_i.name = a$, $2 \geq i \geq u$, a constante, then $\delta_i.type := \texttt{const}$,

- if $\delta_i.name = X$, $2 \geq i \geq u$, X variable, then $\delta_i.type := \texttt{in}$.

v) If $v.attr$ = AND or $v.attr$ = QUERY

if $v.name = \nu()$ then no parameters exist.

if $v.name = \nu(t_1, ..., t_u)$, $u \geq 1$ with parameters δ_1 ,..., δ_u, then the type of δ_i, $i \in \{1, ..., u\}$ is given as follows:

- if $\exists w$ with $e_{v,w} \in E_d$, $w.name = \nu'(t_1, ..., t_m)$, $m \geq 1$ and parameters $\theta_1, ..., \theta_m$ and $\theta_j.name = \delta_i.name$ for $j \in \{1, ..., m\}$ and
  if $\not\exists w'$ with $e_{v,w'} \in E_d$ with w' is left of w and $w.name = \nu''(t'_1, ..., t'_n)$, $n \geq 1$ and parameters $\theta'_1, ..., \theta'_n$ and $\theta'_k.name = \delta_i.name$ for $k \in \{1, ..., n\}$,
  then $\delta_i.type := \theta_j.type$,

- if $\not\exists w$ with $e_{v,w} \in E_d$ with $w.name = \nu'(t_1, ..., t_m)$, $m \geq 1$ and parameters $\theta_1, ..., \theta_m$ and $\theta_j.name = \delta_i.name$ for $j \in \{1, ..., m\}$, then:

  · if $\exists \delta_j.name = \delta_i.name$, $i < j \leq u$, then $\delta_i.type := \delta_i.history$,

  · if $\exists \delta_j.name = \delta_i.name$, $1 \leq j < i$, $\delta_j.type = \texttt{out}$ and $\delta_i.history = \texttt{out}$, then $\delta_i.type := \texttt{undef}$, i.e., this is an error,

  · if $\exists \delta_j.name = \delta_i.name$, $1 \leq j < i$, $\delta_j.type = \texttt{in}$ or $\delta_i.history = \texttt{in}$, then $\delta_i.type := \delta_i.history$.

vi) If $v.attr$ = OR

if $v.name = \nu()$ then no parameters exist.

if $v.name = \nu(t_1, ..., t_u)$, $u \geq 1$ with parameters δ_1 ,..., δ_u, then the type of δ_i, $i \in \{1, ..., u\}$ is given through w with $e_{v,w} \in E_d$, with $w.name = \nu(t_1, ..., t_u)$ and parameters $\theta_1, ..., \theta_u$:

- if $\theta_i.type = \texttt{in}$, then $\delta_i.type := \texttt{in}$,

- if $\theta_i.type = \texttt{out}$ or $\theta_i.type = \texttt{const}$ and $\not\exists w'$ with $e_{v,w'} \in E_d$, with $w'.name = \nu(t_1, ..., t_u)$ and parameters $\theta'_1, ..., \theta'_u$ with $\theta'_i.type = \texttt{in}$, then $\delta_i.type := \theta_i.type$,

- if $\theta_i.type = \texttt{undef}$ and $\not\exists w'$ with $e_{v,w'} \in E_d$, with $w'.name = \nu(t_1, ..., t_u)$ and parameters $\theta'_1, ..., \theta'_u$ with $\theta'_i.type = \texttt{in}$, $\theta'_i.type = \texttt{out}$ or $\theta'_i.type = \texttt{const}$, then $\delta_i.type := \texttt{undef}$, i.e., this is an error.

The result of this analysis is a structure graph with data dependencies.

Definition 6.13: ***A Structure Graph with Data Dependencies***

A structure graph with data dependencies SG^δ for a structure graph $SG = (\mathcal{P}, V, E_d,$ *F, attr, name*) is defined by $SG^\delta = (\mathcal{P}, V^\delta, E_d)$, with $V^\delta = (V, KP, F,$ *attr, name, dfsnr, para, anz, lpara, larnz, res*) and $KP = (T_e, I, IOT,$ *name, uname, history, type*), $I \subset \mathbb{N}$, for which

i) V^δ is a set of nodes.

ii) KP is a set of parameters of nodes.

iii) $dfsnr : V \rightarrow \mathbb{N}$ is a unique node name. The numbering is ordered according to the depth-first, left-right ordering.

iv) $anz : V \rightarrow \mathbb{N}$ is the number of parameters of the node v.
$v.anz = u$, $u \geq 1$, if $v.name = \nu(t_1, ..., t_u)$ in most general form.

v) $para : V \rightarrow \underbrace{KP \times ... \times KP}_{v.anz}$
is a function that assigns parameters to a node:

if v with $v.name = \nu(t_1, ..., t_u)$, $u \geq 1$ with parameters $\delta_1, ..., \delta_u \in KP$, where $\delta_i.name = t_i$ for $i \in \{1, ..., u\}$, then
$v.para_i = \delta_i$, or $para(v) = \{\delta_1, ..., \delta_u\}$ repectively.

vi) $lpara : V \rightarrow \underbrace{KP \times ... \times KP}_{n}$
is a function that assigns parameters to a node:

if $v.name = \nu(t_1, ..., t_u)$, $u \geq 1$ with parameters $\delta_1, ..., \delta_u \in KP$ then
$LP = \{\theta_{j_r} \mid \exists e_{v,w_j} \in E_d, j \geq 0$ with $w_j.name = \nu(t_{j_1}, ..., t_{j_u})$ and parameters $\theta_1, ..., \theta_{j_u}$, $r \in \{1, ..., j_u\}$ and $\theta_{j_r}.name \neq \delta_i.name\ \forall i \in \{1, ..., u\}\}$.

Now, $v.lpara_i := \theta_i$ with $LP = \{\theta_1, ..., \theta_n\}$, $n = |LP|$.

Thus, the set LP of local parameters consists of all pairwise different parameters, that occur in one of the subnodes, but not in the node itself.

vii) $lanz : V \rightarrow \mathbb{N}$ is the number of local parameters of the node v: $v.lanz = u'$ if $u' = |LP|$.

viii) $res : V \rightarrow KP$ is function, that assigns a result parameter to a node.

Local parameters occur solely at AND-nodes, because only AND-nodes and OR-nodes have subnodes and only the subnodes of an AND-node may have different parameters than the node itself. They have to be of type output, because there is no possibility to input a value for the parameter.

For $v \in V : \forall i' \in \{1, ..., v.lanz\}$:

- $v.lpara_{i'}.name = \theta_{i'}.name$

- $v.lpara_{i'}.uname = \theta_{i'}.uname$

- $v.lpara_{i'}.type = \texttt{out}$,

- $v.lpara_{i'}.history = \texttt{out}$.

The result parameter has also to be of type output, because each node produces a result value *success* or *failure*.

- $v.res.name =$ "RES"

- $v.res.uname = -1$

- $v.res.type = \texttt{out}$

- $v.res.history = \texttt{out}$

For all following definitions, all parameters, local parameters and result parameters are refered all together by the term parameter.

6.3 Constructing the Dataflow Graph

The construction of the dataflow graph has to examine for every input variable of every node from which node the variable is passed. This can easily be taken from the arcs of the structure graph and the analysis of the data-dependencies.

Definition 6.14: *Data-Dependencies of Nodes*

i) A node v with parameters $\delta_1, ..., \delta_u$ is data-dependend to a node w with parameters $\theta_1, ..., \theta_m$,
if $\exists \epsilon \in \theta_j.uname$ and $\epsilon \in \delta_i.uname$ for $j \in \{1, ..., m\}$, $i \in \{1, ..., u\}$ and $\theta_j.type = \texttt{out}$ and $\delta_i.type = \texttt{in}$.

ii) A node v is data-dependend to a node w', if v is data-dependend to a node w and w data-dependend to w'.

Definition 6.15: *Dataflow Graph*

A dataflow graph $DG = (\mathcal{P}^e, SG^\delta, V^\delta, E_{KP})$ is a graph given by
a prolog program $\mathcal{P}^e = (C, L'_\mathcal{P}, P, T_\mathcal{P} \cup Z_\mathcal{P})$,
a structure graph with data-dependencies $SG^\delta = (\mathcal{P}, V^\delta, E_d)$,
with $V^\delta = (V^z, KP, F^z,$ *attr, name, dfsnr, para, anz, lpara, larnz, res*),
$KP = (T_e, I, IOT,$ *name, uname, history, type*), $I \subset \mathbb{N}$ and $E_{KP} = E^v_d \cup E^m_d \cup E^r_d$:

i) V^z is a set of nodes $V \cup V^r$.
$V^r = \{v_u | \exists e_{w,v} \in E^r_d \wedge \exists v.para_i.type = \texttt{out},\ i \in \{0, ..., v.anz\}\} \cup \{v_d | \exists e_{w,v} \in E^r_d \wedge \exists v.para_i.type = \texttt{in},\ i \in \{0, ..., v.anz\}\}$

ii) F^z is a set of attributes for the nodes.
$F^z = F \cup \{REC_d, REC_u\}$

iii) The function $attr : V \rightarrow F$ is extended:
$v_u.attr = REC_u\ \forall v_u \in V^r$,
$v_d.attr = REC_d\ \forall v_d \in V^r$.

iv) The function $name : V \rightarrow L$ is extended:
$v_u.name = v.name\ \forall v_u \in V^r$,
$v_d.name = v.name\ \forall v_d \in V^r$.

v) Furthermore:
$v_u.para_i = v.para_i\ \forall v.para_i$ with $v.para_i.type = \texttt{out}$, $i \in \{0, ..., v.anz\}$,
$v_d.para_i = v.para_i\ \forall v.para_i$ with $v.para_i.type = \texttt{in}$, $i \in \{0, ..., v.anz\}$.

vi) $E_{KP} \subset KP \times KP$ is a set of directed edges: $E_{KP} = E^v_{KP} \cup E^{v'}_{KP} \cup E^r_{KP}$, with E^v_{KP}, $E^{v'}_{KP}$ and E^r_{KP} defined as:

- E^v_{KP} is defined as follows:

  · $e_{\theta,\delta} \in E^v_{KP}$
  if $\exists e_{v,w} \in E^v_d \cup E^m_d$ and $v.param_i = \delta, w.param_j = \theta$, $i \leq v.anz, j \leq w.anz$
  with $\theta.name = \delta.name$ and $\theta.type =$`out`.

  · An error occurs, if
  $\exists e_{v,w} \in E^v_d \cup E^m_d$ and $v.param_i = \delta, w.param_j = \theta$, $i \leq v.anz, j \leq w.anz$
  with $\theta.name = \delta.name$, $\theta.type =$`in` and $\delta.type =$`out` and
  $\nexists e_{v,w'} \in E_d$ and $w'.param_{j'} = \theta'$, $j' \leq w'.anz$ with
  $\theta'.name = \delta.name$ and $\theta'.type =$`out` and w' is left of w.

  · $e_{\delta',\theta} \in E^v_{KP}$
  if $\exists e_{v,w} \in E^v_d \cup E^m_d$ and $v.param_i = \delta, w.param_j = \theta$, $i \leq v.anz, j \leq w.anz$
  with $\theta.name = \delta.name$ and

$\theta.type$ =in and $\delta.type$ =in and
$\nexists e_{v,w'} \in E_d$ and $w'.param_{j'} = \theta'$, $j' \leq w'.anz$ with
$\theta'.name = \delta.name$ and $\theta'.type$ =out and w' is left of w and
$\exists e_{\delta',\delta} \in E_{KP}$.
If $e_{\delta',\delta} \in E_{KP}$ not exists, then an error occurs.

· $e_{\theta',\theta} \in E^v_{KP}$
if $\exists e_{v,w} \in E^v_d \cup E^m_d$ and $v.param_i = \delta, w.param_j = \theta$, $i \leq v.anz, j \leq w.anz$
with $\theta.name = \delta.name$ and $\theta.type$ =in and
$\exists e_{v,w'} \in E_d$ and $w'.param_{j'} = \theta'$, $j' \leq w'.anz$ with
$\theta'.name = \delta.name$ and $\theta'.type$ =out and w' ist links von w and
$\nexists e_{v,w''} \in E_d$ and $w''.param_{j''} = \theta''$, $j'' \leq w''.anz$ with
$\theta''.name = \delta.name$ and w'' is left of w'.

- $E^{v'}_{KP}$ is defined as:

· $e_{\theta',\delta} \in E^{v'}_{KP}$
if $\exists e_{v,w} \in E^r_d$ and $v.param_i = \delta, w.param_j = \theta$, $i \leq v.anz, j \leq w.anz$
with $\theta.name = \delta.name$ and $\theta.type$ =out, $\delta.type$ =out
and $w_u.param_k = \theta'$, $k \leq v_u.anz$ with $\theta.name = \theta'.name$.

· An error occurs if
$\exists e_{v,w} \in E^v_d$ and $v.param_i = \delta, w.param_j = \theta$, $i \leq v.anz, j \leq w.anz$
with $\theta.name = \delta.name$ and $\theta.type$ =in, $\delta.type$ =out and
$\nexists e_{v,w'} \in E_d$ and $w'.param_{j'} = \theta'$, $j' \leq w'.anz$ with
$\theta'.name = \delta.name$.

· $e_{\delta',\theta'} \in E^{v'}_{KP}$
if $\exists e_{v,w} \in E^r_d$ and $v.param_i = \delta, w.param_j = \theta$, $i \leq v.anz, j \leq w.anz$
with $\theta.name = \delta.name$ and $\theta.type$ =in, $\delta.type$ =in
and $\exists e_{\delta',\delta} \in E_{KP}$
and $v_d.param_k = \theta'$, $k \leq v_d.anz$ with $\delta.name = \theta'.name$.
If $e_{\delta',\delta} \in E_{KP}$ not exists, then an error occurs.

· $e_{\theta'',\theta'} \in E^{v'}_{KP}$
if $\exists e_{v,w} \in E^r_d$ and $v.param_i = \delta, w.param_j = \theta$, $i \leq v.anz, j \leq w.anz$
with $\theta.name = \delta.name$ and $\theta.type$ =in and
$\exists e_{v,w'} \in E_d$ and $w'.param_{j'} = \theta''$, $j' \leq w'.anz$ with
$\theta''.name = \delta.name$ and $\theta''.type$ =out and w' ist links von w
and $v_d.param_k = \theta'$, $k \leq v_d.anz$ with $\delta.name = \theta'.name$.

· $e_{\theta',\theta''} \in E^{v'}_{KP}$
if $\exists e_{v,w'} \in E^v_d \cup E^m_d$ and $v.param_i = \delta, w'.param_{j'} = \theta''$, $i \leq v.anz, j' \leq w'.anz$
with $\theta.name = \delta.name$ and $\theta''.type$ =in and

$\exists e_{v,w} \in E_d^r$ and $w.param_j = \theta$, $j \leq w.anz$ with
$\theta.name = \delta.name$ and $\theta.type =$`out` and w ist links von w'
and $w_u.param_k = \theta'$, $k \leq v_u.anz$ with $\delta.name = \theta'.name$.

- E_{KP}^r is defined as:

  · $e_{\theta'',\theta'} \in E_{KP}^r$
  if $\exists e_{v,w} \in E_d^r$ and $\exists e_{w,w'} \in E_d^v \cup E_d^m$ $w.param_i = \theta, w'.param_{j'} = \theta''$, $i \leq v.anz, j' \leq w'.anz$
  with $\theta.name = \theta''.name$ and $\theta.type =$`out`, $\theta''.type =$`out`
  and $w_u.param_k = \theta'$, $k \leq v_u.anz$ with $\delta.name = \theta'.name$.

  · $e_{\theta',\theta''} \in E_{KP}^r$
  if $\exists e_{v,w} \in E_d^r$ and $\exists e_{w,w'} \in E_d^v \cup E_d^m$ $w.param_i = \theta, w'.param_{j'} = \theta''$, $i \leq v.anz, j' \leq w'.anz$
  with $\theta.name = \theta''.name$ and $\theta.type =$`in`, $\theta''.type =$`in` and
  if $\nexists e_{w',v'} \in E_d^v \cup E_d^m$ and $v'.param_k = \delta'$, $k \leq v'.anz$
  with $\delta'.name = \theta''.name$ and $\delta'.type =$`in` and $w_d.param_k = \theta'$, $k \leq v_d.anz$
  with $\delta.name = \theta'.name$.

  · $e_{\theta',\delta'} \in E_{KP}^r$
  if $\exists e_{v,w} \in E_d^r$ and $\exists e_{w,w'} \in E_d^v \cup E_d^m$ $w.param_i = \theta, w'.param_{j'} = \theta''$, $i \leq v.anz, j' \leq w'.anz$
  with $\theta.name = \theta''.name$ and $\theta.type =$`in`, $\theta''.type =$`in` and
  if $\exists e_{w',v'} \in E_d^v \cup E_d^m$ and $v'.param_k = \delta'$, $k \leq v'.anz$
  with $\delta'.name = \theta''.name$ and $\delta'.type =$`in` and $w_d.param_k = \theta'$, $k \leq v_d.anz$
  with $\delta.name = \theta'.name$.

  For local parameters and result parameters the definitions are in the same way.

The set E_{KP}^v is the set of edges due to the normal data-dependendies. The sets $E_{KP}^{v'}$ and E_{KP}^r are sets of edges concerning cycles in the graph, i.e., the implementation of recursion.

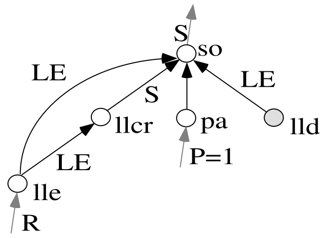
In the structure graph recursion is represented by backward arcs. In the dataflow graph, two new nodes are created, both representing the node the backward arc of the structure graph leads to. These new nodes v_d and v_u have attributes REC_d and REC_u. The first node v_d is for all input parameters, the second node v_u is for all output parameters. They are necessary, because some of the parameters of the recursive call may not be used in other literals, but only in the recursive call again. Therefore an arc would be created, going from v_d directly to v_d again. Or from v_u to v_u.

6.4 Example

The following examples recode parts of a control system for an autonomous road vehicle. They implement the task of steering a car from the right lane to the left lane of a road, to overtake a slower vehicle.

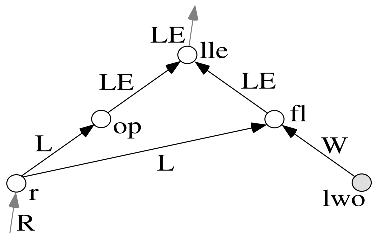
In the rules the "^{i}" denotes a variable of type input, the "^{o}" a variable of type output. The graphs correspond to the given program parts. The edges of the graphs depict the data-dependencies according to the given examples. All predicate names are denoted only with the first letter of each word of the name. The arcs marked in gray denote arcs for input and output of variables.

start_overtake(Roadtype^i*,Steering*^{o}*)* ←
    ll_exists(Roadtype^i*, LaneExists*^{o}*),*
    ll_detected(LaneExists^o*),*
    passing_allowed(1^i*),*
    ll_change_request(LaneExists^i*,Steering*^{o}*).*



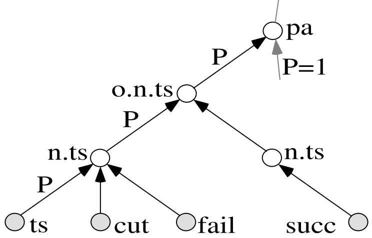
In this example, the rules *ll_exists* and *ll_detected* have the variable *LaneExists* for output, while the rule *ll_change_request* needs the variable for input. Thus an edge leads from the two nodes representing the former rules to the node representing the rule *ll_change_request*.

ll_exists(Roadtype^i*,LaneExists*^{o}*)* ←
    road(Roadtype^i*,Lanes*^{o}*),*
    own_position(Lanes^i*,LaneExists*^{o}*),*
    lanes_with_obstacle(Width^o*),*
    free_lanes(Lanes^i*,Width*^{i}*,LaneExists*^{o}*).*



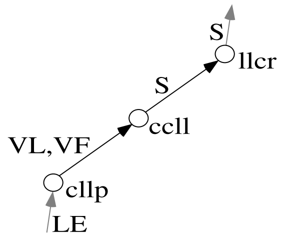
The variable *Roadtype* is used as input for the rule *ll_exists* and in the body of this rule for the rule *road*. Therefore in the graph the input is made directly to the node representing the rule *road*.

passing_allowed(P^i*)* ←
    not(trafficsign(pass^i*, P*^{i}*)).*

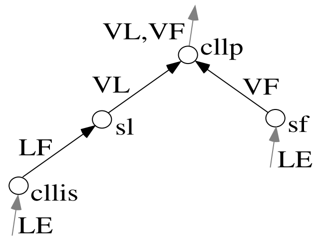


This graph shows an example for the implementation of negation.

ll_change_request(LaneExists^i*,Steering*^{o}*)* $\leftarrow$
change_ll_possible(LaneExists^i*, Velocityleft*^{o}*,*
Velocityfront^o*),*
calculate_change_ll(Velocityleft^i*,*
Velocityfront^i*,Steering*^{o}*).*



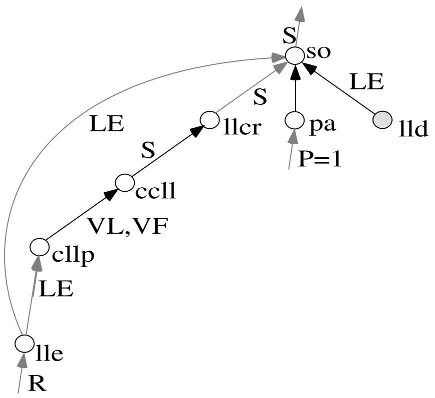
change_ll_possible(LaneExists^i*, Velocityleft*^{o}*,)*
Velocityfront^o*)*$\leftarrow$
change_ll_is_secure(LaneExists^i*,LaneFree*^{o}*),*
slow_left(LaneFree^i*, Velocityleft*^{o}*),*
slow_front(LaneExists^i*, Velocityfront*^{o}*).*



However, as the rules occur all together in a program, the graph is constructed in one single step. Therefore the next examples show how to connect the above graph fragments to gain the complete graph for the example program. Step by step connecting is done only for the purpose of better explanation.

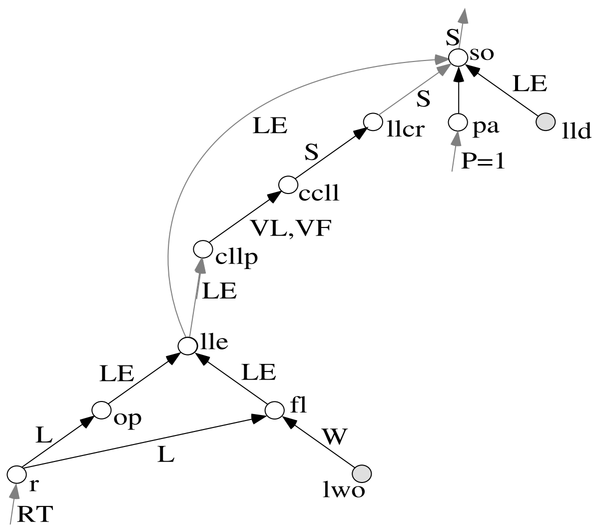
The grey arcs denote the connection of the different graph fragments, the gray nodes facts.

In the first example, the node *ll_change_request* in the graph for *start_overtake* is replaced by the graph for *ll_change_request*.

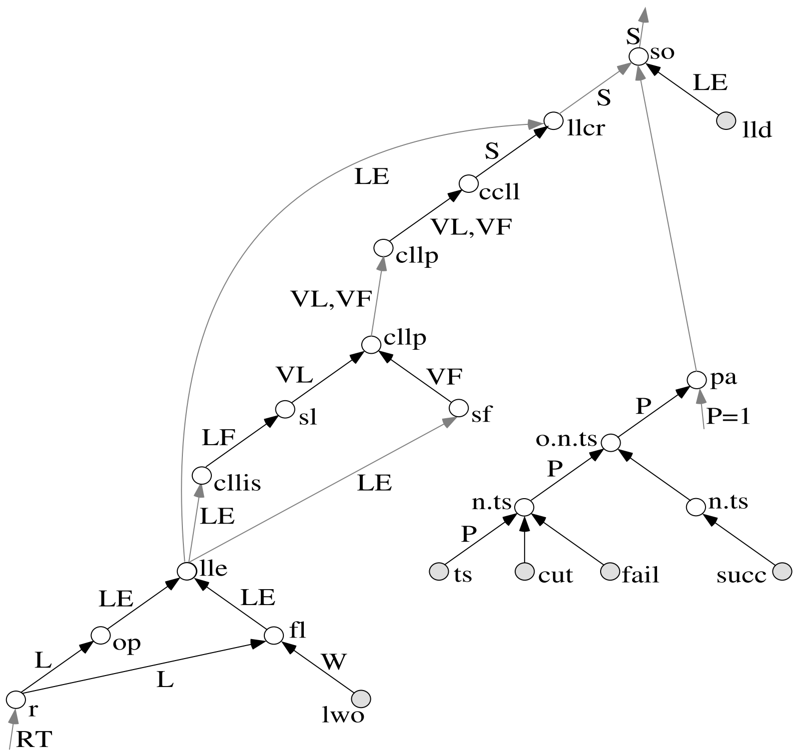


Next, in the resulting graph, the node *ll_exists* is replaced by the corresponding graph.

The output value of *LaneExists*, a parameter of the node *ll_exists*, is needed as an input value for the graph *ll_change_request* and within the graph for the node *change_ll_possible* Thus, an edge now goes directly from *ll_exists* to *ll_change_request*.



Next, the node *change_ll_possible* is replaced by the corresponding graph.



In this graph the nodes *change_ll_is_secure* and *slow_front* both need the value of *Lane-Exists* for input. Therefore, one edge connects the node *ll_exists* directly with the node *change_ll_is_secure* and one edge connects it directly with *slow_front*.

Finally the graph for the node *passing_allowed* is added.

Recursion, one main feature of Prolog, needs a special, more complex graph structure, but can be handled in the same way. Implementations of control systems have shown that cycles in the graph, i.e., recursion in the prolog-program, are rarely found, due to the lack of data organized as lists and due to real-time conditions that demand a straight forward organization of rules.

The dataflow graph is now a graph, where the arcs are directed from the facts, kept in a dynamic database, to the nodes that represent the other predicates, in respect to the sequence in which evaluation must take place.

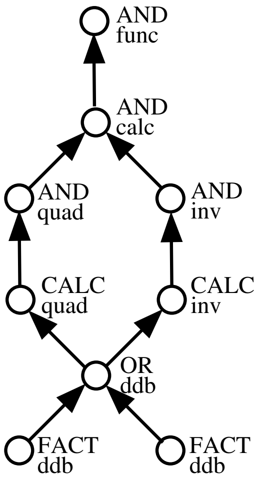


Figure 3: Dataflow graph

6.5 Parallelism

Subnodes of an AND-node, that are not connected via an arc, correspond to data-independent literals in the body of a rule. They are automatically executed in parallel, without any distinguishing marks being necessary. This implements restricted AND-parallelism.

Any node that receives values from another node via a connecting arc represents a literal that is data-dependent to another literal. However, the node does not have to wait for all incoming values of this arc, but can be executed as soon as the first value appears if there are already values on the other input arcs. This leads to the same amout of parallelism as by stream AND-parallelism.

Additionally, the child-AND-nodes of an OR-node independently produce values for the same variables. These values are collected and grouped together by the OR-node. Thus, full OR-parallelism is exploited.

A worst case analysis of the maximum time necessary for one evalution, and thus the longest time frame, can be done by determining the maximum depth of the dataflow graph, i.e., the longest path in the graph in view of the evaluation time for each node along the path.

Values are carried from node to node, i.e., from parameter to parameter by tokens. Thus tokens have to be specified, before defining the functionality.

Definition 6.16: ***Token***

A token is a tuple $TK = (A, \Omega,$ *uname, attr, itnr*), with $A =$ *(success, failure, cut)* and $\Omega = ($*dfsnr, nr,* W, $\omega)$, W is a set of arbitrary values, and

i) *uname*: $TK \to (\mathbb{N}, ..., \mathbb{N})$ is the unique name of the parameter that has produced the token.

ii) *attr*: $TK \to A$

$$tk.attr = \begin{cases} \textit{success} & \text{for a token with valid values or} \\ & \text{for a token signaling a successful calculation.} \\ \textit{failure} & \text{for a token signaling a failed calculation.} \\ \textit{cut} & \text{for the result of a cut node.} \end{cases}$$

iii) *itnr*: $TK \to \mathbb{N}$ is a value for the acutal depth of iteration.

iv) Ω is a tuple with:

- *dfsnr*: $\Omega \to \mathbb{N}$ is the "*dfsnr*" of the node the parameter, that has produced the value, belongs to.
- *nr*: $\Omega \to \mathbb{N}$ is a number for the order in which the values were calculated.
- $\omega : \Omega \to W$ is a value.

Tokens that are sent from node to node along the edges of the graph carry data from every output parameter to each corresponding input parameter. That means, each parameter of a node sends or receives its own token, carrying one or more data items, due to the possible occurance of alternative values gathered by an OR-node.

In addition, each node sends a result parameter signalling the *success* or *failure* of the evaluations already made. This result token does not contain any data.

If there is at least one input token with more than one value, a set of input tuples has to be formed:

- If a parameter receives values from different nodes, an intersection set has to be created from these values, thus leaving only values in the set that every subnode produces as a possible solution. Values that are not considered as a valid result from all subnodes will be deleted.
- Values that are sent from the same node are associated. Thus, any value associated with a value deleted, has to be deleted, too.
- Then a cartesian product has to be formed for all input values.

The result is the set of input tuples. An example will be given:

- if a node representing the literal f(X,Y) gets input values "2" and "3" for X from one subnode and the input value "4" for Y from another one, only a cartesian product has to be formed and the set of input tuples is ((2,4),(3,4)), each tuple containing values for X and Y.
- if a node representing the literal f(X,Y,Z) gets input tuples (2,8), (3,3) and (4,9) for X and Y from one subnode and input tuples (2,6), (2,7) and (3,5) for X and Z from another one, an first intersection set has to be formed for the values of X. The valid values for X are therefore "2" and "3". Thus resulting input tuples are generated then by a cartesian product over the resulting tuples (2,8) for X and Y with (2,6), (2,7) X and Z, because there are more then one tuple with the value "2" for X. The set of input tuples is thus ((2,8,6),(2,8,7),(3,3,5)) for X, Y and Z respectively.

First, some basic definitions have to be made.

Definition 6.17: ***Edges of Parameters***

The definition of a dataflow graph DG is extended:
$DG = (\mathcal{P}^e, SG^\delta, V, E_{KP}, F, attr, name, \tau, \pi, \phi)$

i) $\tau : KP \to E_{KP}$ is a function that outputs the input or output edges of a parameter ($\delta \in KP, e \in E_{KP}$): $\tau(\delta) = e$, if $e = e_{\delta,\theta} \vee e = e_{\theta,\delta}$ for arbitrary θ.

   τ is classified as:
   $\tau^O(\delta) = e$, if $e = e_{\delta,\theta}$, i.e., e is an output edge.
   $\tau^I(\delta) = e$, if $e = e_{\theta,\delta}$, i.e., e is an input edge.

ii) $\pi : E_{KP} \rightarrow TK$ is a function that assigns a token to an edge:
$(e \in E_{KP}, tk_1, ..., tk_n \in TK,\ n \geq 1)$.

$$\pi(e) = \begin{cases} \emptyset & \text{of there is no token on a edge } e \in E_{KP} \\ \{tk_1, ..., tk_n\},\ n \geq 1 & \text{if tokens } tk_1, ..., tk_n \in TK \\ & \text{are on the edge } e \in E_{KP}. \end{cases}$$

For simplifications of the following definitions some terms are introduced:

- $\#in$ = the amount of parameter $v.para_i$ for that $\tau^I(v.para_i)$ exists.
- $\#out$ = the amount of parameter $v.para_i$ for that $\tau^O(v.para_i)$ exists.
- i, j can have values of 1 to $\#in$, where only parameters are counted, with $v.para.type = \texttt{in}$, if they are not defined otherwise.
- i', j' can have values of 1 to $\#out$, where only parameters are counted, with $v.para.type = \texttt{out}$, if they are not defined otherwise.
- $\tau^I_k(v.para_i)$ is the k-th input arc of $v.para_i$.
- $\#\tau^I(v.para_i)$ is the amount of input arcs of $v.para_i$.
- k, l can have values of 1 to $\#\tau^I(v.para_i)$.
- If a token exists $\pi(\tau^I(v.para_i))$ it is called tk^I_i with $\Omega^I_i = \{\gamma^I_{i_1}, ..., \gamma^I_{i_n}\}$, $i_n \geq 1$.
- If a token exists $\pi(\tau^I_k(v.para_i))$ it is called $tk^I_{i^k}$ with $\Omega^I_{i^k} = \{\gamma^I_{i^k_1}, ..., \gamma^I_{i^k_n}\}$, $i^k_n \geq 1$.
- g, h can have values of 1 to i_n.
- For local parameters $v.anz$ is replaced by $v.lanz$, $v.para$ by $v.lpara$.
- For all output tokens ^{I} is replaced by ^{O}, `in` by `out`, $\#in$ by $\#out$, i by i', j by j'.
- For $v.res$ is $i = 0$.

Definition 6.18: ***Output Tuple of a Node***

A node v of a dataflow graph DG with parameters $v.para_{i'}$ has output tuples $(o_{0_g}, o_{1_g}, ..., o_{n_g})$, $n = \#out$, $g \geq 1$, $o_{i'_g} \in W$, $\forall i' \in \{1, ..., n\}$, with

- if $\forall i'\ \exists tk^O_{i'}$ with $tk^O_{i'} = \mathit{success}$ and $\gamma^O_{i'_g}.\omega = o_{i'_g}$, $\forall i' \in \{1, ..., n\}$.
  Furthermore, $o_{0_g} = \texttt{true}$.
- if $\not\forall i' :\ tk^O_{i'} = \mathit{success}$ then $g = 1$, $o_{i'_1} = \texttt{undef}\ \forall i' \in \{1, ..., n\}$.
  Furthermore, $o_{0_g} = \texttt{false}$.

Generally it holds:
$\forall i' :\ tk^o_{i'}.uname = v.para_{i'}.uname$,
$\forall i', g :\ \gamma^O_{i'_g}.nr = g$, $\gamma^O_{i'_g}.dfsnr = v.dfsnr$.

If there are more than one output edges for a parameter, the ouput token of the parameter has to be multiplied for each output edge. In the following this isn't mentioned explicitly.

Definition 6.19: *Input Tuple of a Node (not OR-Node)*

A node v of a dataflow graph DG with parameters $v.para_i$ has input tuples $(a_{1_g}, ..., a_{n_g})$, $n = \#in$, $g \geq 1$, $o_{i_g} \in W$, $\forall i \in \{1, ..., n\}$, with the tuples defined as follows, using only input tokens of the same iteration depth "*dfsnr*" from edges either of $E^v_{KP} \cup E^{v'}_{KP}$ or of $E^r_{KP} \cup E^u_{KP}$:

i) Corresponding values of a value of a parameter are combined to a set M :

$\forall i\ \exists tk^I_i$ with $\Omega^I_{i^k} = \{\gamma^I_{i^k_1}, ..., \gamma^I_{i^k_n}\}$, $i^k_n \geq 1$, then
$M_{i^k_g} = \{ \prod_{j \neq i, l \neq k} \{\gamma^I_{j^l_g}\} | \gamma^I_{i^l_g}.nr = \gamma^I_{i^k_g}.nr,\ \gamma^I_{i^l_g}.dfsnr = \gamma^I_{i^k_g}.dfsnr\}$.

Note: $\gamma^I_{i^k_g} \notin M_{i^k_g}$.

ii) For the values of a parameter with only one input edge, there is nothing to do.

If $\#\tau^I(v.para_i) = 1$, then $N_{i^k_g} = \gamma^I_{i^k_g} \times M_{i^k_g}$.

iii) For the values of a parameter with more than one input edge, that do occur in each token on each input egde, a cartesian product has to be formed.

If $\#\tau^I(v.para_i) \geq 1$ and $\exists l \neq k, h \neq g$ with $\gamma^I_{i^l_h}.\omega = \gamma^I_{i^k_g}.\omega$, then
$N_{i^k_g} = \{\gamma^I_{i^k_g} \times \prod_l \{\bigcup_h M_{i^l_h} | \gamma^I_{i^l_h}.\omega = \gamma^I_{i^k_g}.\omega\} |\ \gamma^I_{i^l_h}.\omega = \gamma^I_{i^k_g}.\omega\}$.

and $\forall l \neq k, h \neq g$ with $\gamma^I_{i^l_h}.\omega = \gamma^I_{i^k_g}.\omega$: $N_{i^l_h} = \emptyset$.

It holds that:

- $M_{i^k_g} \in N_{i^k_g}$
- $N_{i^k_g}$ contains no tuples with the same elements.

iv) The values of a parameter with more than one input edge, that do occur in each token on each input egde have to be deleted.

If $\#\tau^I(v.para_i) \geq 1$ and $\forall l, h:\ l \neq k, h \neq g$ with $\gamma^I_{i^l_h}.\omega \neq \gamma^I_{i^k_g}.\omega$, then $N_{i^k_g} = \emptyset$.

v) $\mathcal{N} = \gamma'_{0_g} \times \prod_{i,k,g} N_{i^k_g}$ with i, k, g in a way, that $N_{i^k_g}$ exists.

Then $(\gamma'_{0_g}, \gamma'_{1_g}, ..., \gamma' n_g) \in \mathcal{N}$ and the input tuples are
$(a_{0_g}, ..., a_{n_g})$ with $\gamma'_{i_g}.\omega = a_{i_g}\ \forall i \in \{1, ..., v.anz\}$.

Example: If a node has parameters X, $Y1$, $Y2$ and Z and gets the following tokens from other nodes:
$(\gamma^I_{i^j_g} = (dfsnr, nr, \omega))$

- For X of node 1: $tk^I_{1^1}$ with
$\gamma^I_{1^1_1} = (1,1,\mathbf{5}), \gamma^I_{1^1_2} = (1,2,\mathbf{5}), \gamma^I_{1^1_3} = (1,3,\mathbf{6}), \gamma^I_{1^1_4} = (1,4,\mathbf{7})$
- For $Y1$ of node 1: $tk^I_{2^1}$ with
$\gamma^I_{2^1_1} = (1,1,\mathbf{1}), \gamma^I_{2^1_2} = (1,2,\mathbf{2}), \gamma^I_{2^1_3} = (1,3,\mathbf{3}), \gamma^I_{2^1_4} = (1,4,\mathbf{4})$
- For $Y2$ of node 1: $tk^I_{2^1}$ with
$\gamma^I_{3^1_1} = (2,1,\mathbf{0}), \gamma^I_{3^1_2} = (2,2,\mathbf{8}), \gamma^I_{3^1_3} = (2,3,\mathbf{9}), \gamma^I_{3^1_4} = (2,4,\mathbf{10})$
- For X of node 2: $tk^I_{1^2}$ with
$\gamma^I_{1^2_1} = (2,1,\mathbf{5}), \gamma^I_{1^2_2} = (2,2,\mathbf{5}), \gamma^I_{1^2_3} = (2,3,\mathbf{7}), \gamma^I_{1^2_4} = (2,4,\mathbf{7})$
- For Z of node 2: $tk^I_{3^1}$ with
$\gamma^I_{4^1_1} = (2,1,\mathbf{11}), \gamma^I_{4^1_2} = (2,2,\mathbf{12}), \gamma^I_{4^1_3} = (2,3,\mathbf{13}), \gamma^I_{4^1_4} = (2,4,\mathbf{14})$

Then
$M_{1^1_1} = \{(\gamma^I_{2^1_1}, \gamma^I_{3^1_1})\}$, $M_{1^1_2} = \{(\gamma^I_{2^1_2}, \gamma^I_{3^1_2})\}$, ..., $M_{2^1_1} = \{(\gamma^I_{1^1_1}, \gamma^I_{3^1_1})\}$, ...

The set $N_{1^1_1} = \gamma^I_{1^1_1} \times (M_{1^1_1} \cup M_{1^1_2}) \times (M_{1^2_1} \cup M_{1^2_2})$ is:
$\{(\gamma^I_{1^1_1}, \gamma^I_{2^1_1}, \gamma^I_{3^1_1}, \gamma^I_{4^1_1}),\ (\gamma^I_{1^1_1}, \gamma^I_{2^1_1}, \gamma^I_{3^1_1}, \gamma^I_{4^2_1}),\ (\gamma^I_{1^1_2}, \gamma^I_{2^1_2}, \gamma^I_{3^2_1}, \gamma^I_{4^1_1}),\ (\gamma^I_{1^1_2}, \gamma^I_{2^1_2}, \gamma^I_{3^2_1}, \gamma^I_{4^2_1})\}$
because $\gamma^I_{1^1_1}.\omega = \gamma^I_{1^1_2}.\omega = \gamma^I_{1^2_1}.\omega = \gamma^I_{1^2_2}.\omega$.

If only the values of this set are displayed, the set is:
$\{(\mathbf{5},\mathbf{1},\mathbf{0},\mathbf{11}), (\mathbf{5},\mathbf{1},\mathbf{0},\mathbf{12}), (\mathbf{5},\mathbf{2},\mathbf{8},\mathbf{11}), (\mathbf{5},\mathbf{2},\mathbf{8},\mathbf{12})\}$

If of the sets $N_{1^1_2}, N_{1^2_1}, N_{1^2_2}$ the values are also displayed solely, the same sets are gained. Because only the values are important for the input, these (duplicate) sets can be deleted.

The set $N_{1^1_3} = \emptyset$, because $\exists i \in \{1,...,4\} : \gamma^I_{1^2_i}.\omega = \gamma^I_{1^1_3}$.

The set $N_{1^1_4} = \gamma^I_{1^1_4} \times M_{1^1_4} \times (M_{1^2_3} \cup M_{1^2_4})$,
if only the values are displayed, is: $\{(\mathbf{7},\mathbf{4},\mathbf{10},\mathbf{13}), (\mathbf{7},\mathbf{4},\mathbf{10},\mathbf{13})\}$

The sets $N_{1^2_3}, N_{1^2_4}$ are identically, iff only the values are shown.

All input tuples are therefore:
$\{(\mathbf{5},\mathbf{1},\mathbf{0},\mathbf{11}), (\mathbf{5},\mathbf{1},\mathbf{0},\mathbf{12}), (\mathbf{5},\mathbf{2},\mathbf{8},\mathbf{11}), (\mathbf{5},\mathbf{2},\mathbf{8},\mathbf{12}), (\mathbf{7},\mathbf{4},\mathbf{10},\mathbf{13}), (\mathbf{7},\mathbf{4},\mathbf{10},\mathbf{13})\}$

Definition 6.20: *Input Tuples of a Node (OR-Node)*

A node v of a dataflow graph DG with parameters $v.para_i$ has input tuples
$(a_{1_g}, ..., a_{n_g})$, $n = \#in$, $g \geq 1$, $o_{i_g} \in W$, $\forall i \in \{1,...,n\}$, with the tuples defined as

follows, using only input tokens of the same iteration depth "*itnr*" from edges either of $E^{v}_{KP} \cup E^{v'}_{KP}$ or of $E^{r}_{KP} \cup E^{u}_{KP}$:

i) $\forall i\ \exists tk^I_i$ with $\Omega^I_{i^k} = \{\gamma^I_{i^k_1}, ..., \gamma^I_{i^k_n}\}$, $i^k_n \geq 1$, then
$M_{i^k_g} = \{\prod_{j,l}\{\gamma^I_{j^l_g}\} | \gamma^I_{i^l_g}.nr = \gamma^I_{i^k_g}.nr,\ \gamma^I_{i^l_g}.dfsnr = \gamma^I_{i^k_g}.dfsnr\}$.

Therefore, it holds that $M_{i^k_g} = M_{j^l_g}$, if $\gamma^I_{i^l_g} \in M_{i^k_g}$.

ii) $\mathcal{N} = \bigcup_{i,k,g} M_{i^k_g}$.

Therefore, it holds that $\mathcal{N}$ has no two equal elements.

Then $(\gamma'_{0_g}, \gamma'_{1_g}, ..., \gamma' n_g) \in \mathcal{N}$ and the input tuples are
$(a_{0_g}, ..., a_{n_g})$ with $\gamma'_{i_g}.\omega = a_{i_g}\ \forall i \in \{1, ..., v.anz\}$.

Example: If an OR-node has parameters X and Y and gets the following tokens from to other nodes:
$(\gamma^I_{i^j_g} = (dfsnr, nr, \omega))$
- For X of node 1: $tk^I_{1^1}$ with
$\gamma^I_{1^1_1} = (1, 1, \mathbf{5}), \gamma^I_{1^1_2} = (1, 2, \mathbf{5}), \gamma^I_{1^1_3} = (1, 3, \mathbf{6}), \gamma^I_{1^1_4} = (1, 4, \mathbf{7})$
- For Y of node 1: $tk^I_{2^1}$ with
$\gamma^I_{2^1_1} = (1, 1, \mathbf{1}), \gamma^I_{2^1_2} = (1, 2, \mathbf{2}), \gamma^I_{2^1_3} = (1, 3, \mathbf{3}), \gamma^I_{2^1_4} = (1, 4, \mathbf{4})$
- For X of node 2: $tk^I_{1^2}$ with
$\gamma^I_{1^2_1} = (2, 1, \mathbf{5}), \gamma^I_{1^2_2} = (2, 2, \mathbf{8})$
- For Y of node 2: $tk^I_{3^1}$ with
$\gamma^I_{2^2_1} = (2, 2, \mathbf{11}), \gamma^I_{2^2_2} = (2, 2, \mathbf{9})$

Then
$M_{1^1_1} = \{(\gamma^I_{1^1_1}, \gamma^I_{2^1_1})\}$, $M_{1^1_2} = \{(\gamma^I_{1^1_2}, \gamma^I_{2^1_2)}\}$, ..., $M_{2^2_1} = \{(\gamma^I_{1^1_2}, \gamma^I_{2^2_2})\}$.

The set $\mathcal{N} = M_{1^1_1} \cup M_{1^1_2} \cup M_{1^2_1} \cup M_{1^2_2} \cup M_{2^1_1} \cup M_{2^1_2}$.

The sets $M_{1^1_1}$ and $M_{1^1_2}$ are identically, as is $M_{1^2_1}$ and $M_{1^2_2}$, $M_{2^1_1}$ and $M_{2^1_2}$

If only the values of the tuples in $\mathcal{N}$ are displayed, then the input tuples are:
$\{(\mathbf{5}, \mathbf{2}), (\mathbf{7}, \mathbf{4}), (\mathbf{8}, \mathbf{9})\}$.

6.5.1 Implementation of restricted-AND/OR parallelism

Definition 6.21: ***Functionality of restricted AND/OR-parallelism in a Dataflow Graph***

The definition of a dataflow graph DG is extended to
$DG = (\mathcal{P}^e, SG^\delta, V, E_{KP}, F, attr, name, \tau, \pi, \phi)$

$\phi : (KP \times ... \times KP) \to (KP \times ... \times KP)$ is defined as follows:

1. $v.attr$ = SUCC: $tk^O_{0^1}.attr = success.$
   The output tuple is (o_{0_g}).

2. $v.attr$ = FAIL: $tk^O_{0^1}.attr = failure.$
   The output tuple is (o_{0_g}).

3. $v.attr$ = CUT: $tk^O_{0^1}.attr = cut.$
   The output tuple is (o_{0_g}).

4. $v.attr$ = FACT:
   If $\exists i : v.para_i.type =$ `in` and
   $\forall i : v.para_i.type =$ `in` $\exists tk^I_i$, then for all input tuples $(a_{1_g}, ..., a_{n_g})$, $n = \#in$, $g \geq 1$ the output tuples are $(o_{0_g}, o_{1_g}, ..., o_{n_g})$, $n = \#out$, $g \geq 1$, or
   if $\not\exists i : v.para_i.type =$ `in`, then the output tuple is $(o_{0_1}, o_{1_1}, ..., o_{n_1})$, $n = \#out$, with:

   - if $\exists i' : para'_i.type =$ `const`, then
     $\gamma_{i'^g}.\omega = o_{i'_g} = a$,
   - if $\exists i' : para'_i.type =$ `out` and
     $\not\exists i : v.para_i.type =$ `in`, $v.para'_i.name = v.para_i.name$, then
     $\gamma^O_{i^g}.\omega = o_{i'_g}$ = the value read out of the database ,
   - if $\exists i' : para'_i.type =$ `out` and
     $\exists i : v.para_i.type =$ `in`, $v.para'_i.name = v.para_i.name$, then
     $\gamma^O_{i'^g}.\omega = a_{i_g}.\omega$.

5. $v.attr$ = FUNC:
   If $\exists i : v.para_i.type =$ `in`, then for all input tuples $(a_{1_g}, ..., a_{n_g})$, $n = \#in$, $g \geq 1$ the output tuples are $(o_{0_g}, o_{1_g}, ..., o_{n_g})$, $n = \#out$, $g \geq 1$, or
   if $\not\exists i : v.para_i.type =$ `in`, then the output tuple is $(o_{0_1}, o_{1_1}, ..., o_{n_1})$, $n = \#out$, with:

   - if $\exists i' : para_{i'}.type =$ `out`, then $\gamma^O_{i'_g}.\omega = o_{i'_g}$, $\gamma^O_{0_g}.\omega =$ `true`
     if the function could be executed successfully for inputs a_{i_g}.
   - if $\exists i' : para_{i'}.type =$ `out`, then $\gamma^O_{i'_g}.\omega =$ `undef`, $\gamma^O_{0_g}.\omega =$ `false`
     if the function could not be executed successfully for inputs a_{i_g}.

   Therfore, it is:

   - $\forall i' :\ tk^O_{i'}.attr = success$, $tk^O_0.attr = success$, if $\exists g :\ \gamma^O_{0_g}.\omega =$ `true`.

- $\forall i' : tk^O_{i'}.attr = failure$, $tk^O_0.attr = failure$, if $\nexists g : \gamma^O_{0_g}.\omega = \mathtt{true}$.
  The output then is $(o_{0_1}, o_{1_1}, ..., o_{n_1})$.

6. $v.attr$ = CALC:
   The functionality is the same as for a FUNC node, except that the term function is replaced by the term operations.

7. $v.attr$ = AND or $v.attr$ = QUERY:
   For all input tuples $(a_{1_g}, ..., a_{n_g})$, $n = v.anz$, $g \geq 1$ are
   the output tuples $(o_{0_g}, o_{1_g}, ..., o_{n_g})$ with $o_{i_g} = a_{i_g}$, $\forall i \in \{1, ..., n\}$.
   - $\forall i : tk^O_i.attr = success$, $tk^O_0.attr = success$, if $\exists g : \gamma^O_{0_g}.\omega = \mathtt{true}$.
   - $\forall i : tk^O_i.attr = failure$, $tk^O_0.attr = failure$, if $\nexists g : \gamma^O_{0_g}.\omega = \mathtt{true}$.
     The output then is $(o_{0_1}, o_{1_1}, ..., o_{n_1})$.

8. $v.attr$ = OR:
   For all input tuples $(a_{1_g}, ..., a_{n_g})$, $n = v.anz$, $g \geq 1$ are
   the output tuples $(o_{0_g}, o_{1_g}, ..., o_{n_g})$ with $o_{i_g} = a_{i_g}$, $\forall i \in \{1, ..., n\}$.
   - $\forall i : tk^O_i.attr = success$, $tk^O_0.attr = success$, if $\exists g : \gamma^O_{0_g}.\omega = \mathtt{true}$.
   - $\forall i : tk^O_i.attr = failure$, $tk^O_0.attr = failure$, if $\nexists g : \gamma^O_{0_g}.\omega = \mathtt{true}$.
     The output then is $(o_{0_1}, o_{1_1}, ..., o_{n_1})$.

9. $v.attr$ = REC$_d$:
   For all input tokens of iteration depth n, the output tokens have the iteration depth $n + 1$. Furthermore, each input token passes the node without change.

10. $v.attr$ = REC$_u$:
    The node realizes AND-functionality. For all input tuples of tokens of iteration depth $n + 1$, the iteration depth for outgoing tokens is n.
    After all values of the input tokens have been consumed, the input tokens get deleted.

In a more informal description the functionality is:

1. CUT-, FAIL- and SUCC-node
   A CUT-node always produces a *cut*-Token, a FAIL-node a *failure*-token and a SUCC-node a *success*-token for the result parameter. These tokens do not contain data.

2. FACT-node
   If all parameters of a FACT-node are of type output, the FACT-node produces a *success*-token for each parameter with the output data for this parameter. This

may either be a constant, stored in the FACT, or a value read out of the data base at the beginning of a time frame. Changings of a value in the data base during one time frame only affect the output for the next time frame.
If there is at least one parameter of type input, the FACT-node has to wait until all input tokens have arrived. Then, the data can be read out and passed onward, along with other data of the data base.
The result-token of a FACT-node is always a *success*-token.

3. FUNC-node
   A FUNC-node can be executed if all input tokens are present on the input arcs. Constants, stored in the node, can be used for input directly.
   If there are input tokens with more than one value, the function has to be executed more than once, with each correct combination of input values, i.e., a tuple out of the set of input tuples.
   If the execution of the function was successful, a *success*-token is generated for each output parameter, containing the values of all executions for this parameter. The result-token then is a *success*-token, too.
   If the execution of the function fails for all input tuples, a *failure*-token is generated for each parameter and the result-parameter.

4. CALC-node
   A CALC-node can be executed if all input tokens are present on the input arcs. Constants can be used for input directly.
   If there are input tokens with more than one value, the function has to be executed more than once, for all input tuples of the set of input tuples.
   If the operation of a CALC-node is an assignment that could be made successfully, a *success*-token is generated for the output parameter with the result of the assignment, along with a *success*-token for the result-parameter. Otherwise a *failure*-token is generated for both.
   If the operation is a comparison, no output parameter exists. The result-token is a *success*-token if the comparison was successful, otherwise it is a *failure*-token.

5. AND- and QUERY-node
   As the conjunctive combination of input values is already done by generating the set of input tuples, the AND-node has only to send each tuple of the set of input tuples further on.
   The functionality of a QUERY-node is the same.

6. OR-node
   If all subnodes of the OR-node deliver a *success*-token for each parameter the values are gathered and sent on with a new *success*-token. If there is a *failure*-token coming

from a subnode, it is ignored. If there are only *failure*-tokens from each subnode, the OR-node sends *failure*-tokens for each parameter and the result-parameter further on.

7. $v.attr$ = REC_d:
   Normally, input values would be sent directly to the input parameters, which is not possible at a recursion call. Therefore, the node REC_d was introduced. All tokens pass this node without any change of the values, but the iteration depth is increased to mark an new recursion cycle.

8. $v.attr$ = REC_u:
   The node realizes the output of a recursion call. It implements AND-functionality because each output parameter would have to pass the head of the recursive rule, i.e., an AND-node, in each iteration step. Furthermore, the iteration depth is decreased.

This realizes the functionality of *restricted AND-parallelism* in Prolog.

- All subnodes of an OR-node can be executed in parallel.
- All data-independent subnodes of an AND-node can be executed in parallel.
- All data-dependent subnodes of an AND-node are executed in sequence, triggerend by the arrival of data values.
- If there is more than one value for a parameter, these values are delivered together.

6.5.2 Implementation of stream AND/OR-parallelism

For the implementation of stream AND/OR-parallelism only a few things have to be changed. Tokens now carry only one value, cause each value produced by a parameter is delivered seperately, thus implementing a kind of stream functionality on the edges.

Tokens can now arrive not only seperately, but in an arbitrary ordering. Values have therefore to be saved at the parameters of a node until graph execution is finished completely.

Again, there may be more than one value, i.e., token for a parameter and therefore other values have to be combined more than once with these values for input, thus creating the same input tuples as is done for restricted AND/OR-parallelism.

The functionality of the nodes changes as follows:

- The definition of parameters of a node is extended by the set Ω of the definition of tokens: $KP = (T_e, I, IOT, \Omega$ *name, uname, history, type*), with Ω = (*dfsnr, nr, W*, ω) as defined for tokens.

- Each value of a token is saved in the set Ω of the parameter along with "dfsnr" and "nr". The token is then deleted.
- The input tuples are now no longer generated from the sets Ω of the tokens but from the sets Ω of the parameters of the node.
  For each new value arriving at a parameter that is not identical to a stored value the set $\mathcal{N}_g$ of input values is created if for all input parameters values are present. If the set is not empty, the node is called with each input tuple of $\mathcal{N}_g$.
  The condition that $\mathcal{N}_g$ is only created if the new value is different from stored ones, guarantees, that $\mathcal{N}_g$ has input tuples only that are also different from old input tuples.
  If a new value arrives and there are still not sufficient input values for the other parameters that a input tuple could be generated, then the set $\mathcal{N}_g$ is empty.
- All saved values at parameters are deleted as soon as the graph evaluation is finished.

7 Improving the Approach by Activity-directed Computation

A special feature of the static dataflow execution model is that it can easily be modified to an activity-directed model of computation, defined by the following three rules, i.e., the way a rank-ordered dataflow graph is evaluated [16]:

- Selection rule
  Only those nodes are selected for possible evaluation at which no further tokens can arrive on an input arc, i.e., all nodes placed on the same rank become selected.
- Eventflow firing rule
  A node that passes the selection rule is evaluated if and only if the node received at least one token on any of its input arcs after its last evaluation.
- Evaluation and result distribution rule
  Node evaluation, i.e., execution of the node's function, only produces a result (event) if this is distinguishable from the previous one, produced and stored as "last result". In this case, the result is transmitted by means of a token to each sucessor node.

The rank ordering of the graph can be done by simple rules:

- Rank 0 is assigned to the leaves of the graph, i.e., all nodes with parameter only of type output.
- Rank $n+1$ is assigned to each node where an forward arc from a node with rank n leads to, but no forward arc from a node with rank $\geq n$,

In the selection rule all nodes of rank $n+1$ get selected, if all nodes of rank n have been selected, examined and executed if necessary. If there are backward arcs and token were sent on them to nodes of rank $m_1 \leq n, m_2 \neq n$, etc..., despite of all nodes of rank $n+1$ all nodes of the lowest rank out of rank $m_1, m_2, ...$ are selected again.

Values for a parameter have to be stored at the parameter and neither to be deleted after node execution nor after finishing the graph evaluation, because they are needed in calculations later on in case they do not change.

If a new value for the parameter arrives, it may be an update for an old value as well as an additional value, since there may be more than one input value for a parameter. Therefore, if arriving values have the same "nr" and "dfsnr" as a stored value, the stored value is overwritten by the new one. If there is no old value with the same "nr" and "dfsnr" the value is simply appended to the list of already stored values.

If a value is not valid anymore, i.e., a node does not produce a result it has produced before anymore, it has to send a "delete"-token with the same "nr" and "dfsnr" of the old value to the nodes, that received the old value for input. By this, following nodes are told, that the input value has to be deleted. All results that a node has calculated before by using the old value for input, have to be deleted, too. Therefore, for each result, a list has to be stored of the "nr" and "dfsnr" of all input values, that were used to calculate the result. For every deleted result, the node has to send a "delete"-token further on.

There would be no sense in implementing activity-directed execution, if at each node near the bottom of the graph at least one change occurs. But the amount of events flowing in a region of a graph depends on the type of facts:

- General data about the system (maximum velocity, maximum ascent, time conditions for an emergency stop, etc..)
- General data about the mission
- General knowledge about the type of environment (street, road, off-road, etc.)
- Data about the mission progress
- Data about the state of the autonomous system (moving, turning, accelerating, etc...)
- Sensor data of the actual environment

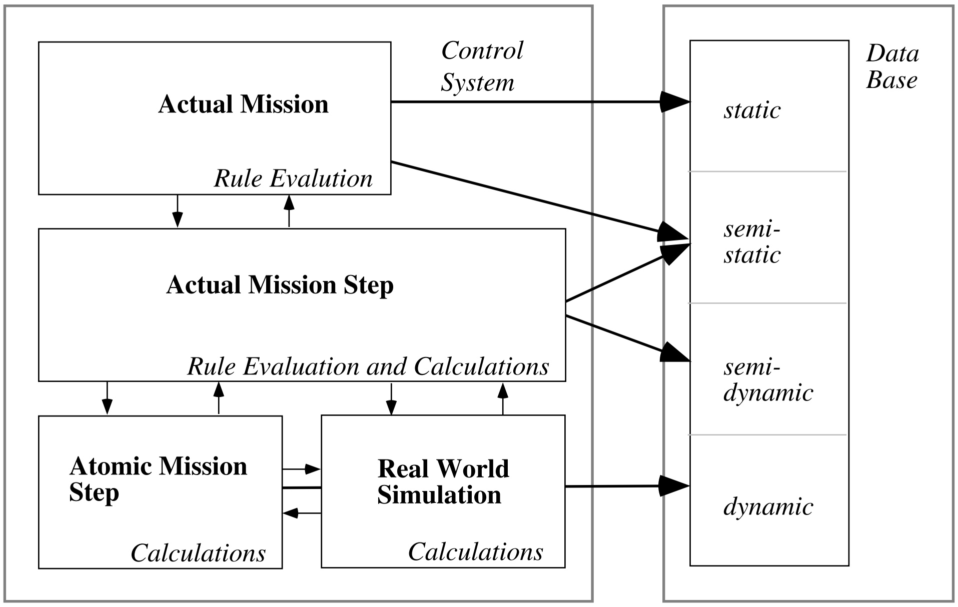


Figure 4: Corresponding hierarchical structure and datatypes

Among these types, general data about the system are the most static ones, whereas sensor data are the most dynamic ones. Furthermore, implementations have shown, that each type of data affects a different cone of the graph. Figure 4 shows how the different parts of a control system are related to different types of data.

Therefore, the advantage of the eventflow model of execution clearly is that only parts of the graph must be evaluated per time frame, since there are cones that depend on data that do not change from time frame to time frame. Concerning ideal hardware with an infinite number of processors, only the depth of the graph sets the measure for evaluation time. In reality, however, there is a restricted number of processors and every node that has not to be evaluated shortens the reaction time of the control system.

For activity-directed dataflow computing, the node's functionality has to be changed as follows:

- Values arriving at a parameter have to be stored at the parameter even in restricted AND/OR-parallelism. They must not be deleted at the end of graph evaluation.
- Values have to be labelled to identify if a new value that arrives at a parameter represents an additional value, i.e., if none of the stored values has the same label, or if it replaces a stored one, i.e., if it has the same label as a stored values.
- A value that is no longer valid, because it is no longer produced, has to be deleted by a *failure*-token with the same markings, produced by the node that does not produce the value any more.

8 Combining Calculations and Rule Evaluations in the Modell

However, the better the control system is to take into account real-world-knowledge, the more complex the set of rules becomes. Even a most efficient implementation of rule-based acting does not allow reaction times close to real time conditions for autonomous systems.

Therefore it is part of this new approach to rely on the one hand on rule evaluation for high level decisions and on the other hand on mathematical calculations for the basic behaviour in combination with calculating real-world-models.

These real world models include a representation of all objects, that may occur in the environment, including the autonomous system itself, in 4 dimensions: the 3D shape models allow for recognition and tracking of objects in the real world, while the fourth dimension of time models the object's possible motion.

Simulating real-world models allows on the one hand for a simplified evaluation of sensor data, i.e., recognition of objects in the real-world due to knowledge about their possible position and shape by predicting the most probable object's behaviour. For safety reasons the predicted behaviour is compared with actual behaviour. Differences may be used to tune the object models. On the other hand, control outputs for the basic bahaviour can be calculated more optimal taking process models into account. And, even planning and deciding longer-term actions can be simplified by using dynamic process models [13].

Combining mathematical calculations with the evaluation of rules within the proposed dataflow model is easy: In a Prolog-program, complex mathematical functions can be called by a *metapredicate*. The result of the execution of the metapredicate is given to the program via an output parameter. On the implementation level, calculations can be expressed as dataflow graphs as well as rules [11]. Complex data structures, e.g., arrays, can be handled by I-structures [3]. Thus, the node representing a metapredicate is simply replaced by the graph for the calculation.

As a result, there is no need to implement two different modules in the control system, each one with a different evalution model. And there is also no need to implement an interface and a protocoll to exchange data, which would be very time-consuming.

It is the unique advantage of this approach that rule-based acting as well as simulation-oriented computing can be coded by dataflow graphs in a unified way and both be implemented in one module.

9 Results

The implementation of the proposed approach consists of two parts: The first part is a compiler that transforms any given Prolog-program into a dataflow graph according to the algorithm given above. The second one is an interpreter for the resulting dataflow graphs, for testing the correctness of the transformation and for estimating the degree of efficiency.

The next step is the evaluation of a series of knowledge-based control systems for a comparison of the evaluation time for the original implementation with the equivalent solution due to this approach.

Additionally, there will be a simulation environment for the special implementation of a control system for autonomous vehicles.

Conclusion

The dataflow computation model takes advantage of the fact that knowledged-based control of autonomous systems considerably differs from the task of general-purpose expert systems.

Specifying knowledge-based control for concurrent evaluation can be done in any parallel Prolog language. But only the approach of this report allows to exploit parallelism without special language-constructs in the logic program.

The general advantage of this new dataflow model is that there is no need for demand-driven execution, i.e., it is not necessary that evaluation goes through the graph from a query to the facts first and then calculated data is passed the whole way back. Now, activity stimulated by changes of facts only goes through the graph bottom-up to the queries. Further, even a new calculation, distinguished from a previous one by a different “color”, can be started before the previous one has finished. This leads to higher evaluation frequency and thus supports control in real-time.

Furthermore, the eventflow evaluation, a special evaluation method for dataflow graphs, allows increased complexity for expert systems without reducing the possiblity to fulfil real-time requirements, since in each iteration only small parts of the system have to be updated.

There is a wide application field for intelligent autonomous mobile systems. It ranges from fully autonomous robots that operate in live-dangerous environments, e.g., robots for mine clearing [10], to decision supporting systems, e.g., navigation help- and warning systems for vehicles [8].

Various autonomous vehicles have already been developed, e.g., the Autonomous Land Vehicle (ALV) at the Hughes Laboratories at Denver [1], Navlab, a vehicle, that is able to drive on roads as well as off-road, at the Carnegie Mellon University, or ROVA (Road Vehicle Autonomous), achieving a velocity of 40 km/h, as the result of a British project [6]. Companies like Nissan and Toyota work on comparable projects [24].

Due to its maximum velocity of 90 km/h on highways, the most improved one is “VaMoRs”, an autonomous mobile, developed by Prof. Dr. Dickmanns, University of the Federal Armed Forces, at Munich, Germany [12]. Its control system contains an “intelligent” module, Behavior Decision, recently coded in C [19]. This module serves not only as a prove for the suitability of activity-directed dataflow computing for knowledge-based control of autonomous systems by real world data, but a reduced calculation time can open the way on the one hand to a more complete set of rules or, on the other hand, to further improvement of the maximum velocity.

References

[1] S. Amarel, "AI Research in DARPAs Strategic Computing Initiative", *IEEE expert*, No. 3, 1991, pp. 7-11

[2] Arvind, R. E. Thomas, "I-Structures: An efficient Data Type for Functional Languages", TM-178, Laboratory for Computer Science, MIT, Cambridge, 1980

[3] Arvind, R. S. Nikhil, "I-Structures: Data Structures for Parallel Computing", *ACM Transactions on Programming Languages and Systens*, Vol. 11, Nr. 4, 1989, pp. 598-632

[4] P. J. Antsaklis, K. M. Passino, S. J. Wang, "An Introduction to Autonomous Control Systems", *IEEE Control Systems*, No. 4, 1991, pp. 5-13

[5] L. Bic, C. Lee, "A Data-Driven Model for a Subset of Logic Programming" *ACM Transactions on Programming Languages and Systems*, Vol. 9, 1984, pp. 618-645

[6] C. P. Blackman, "The ROVA nd MARDI Projects" *IEEE Colloquium on Advanced Robotic Initiatives in the UK* (Digest No. 081), 1991

[7] I. Bratko, "Prolog, Programming for Artificial Intelligence" Addison-Wesley Verlag, 1987

[8] C. Brüdigam, " Intelligente Fahrmanöver sehender autonomer Fahrzeuge in autobahnähnlicher Umgebung", Dissertation, Universität der Bundeswehr, München, Fakultät für Luft- und Raumfahrttechnik, 1994

[9] K. L. Clark, S. Gregory, "PARLOG: Parallel Programming in Logic" *Research Report 84/4* Departement of Computing, Imperial College of Science and Technology, England, 1984

[10] J. Cornelius, H. Sahli, M. Acheroy, Y. Baudoin, "Anti Personal Mines, a world-wide problem: From political conscience towards humanitarian, resaerch and industrial action", *Proceedings of the sixth International Symposium on Measurement and Control in robotics*, ISMCR'96, Bussels, Belgium, 1996, pp. 1-5

[11] J. B. Dennis, "Data Flow Supercomputers", *Computer*, Vol. 13, No. 2, 1980, pp. 48-56

[12] E. D. Dickmanns, "Automobile lernen sehen" Universität der Bundeswehr, München, Fakultät für Luft- und Raumfahrttechnik, 1992.

[13] E. D. Dickmanns, *Vehicles capable of dynamic vision* Proc. IJCA-97 15th International Joint Conference on Artificial Intelligence, Nagoya, Japan, 1997

[14] T. Dodd, "Prolog: A logical approach", Oxford Science Publications, 1990

[15] Z. Halim, "A Data Driven Machine for OR-Parallel Evaluation of Logic Programs", *New Generation Computing*, 1986, pp. 5-33

[16] W. Hahn, A. Hagerer, H. Anger, "Update-Dataflow Computing: A Way to Supercomputing in Discrete Simulation of Digital Systems", *Proc. IFIP 12th World Computer Congress*, Madrid, Part 1, pp. 509-517, 1992

[17] N. Ito, "Data-flow Based Execution Mechanism of Parallel and Concurrent Prolog", *New Generation Computing*, No. 3, 1985, pp. 15-41

[18] P. Kascuk, "Distributed Data Driven Prolog Abstract Machine" *Implementations of Distributed Prolog*, P. Kascuk, M. J. Wise, Wiley Verlag, pp. 89-118, 1992

[19] C. Kujawski, *Der Modul Behaviour Decision* Universität der Bundeswehr, München, Fakultät für Informatik, 1993

[20] G. M. Papdopoulos, D. E. Culller, "Monsoon: an Explicit Token Store Dataflow Architecture", *Proc. of the 17th Int. Symp. on Computer Architecture*, Seattle, 1990

[21] E. Shapiro, "Concurrent Prolog: A progress report" *Fundamentals of Artificial Intelligence* W. Bibel and Ph. Jorrand, Springer Verlag, 1986, pp. 277-313

[22] L. Sterling, E. Shapiro, "The Art of Prolog, Advanced Programming Techniques", Massachusetts Institute of Technology, MIT Press, 1986

[23] P. C. Treleaven, R. P. Hopkins, P. W. Rautenbach, "Compining Data Flow and Control Flow Computing", *Computer Journal*, Vol. 25, No. 2, 1982, pp. 207-217

[24] S. Tsugawa, "Vision-Based Vehicles in Japan: The Machine Vision Systems and Driving Control Systems", *1. IEEE International Symposium on Industrial Electronics*, ISIE 93, Budapest, 1993, pp. 243-248

[25] S. Umeyama, K. Tamura, "A Parallel Execution Model of Logic Programs", *Proc. of the 10th Int. Symp. on Computer Architecture*, 1983, pp. 349-355

[26] A. H. Veen, "Dataflow Machine Architecture" ACM Computing Surveys, Vol. 18, No. 4, 1986, pp. 365-396

[27] M. J. Wise, "A Parallel Prolog: The Construction of a Data Driven Modell" *ACM Conference on LISP and Functional Programming*, 1982, pp. 56-67