Dataflow Computation for Knowledge-Based Control of Autonomous Systems* **

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

MIP 9612

Abstract

This paper presents a new data-driven model for parallel and efficient execution of logic programs. Applied to knowledge-based control of autonomous systems, it takes advantage of the fact that these systems use their set of rules in a special way.

A two-step method for constructing dataflow graphs is outlined. Its core part is the analysis of static data-dependencies, as they exist in the sets of rules for knowledge-based control.

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. Furthermore, control efficiency is increased by the model's purely data-driven evaluation scheme that saves any distribution of demands-for-evaluation.

* *A short version of this paper was published with the same title in the* **Conference Proceedings of AI, Simulation and Planning in High Autonomy Systems**, *San Diego, 1996, pp. 323-330.*

** *Another short version of this paper was published with the title "Knowledge-based Dataflow Computation for Autonomous System" in the* **Proceedings of the 6th International Symposium on Measurement and Control in Robotics**, *Brussels, 1996, pp. 531-536.*

Both papers are extended by the Prolog-example of a control system for an autonomous vehicle, including examples for the calculated output and the complete structure and dataflow graphs for this example.

1 Introduction

Knowledge-based control of highly autonomous systems should be organized hierarchically [1]: Lowest-level control must be fast and calculate exactly to guarantee rapid, reflex-like reactions when necessary. Therefore, these calculations mainly evaluate mathematical formulas. Higher control levels more and more coordinate fast reactions with planning according to a mission goal. In some cases, control is again based on mathematical formulas, but for complex control systems an expert system for planning and deciding must be employed. Such an expert system consists of a database and a set of rules evaluating the database according to the requested output [2].

A common language for implementing an expert system is Prolog [8], since it is well suited for expressing even a complex set of rules over an extensive database in an easy way. The disadvantage though lies on the performance side, due to Prolog's depth-first search algorithm with backtracking.

Various parallel Prolog languages, e.g., Concurrent Prolog [27] and PARLOG [9], have been worked out to overcome this disadvantage. Evaluation time decreases compared to sequential forms of Prolog, however, performance of parallel implementations is still not high enough to allow for control of autonomous systems in real-time. As one cannot do without specifying a knowledge-based control system in a high level language like Prolog, a new evaluation model for logic programming is needed.

The paper starts with an overview of existing dataflow models for logic programming. Next, the base concepts of logic programming and dataflow architectures will be explained briefly, followed by examining the different conditions for knowledge-based control in comparision to general-purpose knowledge-based systems to illustrate how a new dataflow model, resulting in a more efficient type of evaluation, is possible.

The paper continues with a detailed description how to construct dataflow graphs due to the new model. 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. Based on this information the dataflow graph can easily be developed in the next step. Finally, an overview of achievable parallelism will be given.

2 State of the Art

Logic programming [29], a mathematical formalism for certain classes of problems that require deductive reasoning [13], is the base concept of Prolog, a high-level, non-procedural programming language. Furthermore, the language was not developed for a special computer architecture, especially not with the von Neumann architecture in mind.

Prolog contains highly parallel structures (OR- and AND-parallelism [27]), at the level of simple predicates as well as at the level of rules. 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 [32, 30]. To exploit the performance potential of dataflow architectures for knowledge-based systems, the question is how Prolog can be compiled into a graph for using Prolog as a high-level language for dataflow computers.

L. Bic [6, 7] presented an unconventional method that is suited best for a special group of Prolog-programs, i.e., those over a relational database. Concerning control systems, however, databases are not relational, but usually consist of unlinked facts, e.g. sensor data.

Another concept for a data-driven evaluation model of Prolog, without any implementation details worked out, can be found in [33]. By extending Prolog to a new language, Epilog, M. J. Wise came to an implementation on general-purpose multiprocessor-systems [34].

A more conventional model was developed by P. Kaszuk [21, 22, 23]. He presented a completely worked-out concept for translating Prolog-programs into dataflow graphs for a special architecture, called Homogenous Processor Space, but, since the structure of a Prolog-program is mapped closely onto a graph structure without changing Prolog's evaluation mechanism, the model finally comes out with demand-driven computation instead of data-driven computation.

By another approach Z. Halim [17, 18] made a step towards data-driven evaluation. His more complicated graph allows for unidirectional flow of data. However, the drawback is, that the graphs are derived directly from the syntactical structure of Prolog and thus still lack from efficiency. Similar, even more complex dataflow models for Prolog were worked out by S. Umeyama and K. Tamura [31], and also by N. Ito and H. Shimizu [20].

However, all this work is related to general-purpose Prolog-programs, i.e., to formulate rules in Prolog for answering a variety of queries over a mainly static set of facts. 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

Details of logic programming in Prolog can be found in [29]. In this paragraph only a short overview is intended.

Fundamentals of Prolog

A *logic program* is a set of clauses (or rules) of a certain logic, the Horn clause logic. A successful computation corresponds to the proof of a statement (called goal statement) from these clauses.

Definition: *Logic program*
A logic program $\mathcal{L}$ is a tuple (C, L, P, T) where C is a set of clauses, L a set of literals, P a set of predicate symbols, and T a set of terms.

i) Each *clause* $c_i \in C$, $i \geq 1$ is of the form
$p_i^0 \leftarrow p_i^1, ..., p_i^n$. .

ii) Each $p_i^j \in P, j = 0, ..., n$ is called a literal and is of the form $\nu(t_1, ..., t_m)$,
- where $\nu \in P$ is a predicate symbol and
- $t_1, ..., t_m \in T, m \geq 1$ are terms, which may be constants, variables, or again terms.

p^0 is called the *head* of the rule, the $p^1, ..., p^n$ form the *body* of the rule.

A rule with an empty body represents a *fact*
p^0. ,
while a rule without a head is called a *goal* (or a set of goals):
$\leftarrow p^1, ..., p^n$. .

A goal (or a set of goals) is a starting point for the execution of a program.

Beyond this concept of pure logic programming, Prolog has some extensons, that will be explained briefly.

- Literals may specify predefined functions. They are called metapredicates.
- 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.

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

- 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.

The "," in the body of a rule forms a conjunctive combination of the literals in the body of the rule.

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 represent 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**.**.

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.

Evaluation Mechanism of Prolog

The core of interpreting a Prolog-program is the Unification Algorithm, where a goal is unified with the head of a rule, or with a fact. If the goal is unified with the head of a rule, the rule is applied.

Binding a variable to a value or a set of values is mostly done by this unification, not by assignments or by applying a metapredicate, though both alternatives are possible. If a value (or a set of values) for a variable is calculated by a node and needed by any other one, the value is passed from the calculating node to the latter one.

An algorithm for unification as the main evaluation concept of Prolog in Fig. 1 and an abstract interpreter for Prolog-programs in Fig. 2 are given here for completeness. Both can also be found in [29].

Algorithm: *Unification Algorithm for Logic Programs*
Input: t_1, t_2.

Output: $(\Phi = mgu\ (t_1, t_2))$ or *failure*.

Algorithm: $\Phi = \{\}$;
$Stack := \{t_1 = t_2\}$;
$failure := FALSE$;
while $(Stack \neq \{\})$ and not $failure$ do
  for $\{X = Y\} \in Stack$ do
    $Stack := Stack \backslash \{X = Y\}$;
  if $(var(X)$ and $var(Y))$ then ;
  else if $(var(X)$ and $(X \notin Y))$ then
    forall $X \in (\Phi$ and $Stack)$ do $[Y/X]$;
    $\Phi := \Phi \cup \{X = Y\}$;
  else if $(var(Y)$ and $(Y \notin X))$ then
    forall $Y \in (\Phi$ and $Stack)$ do $[X/Y]$;
    $\Phi := \Phi \cup \{Y = X\}$;
  else if $((X = f(X_1, ..., X_n)$ and $(Y = Y_1, ..., Y_n))$,
      $func(f)$ and $n > 0$ then
    $\Phi := \Phi \cup \{X_n = Y_n\} \forall n = 1, ..., n$;
  else $failure$ = TRUE;

Figure 1: Unification Algorithm for Logic Programs

Algorithm: *Abstract Interpreter for Logic Programs*
Input: $\mathcal{G} \in Goals$,
$\mathcal{P} \in Programs$.
Output: An instance of $\mathcal{G}$ that is a logical consequence of $\mathcal{P}$,
or *failure*.
Algorithm: $\mathcal{R} := \mathcal{G}$.
while $(\mathcal{R} \neq \{\})$ do
  for $p_0 \in \mathcal{R}$ do
    if $p'_0 \leftarrow p_1, ..., p_n \in \mathcal{P}, p_0 = \Phi \circ p'_0$ then
      $\mathcal{R} = \mathcal{R} \backslash p_0$;
      $\mathcal{R} = \mathcal{R} \cup p_1, ..., p_n$;
      $\mathcal{R} = \Phi \circ \mathcal{R}$;
      $\mathcal{G} = \Phi \circ \mathcal{G}$;
    else $failure$ = TRUE;
if $(\mathcal{R} = \{\})$ then return $\mathcal{G}$;
else return $failure$;

Figure 2: Abstract Interpreter for Logic Programs

Solving the query $\mathcal{G}$ with respect to a program $\mathcal{P}$ can be done by the abstract interpreter as given above.

A *computation* of a goal $\mathcal{Q} = \mathcal{Q}_0$ by a program $\mathcal{P}$ is a (possible infinite) sequence of triples $(\mathcal{Q}_i, \mathcal{G}_i, \mathcal{C}_i)$. $\mathcal{Q}_i$ is a (conjunctive) goal, $\mathcal{G}_i$ is a goal occuring in $\mathcal{Q}_i$ and $\mathcal{C}_i$ is a clause $p_0 \leftarrow p_1, ..., p_n$ in $\mathcal{P}$ renamed so that it contains new variable symbols not occuring in $\mathcal{Q}_j$, $0 \leq j \leq i$.

For all $i > 0$ $\mathcal{Q}_{i+1}$ is the result of replacing $\mathcal{G}_i$ by the body of $\mathcal{C}_i$ in $\mathcal{Q}_i$, and applying the substitution Φ_i, the most general unifier of $\mathcal{G}_i$ and p_0, the head of $\mathcal{C}_i$; or the constant `SUCCESS` if $\mathcal{G}_i$ is the only goal in $\mathcal{Q}_i$ and the body of $\mathcal{C}_i$ is empty; or the constant `FAIL` if $\mathcal{G}_i$ and the head of $\mathcal{C}_i$ do not unify.

Parallel Forms of Prolog

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 parallized and, if more than one goal is found, the goals are executed in parallel.

2a *Full AND-parallelism*
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 other 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 extention 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.

In the rest of the paper it will be shown that the same amount of parallelism can be obtained as by OR parallelism combined with restricted AND-parallelism.

4 Dataflow Architecture Principles

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 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 are present on the input arcs. 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.

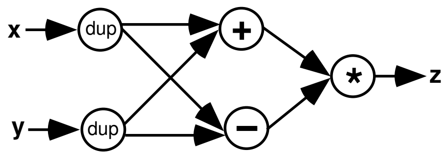


Figure 3: Example of a dataflow graph

Since processing of nodes is driven by the availability of the required data, no centralized control is necessary. This fundamental principle permits nodes to be evaluated in parallel, without explicitly specifiying the parallelism. Fig. 3 shows a simple example, where first the two independent dublicate operations ("dup") and later on the "+" and the "−" operation can be executed in parallel.

Underlying Machine Architecture

Dataflow architectures and their implementations have a long-standing research history. J.Dennis [10, 11] led a project at the Massachusetts Institute of Technolgy that resulted in the MIT Static Dataflow Architecture. Also at the MIT, Arvind and Gostelow [3, 4] developed the MIT Tagged-Token Dataflow Architecture. Later on, this project was continued as MONSOON-project [25]. At the University of Manchester, the Manchester Dataflow Computer was built [14]. In Japan, the SIGMA-1 was designed at the Electrical Laboratory in Tsukuba [28, 19]. At our laboratory, the design of the Munich Simulation Computer (MuSiC), an eventflow computer for high performance simulation of digital systems, was finished [15, 16]. On the market, the ideas behind these systems led to implementations, mainly as Digital Signal Processors (DSPs). A more detailed overview is given in [26].

5 Dataflow Graphs for Knowledge-based Control

Control of Autonomous Systems

A Prolog-program for knowledge-based control differs in three aspects from a general purpose Prolog-program:

- *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. In the simple case of an autonomous mobile only the queries "faster?", "slower?", "more left?" and "more right?" have to be answered. Moreover, these queries do not change during run-time. They are known at specification time and can be compiled into the program.
- *Dynamic database*
  The database contains information about the autonomous system, the mission goal, the environment and the actual situation. This means, that at least some data items of the database change with high frequency, e.g., the frequency with which the environment is sensored. In contrast the database of a general-purpose Prolog-program is mostly static. Changes during run-time can even cause enormous side-effects, e.g., incomplete results or contradictions.
- *Frequently repeated evaluation*
  Due to real-time conditions, a 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.

These three differences affect the evaluation method: Since the queries are static, they can be compiled into the program. Thus, they no longer have to cause demands-for-evaluation flowing along the graph from the queries to the facts. Rather calculation 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 a calculation has started, only affect the next calculation.

The Structure Graph

Translating a Prolog-program into a dataflow graph for evaluation in a purely data-driven way requires two steps. In the first one, a structure graph is constructed, that is transformed into a dataflow graph in the second one.

More detailed, at first a simple graph is created where every predicate is represented by a node. 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. Thus, each path of the graph is a sequence of AND-OR-AND- and AND-AND-structures, respectively, in any arbitrary combination.

All other elements of a Prolog-program, i.e., facts, assignments, comparisions, 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 comparisions (e.g. $X \leq 0$), are represented by macronodes. Behind these again a dataflow graph structure stands for evaluating the expression.

All nodes are connected by arcs according to the structure of the original program. The result is a tree-like graph that may contain cycles. Since this graph just reflects the structure of the original Prolog-program it is simply called structure graph.

Definition: *Structure Graph*

A structure graph SG for a Prolog-program $\mathcal{P}$ is given by $SG = (\mathcal{P}, V, E_d, F)$, with

i) V is a set of nodes, with

- a node v_q as a starting point for the query.
- a node $v_i^0, i \geq 1$ for every head of a rule p_i^0 and every fact p_i^0.
- a node $v_i^r, i, r \geq 1$ for every literal p_i^r in the body of a fact that is a metapredicate, or an arithmetic or logic operation, or a cut.
- one node $v_j^+, j \geq 1$ for every set of heads of rules or set of facts with the same predicate and number of parameters.

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

- from v_q and every node $v_i^0, i \geq 1$ to every node $v_j^0, j \geq 1$, if there is a literal $p_i^r, r \geq 1$ in the body of the rule with head p_i^0 or in the query that is of same name and type as the literal p_j^0 represented by v_j^0.
- from v_q to every node in the query and from every node $v_i^0, i \geq 1$ to every node in the body of the rule with head p_i^0 that is a metapredicate, or an operation, or a cut.
- from every node $v_i^+, i \geq 1$ to every node of the set of alternatives v_i^0 represented by v_i^+.

$e_{v,u} \in E_d$ denotes edge going from v to u.

iii) $F = \{$AND, OR, FACT, CALC, FUNC, CUT, FAIL, SUCC, QUERY$\}$ is a set of attributes assigned to the nodes. $v.attr$ denotes the attribute assigned to node v.

- $v_i^r.attr =$

$$\begin{cases} \text{AND} & \text{if } p_i^r \text{ is head of a clause }, r = 0 \\ \text{FACT} & \text{if } p_i^r \text{ is a fact }, r = 0 \\ \text{SUCC} & \text{if } p_i^r = \texttt{succ}, r \geq 1 \\ \text{FAIL} & \text{if } p_i^r = \texttt{fail}, r \geq 1 \\ \text{CUT} & \text{if } p_i^r = \texttt{cut}, r \geq 1 \\ \text{FUNC} & \text{if } p_i^r \text{ is a metapredicate }, r \geq 1 \\ \text{CALC} & \text{if } p_i^r \text{ is an operation }, r \geq 1 \end{cases}$$

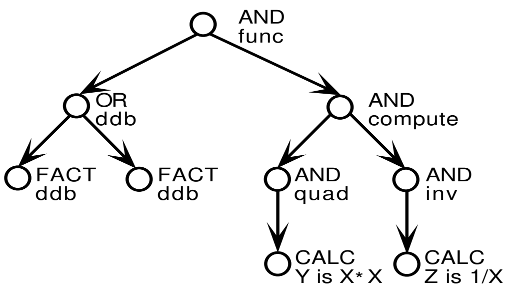
- $v_i^+.attr =$ OR

- $v_q.attr =$ QUERY

An example is given in Fig. 4. Facts are represented by FACT-nodes, macronodes by CALC-nodes.

$$\begin{array}{ll} func(X,Y,Z) & \leftarrow ddb(X), compute(X,Y,Z). \\ compute(X,Y,Z) & \leftarrow quad(X,Y), inv(X,Z). \\ quad(X,Y) & \leftarrow Y \ is \ X * X. \\ inv(X,Z) & \leftarrow Z \ is \ 1/X. \\ ddb(2). & \\ ddb(4). & \end{array}$$

a) Example-program



b) Corresponding structure graph

Figure 4: Prolog-program and its structure graph

To the definition given above some extentions have to be made:

- In the head of a rule there may be assignments (e.g. $p(X-1,Y)$) or even list constructions (e.g. $p([X|Xs], X, Xs)$). Normally these are evaluated by an extended unification algorithm. 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 macronode evaluating the expression is inserted as child-node.

  For example, if and only if X is an input variable and Y an output variable,
  $p(X-1, Y*2) \leftarrow q(X,Y)$.

is translated into

$p(X^1, Y^1) \leftarrow X \ \ is \ X^1 - 1, q(X, Y), Y^1 \ \ is \ Y * 2.$.

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

- Negation by failure is included by a special graph structure, i.e., the general implementation of the metapredicate **not** is, while X has either to be a constant or a variable instanteated with a value:

  $not(g(X)) \leftarrow g(X), !, fail.$

  $not(g(X)).$

  and the corresponding structure graph is as depicted in Fig. 5. FAIL represents the metapredicate fail, CUT the cut and SUCC is added to let the computation path be successful without any further calculation.

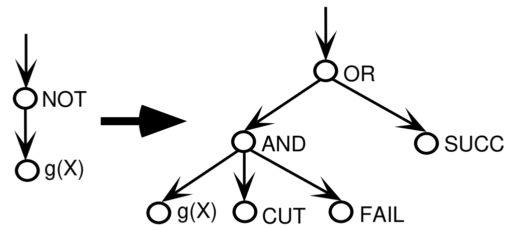


Figure 5: Structure Graph for Negation as Failure

Analysis of Data Dependencies

The second step of the new approach is analyzing data-dependencies between the literals and changing arcs according to these dependencies. In this context analyzing means determining which parameters of a rule or a predicate are input-variables and which ones are output-variables. Since Prolog is a single assignment-language, this analysis can be done by simple rules while the program is compiled:

Definition: Input- and Output-Type of Parameters of a Node

$t_i(v).type$ denotes type input or output for term $t_i \in T, i \geq 1$ of the literal assigned to node v.

i) $v.attr$ = FACT:

- $t_i(v).type$ = output, if $t_i(v)$ is a constant.

- $t_i(v).type$ = input, if $t_i(v)$ is a variable.

ii) $v.attr$ = FUNC:

the type of the parameter $t_i(v)$ is given by the declaration part of the meta-predicate.

iii) $v.attr$ = CALC:

- if the operation is a comparison, then $t_i(v).type$ = output

- if the operation is an assignment, then

  · $t_i(v).type$ = output, if $t_i(v)$ appears left of the assignment symbol.

  · $t_i(v).type$ = input, otherwise.

iv) $v.attr$ = AND:

if $e_{v,u} \in E_d$ with $u \in V$ exists and due to the left-to-right-appearence of edges in the structure graph, u is the first node that uses t, the type of t in v is the same type as of t in u.

Of an **AND-node** the type of a parameter is determined by that child-node out of all child-nodes that uses the parameter first, due to the left-to-right-appearence in the structure graph. If this first appearence enforces the type output, the parameter may be of type input or output at the other child-nodes. On the other hand, if the type input is enforced, the parameter has to be of type input in every other child-node that uses the parameter. Otherwise an error occurs, since the value is assigned twice; first, when defining the input value for the previous child-node, second by evaluating the latter child-node.

v) $v.attr$ = OR:

The type of the parameters of an OR-node results from the type of the parameters of its AND-child-nodes. If $e_{v,u} \in E_d$ with $u \in V$ exists, the type of the parameter t in v is the same type as the type of t in u, since every OR-node has the same set of parameters as the AND-nodes which the OR-node connects.

The type of the parameters of an **OR-node** also results from the type of the parameters of its AND-child-nodes. Parameters of all AND-child-nodes of an OR-node have to be of the same type. Otherwise again an error occurs, due to the violation of the single-assignment condition.

Terms with arithmetic operations and list constructions are not considered, as well as constants in any other node than a fact, since these constructions are extracted from the list of parameters, as mentioned above. Thus, parameters can only be variables and, at a fact, variables or constants.

The further analysis of data-dependencies 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.

The following examples are parts of a control system for an autonomous road vehicle. They implement a change of the actual lane to the left lane, to overtake a slower vehicle.

Example 1 a:

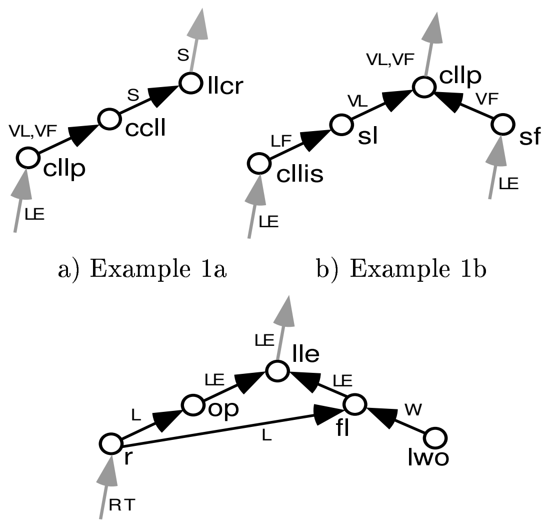
leftlane_change_request(LaneExists^i*,Steering*^{o}*)* ←
  change_leftlane_possible(LaneExists^i*, Velocityleft*^{o}*,Velocityfront*^{o}*),*
  calculate_change_leftlane(Velocityleft^i*, Velocityfront*^{i}*,Steering*^{o}*).*

Example 1 b:

change_leftlane_possible(LaneExists^i*,Velocityleft*^{o}*, Velocityfront*^{o}*)* ←
  change_leftlane_is_secure(LaneExists^i*,LaneFree*^{o}*),*
  slow_left(LaneFree^i*,Velocityleft*^{o}*),*
  slow_front(LaneExists^i*,Velocityfront*^{o}*).*

Example 1 c:

leftlane_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}*).*



a) Example 1a b) Example 1b

c) Example 1c

Figure 6: Data-dependencies of example 1a - c

In the rules the "^{i}" denotes a variable of type input, the "^{o}" a variable of type output.

The graphs in Fig 6 depict the data-dependencies according to the 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 overall input and overall output of variables.

The principle behind overall inputs is, not to pass a value from the top of the graph down to the nodes where it is needed, but to have the value sent to these nodes directly.

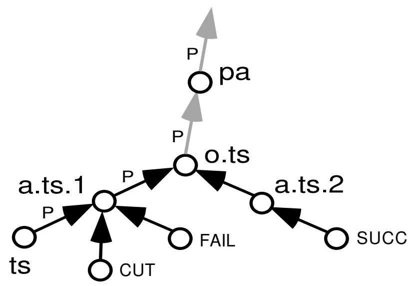


Figure 7: Example for negation as failure

Example 2 contains negation as failure:
  passing_allowed(P^o) $\leftarrow$
    not(trafficsign($pass^o$, P^o)).

The graph structure is depicted in Fig. 7, derived from the implementation of *not* in the structure graph as discussed above.

The analysis of data-dependencies, considers the whole structure graph in one pass. Therefore, the next examples discuss how to connect graph fragments like those in Fig. 6a) to Fig. 6c).

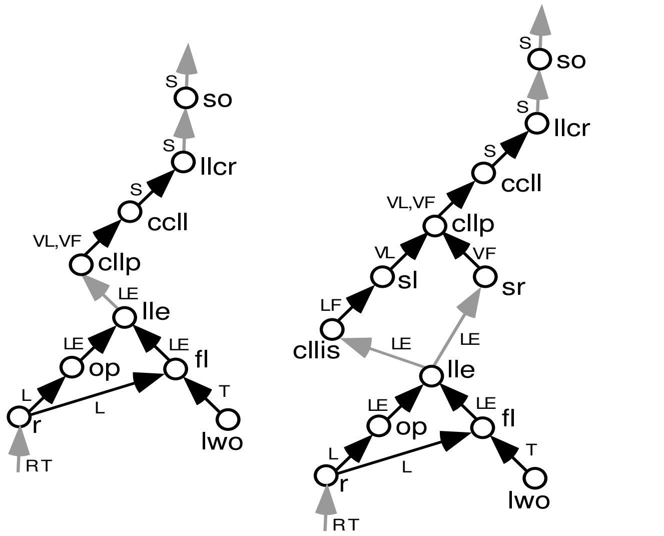
The rules for `llcr` and `lle` from Fig. 6a) and Fig. 6c) may occur, for instance, together as in example 3:
  start_overtake($Roadtype^i$,$Steering^o$) $\leftarrow$
    leftlane_exists($Roadtype^i$, $LaneExists^o$),
    leftlane_change_request($LaneExists^i$,$Steering^o$). .

The resulting graph is depicted in Fig 8a).

Fig 8b) shows the result of the next analyzing step in which `cllp` is analyzed inside of `llcr`. The outcome is that the node `cllp` becomes replaced by the graph fragment of Fig 6b).

The rule `lle` produces a variable X, which is used as input for the rule `llcr`, in `llcr` the variable X is input for `cllp` and in `cllp` it is input for `r` and `lwo`. Therefore `lle` has to send the value of X directly to `r` and `lwo`.

If the rule of example 3 is extended to the following rule, denoted in example 4:
  start_overtake($Roadtype^i$,$Steering^o$) $\leftarrow$
    leftlane_exists($Roadtype^i$, $LaneExists^o$),



a) Combining `llcr` and `lle` b) Inserting a graph for `cllp`

Figure 8: Connecting graph fragments

leftlane_detected(LaneExists^o),
passing_allowed(1^o),
leftlane_change_request(LaneExists^i,Steering^o).

`lle` as well as `lld` calculate values for the variable X used in `llcr`.

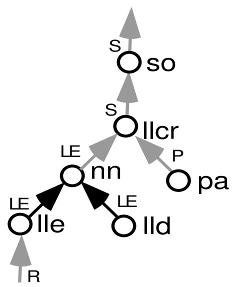


Figure 9: Adding a new AND-node `nn`

Thus, as depicted in Fig. 9, a new AND-node, named `nn`, has to be inserted, to form an intersection of the calculated values from `lle` and `lld` for passing only correct values to `llcr`.

In case of a cut, sequentiality is conserved; in the rule
$p_0 \leftarrow p_1, ..., p_k, !, p_{k+1}, ..., p_n.$

the literals $p_1, ..., p_k$ may be executed in parallel, then the cut is executed, and then the literals $p_{k+1}, ..., p_n$ may also be executed in parallel.

The only way to realize loops in Prolog is recursion. Recursion, though, needs a special, more complexly structured dataflow graph, but can be realized in an analog way.

A simple example for recursion, calculating the powers of 2, is:

$f(X,Y) \leftarrow X = 1, Y \ is\ 2.$

$f(X,Y) \leftarrow X > 1, V \ is\ X - 1, f(V,Z)$,Y $is\ Z * 2.$.

The second line contains the recursion call, while the first line terminates the recursion. The operational semantics of the two lines is: If X is equal to 1, the recursion stops with the value "2" as output for Y, otherwise, X is decreased and $f(X,Y)$ is called again. After succesfully executing the termination rule, Y, of value "2", is multiplied $(X-1)$ times with the value "2". The result is 2^X.

Fig. 10 depicts the graph structure for a simple recursion like the example above.

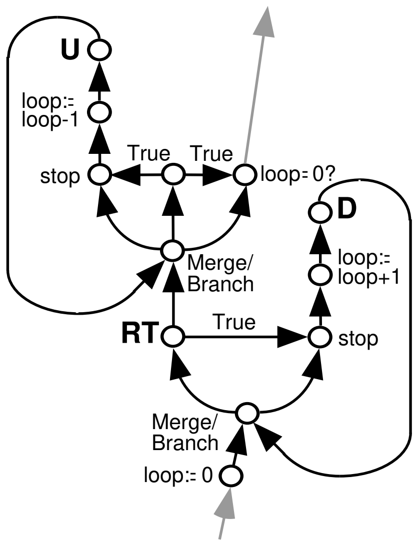


Figure 10: Recursion sheme

All rules that terminate the recursion are referred to as `RT`. In the rule with the recursion call, the `D` denotes the literals standing before the recursion call and the recursion call itself, while the `U` denotes the ones standing behind the recursion call. The literals in `D` can be executed in parallel, also the literals in `RT`, and the literals in `U`. But the blocks `U, RT` and `D` may be executed in sequence.

If there are no literals before the recursion call in the recursive rule, `D` consists solely of the recursion call itself. If there are no literals after the recursion call, `U` is empty.

This model enforces sequentiality. Any rule containing the recursion call is executed in sequence to the rules containing the termination. Additionally the literals appearing before the recursion call, the recursion call itself und the literals appearing behind it are executed in sequence. Thus, recursion stops immediately, if the rule, containing the termination, succeeds, or if any literal in the recursive rule appearing before the recursion call fails. Executing the recursion fully in parallel would require a far more complex graph structure to terminate the recursion.

More complex forms of recursion like cascadic recursion or nested recursion can be modelled by extending this graph-sheme.

Other dataflow graphs for recursion have been investigated by [5], and have therefore shown that recursion is possible in data flow graphs.

The Dataflow Graph

The result of analyzing the data-dependencies of a structure graph is a dataflow graph. This is now a graph, where the direction of the arcs goes from the facts, kept in a dynamic database, up to the nodes that represent the other predicates, in respect to the sequence in which calculation must take place.

Definition: *Dataflow Graph*

A dataflow graph DG for a Prolog-program $\mathcal{P}$ and a structure graph SG is given by $DG = (\mathcal{P},\ SG,\ V',\ E_{DF}, F, \Phi)$, with

i) $E_{DF} \subset V' \times V'$ is a set of directed edges.

- If $v \in V$ with $v.attr$ = AND:
  - If in the structure graph exist $u, u' \in V$ with $e_{v,u}, e_{v,u'} \in E_d$ and $t_i(u), t_j(u') \in T, i, j \geq 1$ are parameters of u and v with $t_i(u).attr = output, t_i(u').attr = input$ and $t_i(u) = t_j(u')$, then $e'_{u,u'} \in E_{DF}$ is created.
  - If in the structure graph exists $u \in V$ with $e_{v,u} \in E_d$ and u and v have parameters $t_i(u), t_j(v) \in T, i, j \geq 1$ with $t_i(u).attr = t_j(v).attr = output$ and $t_i(u) = t_j(v)$, then $e'_{u,v} \in E_{DF}$ is created.
- if $v \in V$ with $v.attr$ = FACT or $v.attr$ = FUNC or $v.attr$ = CALC and v has parameter $t_i(v) \in T, i \geq 1$ with $t_i(v).attr = input$ and $\exists w \in V$ with $w.attr$ = OR or ($\exists w \in V$ with $w.attr$ = AND and $\nexists w' \in V$ with $w'.attr$ = OR and $e_{w,w'} \in E$) and w has parameter $t_j(w) \in T, j \geq 1$ with $t_j(w).attr = output$, then $e'_{w,v} \in E_{DF}$ is created.
- If $v \in V$ with $v.attr$ = OR and exists $u \in V$ and $e_{v,u} \in E_d$ $e'_{u,v} \in E_{DF}$ is created.

- If $v \in V$ with $v.attr$ = QUERY and exists $u \in V$ and $e_{v,u} \in E_d$ $e'_{u,v} \in E_{DF}$ is created.

ii) V' is a set of nodes, $V' = V \cup V_\wedge$ and $V_\wedge$ is the set of new AND-nodes:

if $existsv, v', w \in V$ with $v.attr = v'.attr \in$ {AND, OR} and $w.attr \in$ {FACT, FUNC, CALC} and $\exists e_{v,w}, e_{v',w} \in E_{DF}$ then $w' \in V_\wedge$ is created with $w'.attr$ = AND and $e_{v,w'}, e_{v',w'}, e_{w',w} \in E_{DF}$ are created, replacing $e_{v,w}, e_{v',w} \in E_{DF}$

iii) Φ is a set of functions assigned to the nodes.
$\Phi = \{f_v \mid v \in V, f_v : \underbrace{T \times ... \times T}_{in(v)\ arguments} \rightarrow \underbrace{T \times ... \times T}_{out(v)\ results}$
with $in(v) := |\{w \mid (w,v) \in E, w \in V\}|$ denoting the number of ingoing arcs, and $out(v) := |\{w \mid (v,w) \in E, w \in V\}|$ denoting the number of outgoing arcs.

Tokens that are sent from node to node can be classified: A successful computation, i.e., an arithmetic or logic operation, results in a *success*-token carrying calculated values. A failed computation results in a *failure*-token. The cut sends a *cut*-token.

Definition: *Functionality of the Nodes of a Dataflow Graph*

Φ is defined as:

- if $v.attr$ = SUCC, the output token is a *success*-token without any values.
- if $v.attr$ = FAIL, the output is a *failure*-token
- if $v.attr$ = CUT, the output is a *cut*-token.
- if $v.attr$ = FACT, the output is a *success*-token, if the values of any incoming token match the values of the fact, or if there is no incoming token needed. Otherwise the output is a *failure*-token.
- if $v.attr$ = CALC or FUNC, the output is a *success*-token if the computation of the node on an incoming token was executed successfully.
- if $v.attr$ = AND, all incoming tokens are collected, while contradicting values for a variable are deleted. Then a *success*-token is sent as output, if at least one valid value exists. Otherwise a *failure*-token is sent.
- if $v.attr$ = OR, all incoming tokens are collected, all *success*-tokens are sent forward, and if there is no *success*-token, one *failure*-token is sent.
- if $v.attr$ = QUERY, there is no function to be applied. The tokens arriving at this node are the graph's output.

As an example, Fig. 11 depicts the dataflow graph corresponding to the structure graph in Fig. 4.

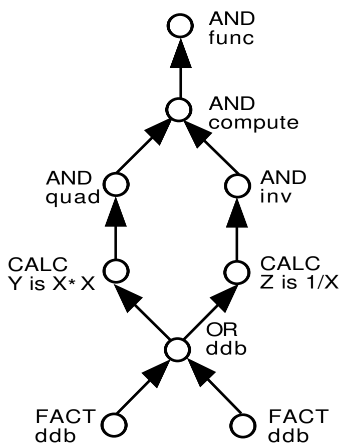


Figure 11: Dataflow graph

A comparision of the dataflow graph with the structure graph shows that most arcs only change their direction from top-down to bottom-up. In addition, however, some arcs are deleted and new arcs are inserted. This modification fullfills the condition, that on the one hand, if a variable is of type input, the value must be known before calculation in the literal takes place. On the other hand, if a variable is of type output, its value is the result of calculating the literal.

Result

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

Where the analysis detects data-independent literals in the body of a rule, these are marked for execution in parallel. This leads to the same amout of parallelism as restricted AND-parallelism. Additionally, the child-AND-nodes of an OR-node independently calculate values for the same variables. These values are collected and grouped together by the OR-node. Thus, full OR-parallelism is exploited.

In parallel Prolog-programs the cut is a construct to enforce sequentiality at the position where it appears. Recursion also requires sequentiality to avoid non-terminating recursions. Both forms of sequentiality are preserved by this model.

The general advantage of this new dataflow model, however, 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 are passed the whole way back. Now, activity stimulated by changes of facts goes only through the graph bottom-up to the queries. Further, even a new calculation can be started, before the previous one has finished. This guarantees higher calculation frequency and thus supports control in real-time.

Conclusion

A new concept suited for knowledge-based control of autonomous systems is presented. At the moment, control systems for autonomous mobiles are the primary concern of this work.

A test example is "VaMoRs", an autonomous mobile, developed by Prof. Dr. Dickmanns, University of the Federal Armed Forces, Munich [12]. Of special interest is the module "Behavior Decision" [24] that implements the set of rules for the "intelligent" behavior of this vehicle by von Neumann-computing. The goal of this work is to reimplement this module to prove the suitability of dataflow computation for knowledge-based control of autonomous systems by real world data.

Appendix A

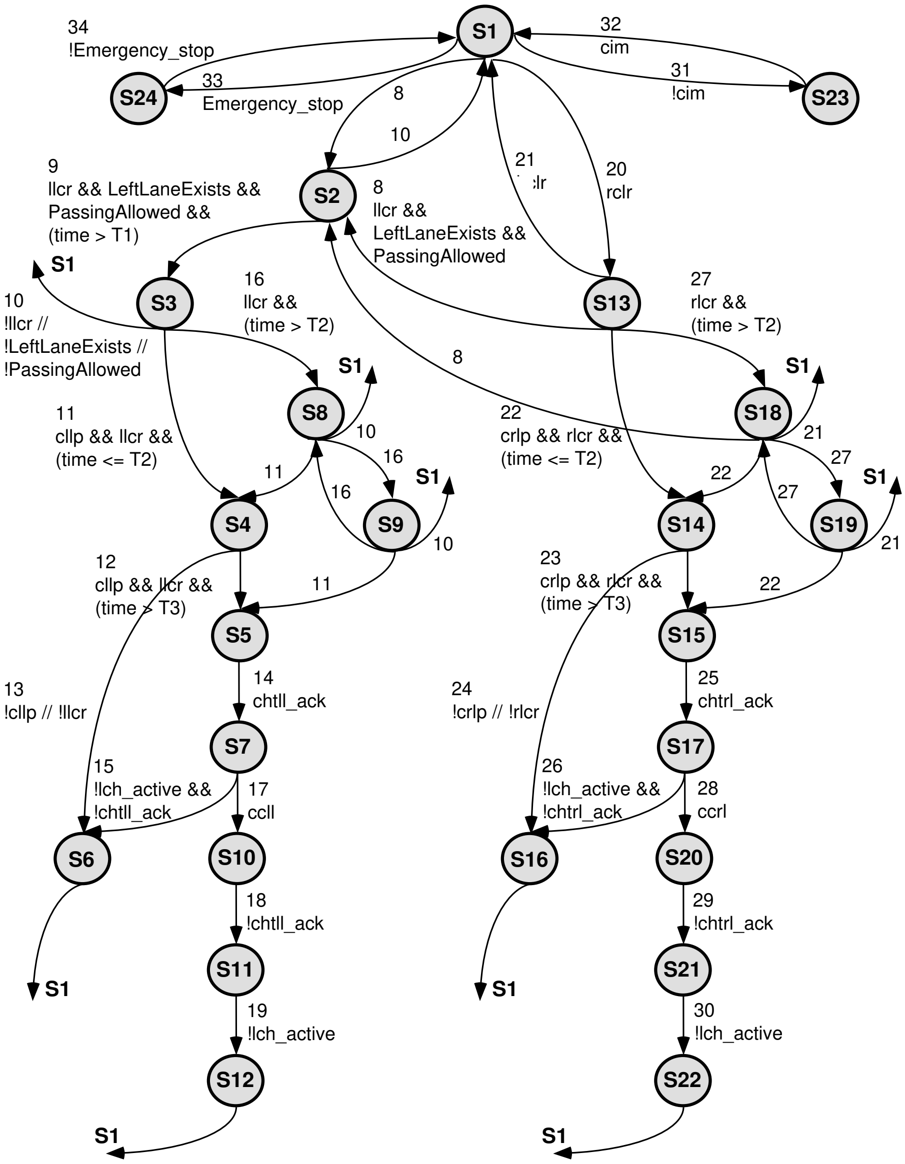


Figure 12: State transitions

Fig. 12 shows the state transistions of the module “Behavior Decision” [24].

Appendix B

This is the module "Behaviour Desicion" implemented as a rule system in Prolog.

```
%--------------------------------------------------------------
%
%  Input:
%    vplanned : Wished velocity
%    vallowed : Maximal allowed velocity
%    vcarmax  : Maximal possible velocity of the car
%    endpoint : Distance to a stop point
%    cdist    : Distance to a curve
%    cangle   : Angle and left/right orientation of the curve
%    clength  : Lenght of the curve.
%    gradient : Steepness of the road
%
%  Ouput:
%    vactual  : actual velocity
%    delta    : diffence of actual velocity to the
%               velocity of the last calculation
%    gas      : accelerator output
%    brake    : brake output
%    curve    : wheel output
%
%  Constants:
%     TRUE   = 1     FALSE  = 0%    OWN    = 0     NOTDEF = -1
%     NONE   = 0     RIGHT  = 1     LEFT  = 2      BOTH  = 3
%
%--------------------------------------------------------------

time_less(Tau) :-
   clock(CL),
   time(TL),
   TL + Tau > CL.

%--------------------------------------------------------------

%  car is moving
cim(1) :-
   vc_vms(VMS),
   vmin(VMIN),
   VMS >= VMIN.
cim(0).

%  lelt lane change request
```

```
llcr(1) :-
   vepsilon(Vepsilon),
   vc_vms(VMS),
   vleft(Vleft), vfront(Vfront),
   dm_vpl(Vplanned),
   ddb_lle(1),
   vc_convoy(1),
   Vplanned > VMS + Vepsilon,
   Vleft > Vfront + Vepsilon.
llcr(0).

%  right lane change request
rlcr(1) :-
   ddb_rle(1).
rlcr(0).

%  change to left lane is possible
cllp(1) :-
   bl(Wert),
   dm_passall(1),
   ddb_lle(1),
   Wert >= 1,
   dm_cllp(1).
cllp(0).

%  change to right lane is possible
crlp(1) :-
   br(Wert),
   ddb_rle(1),
   Wert >= 1,
   dm_crlp(1).
crlp(0).

%  cancel change to left lane
ccll(1) :-
   cancel(Abbruch),
   bl(Wertl), bo(Werto),
   vc_chtlack(2),
   ddb_cwrl(0),
   Wertl < Abbruch, Werto >= 1,
   dm_ccll(1).
ccll(1) :-
   cancel(Abbruch),
   bo(Werto), br(Wertr),
   vc_chtlack(2),
```

```
   ddb_cwrl(0),
   Werto < Abbruch, Wertr >= 1,
   dm_ccll(1).
ccll(0).

%  cancel change to right lane
ccrl(1) :-
   cancel(Abbruch),
   br(Wertr), bo(Werto),
   vc_chtlack(1),
   ddb_cwrl(0),
   Wertr < Abbruch, Werto >= 1,
   dm_ccrl(1).
ccrl(1) :-
   cancel(Abbruch),
   bo(Werto), bl(Wertl),
   vc_chtlack(1),
   ddb_cwrl(1),
   Werto < Abbruch, Wertl >= 1,
   dm_ccrl(1).
ccrl(0).

%-----------------------------------------------------------------

% Calculating the actions in each state

calc_statetransition(State, 24) :-
   State =\= 23,
   vc_emstop(1).

calc_statetransition(State, 23) :-
   State =\= 24,
   vc_emstop(0),
   not(cim(1)).

calc_statetransition(1, 13) :-
   rlcr(1).
calc_statetransition(1, 2) :-
   llcr(1),
   ddb_lle(1),
   dm_passall(1).

calc_statetransition(2, 1) :-
   not(llcr(1)).
calc_statetransition(2, 1) :-
```

```
   ddb_lle(0).
calc_statetransition(2, 1) :-
   dm_passall(0).
calc_statetransition(2, 3) :-
   clt1(T1),
   llcr(1),
   ddb_lle(1),
   dm_passall(1),
   not(time_less(T1)).

calc_statetransition(3, 1) :-
   not(llcr(1)).
calc_statetransition(3, 1) :-
   ddb_lle(0).
calc_statetransition(3, 1) :-
   dm_passall(0).
calc_statetransition(3, 4) :-
   clt2(T2),
   cllp(1),
   llcr(1),
   time_less(T2).
calc_statetransition(3, 8) :-
   clt2(T2),
   llcr(1),
   not(time_less(T2)).

calc_statetransition(4, 5) :-
   clt3(T3),
   cllp(1),
   llcr(1),
   not(time_less(T3)).
calc_statetransition(4, 6) :-
   not(cllp(1)).
calc_statetransition(4, 6) :-
   not(llcr(1)).

calc_statetransition(5, 7) :-
   vc_chtlack(2).

calc_statetransition(6, 1).

calc_statetransition(7, 6) :-
    vc_lchact(0),
    vc_chtlack(0).
```

```
calc_statetransition(7, 10) :-
    ccll(1).

calc_statetransition(8, 1) :-
   not(llcr(1)).
calc_statetransition(8, 1) :-
   ddb_lle(0).
calc_statetransition(8, 1) :-
   dm_passall(0).
calc_statetransition(8, 4) :-
   clt2(T2),
   cllp(1),
   llcr(1),
   time_less(T2).
calc_statetransition(8, 9) :-
   clt2(T2),
   llcr(1),
   not(time_less(T2)).

calc_statetransition(9, 1) :-
   not(llcr(1)).
calc_statetransition(9, 1) :-
   ddb_lle(0).
calc_statetransition(9, 1) :-
   dm_passall(0).
calc_statetransition(9, 5) :-
   clt2(T2),
   cllp(1),
   llcr(1),
   time_less(T2).
calc_statetransition(9, 8) :-
   clt2(T2),
   llcr(1),
   not(time_less(T2)).

calc_statetransition(10, 11) :-
   vc_chtlack(0).

calc_statetransition(11, 12) :-
   vc_lchact(0).

calc_statetransition(12, 1).

calc_statetransition(13, 1) :-
    not(rlcr(1)).
```

```
calc_statetransition(13, 2) :-
   llcr(1),
   ddb_lle(1),
   dm_passall(1).
calc_statetransition(13, 14) :-
   clt2(T2),
   crlp(1),
   rlcr(1),
   time_less(T2).
calc_statetransition(13, 18) :-
   clt2(T2),
   rlcr(1),
   not(time_less(T2)).

calc_statetransition(14, 15) :-
   clt3(T3),
   crlp(1),
   rlcr(1),
   not(time_less(T3)).
calc_statetransition(14, 16) :-
   not(crlp(1)).
calc_statetransition(14, 16) :-
   not(rlcr(1)).

calc_statetransition(15, 17) :-
   vc_chtlack(1).

calc_statetransition(16, 1).

calc_statetransition(17, 16) :-
    vc_lchact(0),
    vc_chtlack(0).
calc_statetransition(17, 20) :-
   ccrl(1).

calc_statetransition(18, 1) :-
   not(rlcr(1)).
calc_statetransition(18, 1) :-
   ddb_rle(0).
calc_statetransition(18, 14) :-
   clt2(T2),
   crlp(1),
   rlcr(1),
   time_less(T2).
calc_statetransition(18, 19) :-
```

```
   clt2(T2),
   rlcr(1),
   not(time_less(T2)).

calc_statetransition(19, 1) :-
   not(rlcr(1)).
calc_statetransition(19, 1) :-
   ddb_rle(0).
calc_statetransition(19, 15) :-
   clt2(T2),
   crlp(1),
   rlcr(1),
   time_less(T2).
 calc_statetransition(19, 18) :-
   clt2(T2),
   rlcr(1),
   not(time_less(T2)).

calc_statetransition(20, 21) :-
   vc_chtlack(0).

calc_statetransition(21, 22) :-
   vc_lchact(0).

calc_statetransition(22, 1).

calc_statetransition(23, 1) :-
   cim(1).

calc_statetransition(24, 1) :-
   vc_emstop(0).

calc_statetransition(State, State).

%-----------------------------------------------------------------

% Calculating the state transistions

execute_stateactions(1, 0, 0, 0, 0, 0).
execute_stateactions(2, 0, 0, 0, 0, 0).
execute_stateactions(3, 0, 0, 0, 0, 0).
execute_stateactions(4, 0, 0, 0, 2, 0).
execute_stateactions(5, 2, 0, 0, 2, 0).
execute_stateactions(6, 0, 0, 0, 0, 0).
execute_stateactions(7, 0, 0, 0, 2, 0).
```

```
execute_stateactions(8, 0, 0, 0, 2, 0).
execute_stateactions(9, 0, 0, 1, 2, 0).
execute_stateactions(10, 0, 1, 0, 1, 0).
execute_stateactions(11, 0, 0, 0, 1, 0).
execute_stateactions(12, 0, 0, 0, 0, 0).
execute_stateactions(13, 0, 0, 0, 0, 0).
execute_stateactions(14, 0, 0, 0, 1, 0).
execute_stateactions(15, 1, 0, 0, 1, 0).
execute_stateactions(16, 0, 0, 0, 0, 0).
execute_stateactions(17, 0, 0, 0, 1, 0).
execute_stateactions(18, 0, 0, 0, 1, 0).
execute_stateactions(19, 0, 0, 1, 1, 0).
execute_stateactions(20, 0, 1, 0, 2, 0).
execute_stateactions(21, 0, 0, 0, 2, 0).
execute_stateactions(22, 0, 0, 0, 0, 0).
execute_stateactions(23, 0, 0, 0, 0, 0).
execute_stateactions(24, 0, 0, 0, 3, 1).

%-------------------------------------------------------------

% Main execution routine of the car

execute_anything(_, error).
execute_anything(_, nothing).
execute_anything(_, last) :-
   execute_anything(_, ok).
execute_anything(NewState, ok) :-
   state(State),
   calc_statetransition(State, NewState),
   execute_stateactions(NewState, LCH, CLCH, OSCI, TRAFF, EMST).

%-------------------------------------------------------------

% Starting the car

check_activation(error, _, error).
check_activation(nothing, _, off).
check_activation(last, _, off).
check_activation(ok, wait, wait) :-
   dm_state(DMS),
   DMS =\= 0,
   DMS =\= 2.
check_activation(ok, wait, wait) :-
   rt_state(RTS),
   RTS =\= 2.
```

```
check_activation(ok, wait, wait) :-
   odt_state(ODTS),
   ODTS =\= 2.
check_activation(ok, wait, wait) :-
   vc_state(VCS),
   VCS =\= 2.
check_activation(ok, wait, off) :-
   dm_state(0).
check_activation(ok, wait, active) :-
   dm_state(2),
   rt_state(2),
   odt_state(2),
   vc_state(2).
check_activation(ok, active, active).

%-------------------------------------------------------------

% Main routine

% Waiting for activation
main_loop(wait) :-
   check_activation(Result, wait, State),
   main_loop(FState).

% Executing after activation
main_loop(active) :-
   check_activation(Result, active, State),
   execute_anything(NewState, Result),
   main_loop(State).

% end
main_loop(off).

% stop by error
main_loop(error).
```

6 Appendix C

This part contains some examples for input and output of the rule system listed above.

```
% Input:
% DMS RTS ODTS VCS VMS CONV VPL LLE RLE PA
% CLLP CRLP CCLL CCRL LACK LCHA CWRL EMS VL VF BO BL BR CLCK

# Start
1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0
1 2 2 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 2
2 2 2 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 4

# Activation
2 2 2 2 0 0 80 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 6

# Car drives
2 2 2 2 20 0 80 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 8
2 2 2 2 40 0 80 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 10
2 2 2 2 60 0 80 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 12
2 2 2 2 80 0 80 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 14

# Emergency stop
2 2 2 2 80 0 80 0 0 0 0 0 0 0 0 0 0 1 0 0 1 0 0 16
2 2 2 2 60 0 80 0 0 0 0 0 0 0 0 0 0 1 0 0 1 0 0 18
2 2 2 2 40 0 80 0 0 0 0 0 0 0 0 0 0 1 0 0 1 0 0 20
2 2 2 2 20 0 80 0 0 0 0 0 0 0 0 0 0 1 0 0 1 0 0 22
2 2 2 2 0 0 80 0 0 0 0 0 0 0 0 0 0 1 0 0 1 0 0 24
2 2 2 2 0 0 80 0 0 0 0 0 0 0 0 0 0 1 0 0 1 0 0 26

# Car drives again
2 2 2 2 20 0 80 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 28
2 2 2 2 40 0 80 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 30
2 2 2 2 60 0 80 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 32
2 2 2 2 80 0 80 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 34

# Change to the right lane
2 2 2 2 80 0 80 0 1 0 0 0 0 0 0 0 0 0 0 0 1 0 1 44
2 2 2 2 80 0 80 0 1 0 0 1 0 0 0 0 0 0 0 0 1 0 1 46
2 2 2 2 80 0 80 0 1 0 0 1 0 0 0 0 0 0 0 0 1 0 1 48
2 2 2 2 80 0 80 0 1 0 0 1 0 0 0 0 0 0 0 0 1 0 1 50
2 2 2 2 80 0 80 0 1 0 0 1 0 0 0 0 0 0 0 0 1 0 1 52
2 2 2 2 80 0 80 0 1 0 0 1 0 0 0 0 0 0 0 0 1 0 1 54
2 2 2 2 80 0 80 0 1 0 0 1 0 0 1 0 0 0 0 0 1 0 1 56
2 2 2 2 80 0 80 0 1 0 0 1 0 0 1 1 0 0 0 0 1 0 1 58
```

```
2 2 2 2 80 0 80 0 1 0 0 1 0 0 1 1 0 0 0 0 1 0 1 60
2 2 2 2 80 0 80 0 1 0 0 1 0 0 1 1 1 0 0 0 1 1 0 62
2 2 2 2 80 0 80 0 1 0 0 1 0 0 0 1 1 0 0 0 1 1 0 64
2 2 2 2 80 0 80 0 1 0 0 1 0 0 0 0 0 0 0 0 1 1 0 66
2 2 2 2 80 0 80 0 1 0 0 1 0 0 0 0 0 0 0 0 1 1 0 68

# Another car gets visible in front
2 2 2 2 80 0 80 0 0 0 0 1 0 0 0 0 0 0 0 60 1 0 0 272
2 2 2 2 70 0 80 0 0 0 0 0 0 0 0 0 0 0 0 60 1 0 0 274
2 2 2 2 60 0 80 0 0 0 0 0 0 0 0 0 0 0 0 60 1 0 0 276

# Driving in a convoy
2 2 2 2 60 1 80 0 0 0 0 0 0 0 0 0 0 0 0 60 1 0 0 278
2 2 2 2 60 1 80 0 0 0 0 0 0 0 0 0 0 0 0 60 1 0 0 280

# Left lane exists, Overtaking allowed
# but volocity on the left lane to slow
2 2 2 2 60 1 80 1 0 1 0 0 0 0 0 0 0 0 60 60 1 0 0 282

# Now velocity on the left lane is fast enough
2 2 2 2 60 1 80 1 0 1 0 0 0 0 0 0 0 0 80 60 1 0 0 284
2 2 2 2 60 1 80 1 0 1 0 0 0 0 0 0 0 0 80 60 1 0 0 286

# Overtaking not allowed
2 2 2 2 60 1 80 1 0 0 0 0 0 0 0 0 0 0 80 60 1 0 0 288
2 2 2 2 60 1 80 1 0 0 0 0 0 0 0 0 0 0 80 60 1 0 0 290

# Overtaking allowed again
2 2 2 2 60 1 80 1 0 1 0 0 0 0 0 0 0 0 80 60 1 0 0 292
2 2 2 2 60 1 80 1 0 1 0 0 0 0 0 0 0 0 80 60 1 0 0 294
2 2 2 2 60 1 80 1 0 1 0 0 0 0 0 0 0 0 80 60 1 0 0 296
2 2 2 2 60 1 80 1 0 1 0 0 0 0 0 0 0 0 80 60 1 0 0 298

# Changing to left lane, but then stopping the change
# because the left lane gets insecure
2 2 2 2 60 1 80 0 0 1 1 0 0 0 0 0 0 0 80 60 1 1 0 376
2 2 2 2 60 1 80 1 0 1 1 0 0 0 0 0 0 0 80 60 1 1 0 378
2 2 2 2 60 1 80 1 0 1 1 0 0 0 0 0 0 0 80 60 1 1 0 380
2 2 2 2 60 1 80 1 0 1 1 0 0 0 0 0 0 0 80 60 1 1 0 382
2 2 2 2 60 1 80 1 0 1 1 0 0 0 0 0 0 0 80 60 1 1 0 384
2 2 2 2 60 1 80 1 0 1 1 0 0 0 0 0 0 0 80 60 1 1 0 386
2 2 2 2 60 1 80 1 0 1 1 0 0 0 0 0 0 0 80 60 1 1 0 388
2 2 2 2 60 1 80 1 0 1 1 0 0 0 0 0 0 0 80 60 1 1 0 390
2 2 2 2 60 1 80 1 0 1 1 0 0 0 0 0 0 0 80 60 1 1 0 392
2 2 2 2 60 1 80 1 0 1 1 0 0 0 0 0 0 0 80 60 1 1 0 394
```

```
2 2 2 2 60 1 80 1 0 1 1 0 0 0 0 0 0 0 80 60 1 1 0 396
2 2 2 2 80 1 80 1 0 1 1 0 0 0 0 0 0 0 80 60 1 1 0 398
2 2 2 2 60 1 80 1 0 1 1 0 0 0 2 0 0 0 80 60 1 1 0 400
2 2 2 2 60 1 80 1 0 1 1 0 0 0 2 1 0 0 80 60 1 1 0 402
2 2 2 2 60 1 80 1 0 1 1 0 1 0 2 1 0 0 80 60 1 0.5 0 404
2 2 2 2 60 1 80 1 0 1 1 0 1 0 2 1 2 0 80 60 1 0.5 0 406
2 2 2 2 60 1 80 1 0 1 1 0 1 0 2 1 2 0 80 60 1 0.5 0 408
2 2 2 2 60 1 80 1 0 1 1 0 0 0 0 1 2 0 80 60 1 0 0 410
2 2 2 2 60 1 80 1 0 1 1 0 0 0 0 1 2 0 80 60 1 0 0 412
2 2 2 2 60 1 80 1 0 1 1 0 0 0 0 1 0 0 80 60 1 0 0 414

# End
```

This is the calculated output

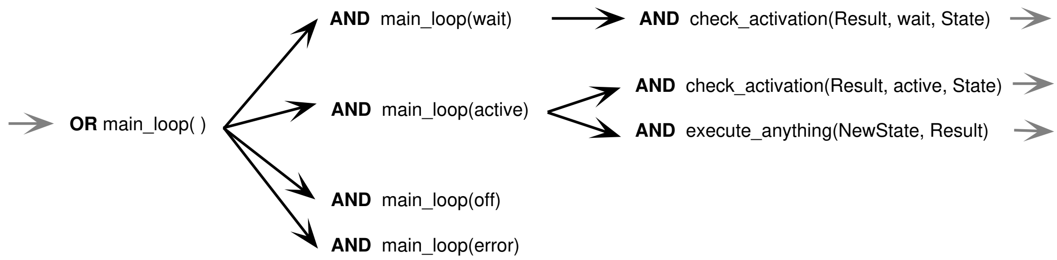
```
% Output:
% CIM LLCR RLCR CLLP CRLP CCLL CCRL -> ST LCH CLCH OSC TRAF EMST

# Start

# Activation
0  0  0  0  0  0  0 -> 0  0  0  0  0  State: 23 Time: 0

# Car drives
1  0  0  0  0  0  0 -> 0  0  0  0  0  State: 1  Time: 0
1  0  0  0  0  0  0 -> 0  0  0  0  0      Time: 2
1  0  0  0  0  0  0 -> 0  0  0  0  0      Time: 4
1  0  0  0  0  0  0 -> 0  0  0  0  0      Time: 6

# Emergency stop
1  0  0  0  0  0  0 -> 0  0  0  3  1  State: 24 Time: 0
1  0  0  0  0  0  0 -> 0  0  0  3  1      Time: 2
1  0  0  0  0  0  0 -> 0  0  0  3  1      Time: 4
1  0  0  0  0  0  0 -> 0  0  0  3  1      Time: 6
0  0  0  0  0  0  0 -> 0  0  0  3  1      Time: 8
0  0  0  0  0  0  0 -> 0  0  0  3  1      Time: 10

# Car drives again
1  0  0  0  0  0  0 -> 0  0  0  0  0  State: 1  Time: 0
1  0  0  0  0  0  0 -> 0  0  0  0  0      Time: 2
1  0  0  0  0  0  0 -> 0  0  0  0  0      Time: 4
1  0  0  0  0  0  0 -> 0  0  0  0  0      Time: 6

# Change to the right lane
```

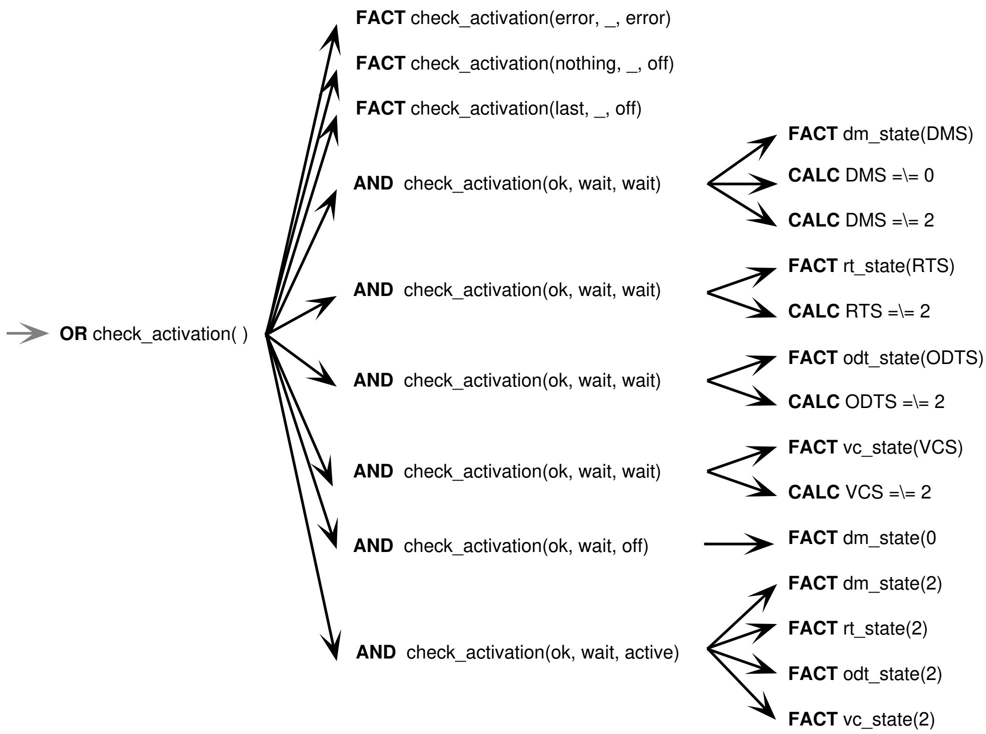
```
1  0  1  0  0  0  0 -> 0  0  0  0  0  State: 13 Time: 0
1  0  1  0  1  0  0 -> 0  0  0  1  0  State: 14 Time: 0
1  0  1  0  1  0  0 -> 0  0  0  1  0      Time: 2
1  0  1  0  1  0  0 -> 0  0  0  1  0      Time: 4
1  0  1  0  1  0  0 -> 1  0  0  1  0  State: 15 Time: 0
1  0  1  0  1  0  0 -> 1  0  0  1  0      Time: 2
1  0  1  0  1  0  0 -> 0  0  0  1  0  State: 17 Time: 0
1  0  1  0  1  0  0 -> 0  0  0  1  0      Time: 2
1  0  1  0  1  0  0 -> 0  0  0  1  0      Time: 4
1  0  1  0  0  0  0 -> 0  0  0  1  0      Time: 6
1  0  1  0  0  0  0 -> 0  0  0  1  0      Time: 8
1  0  1  0  0  0  0 -> 0  0  0  0  0  State: 16 Time: 0
1  0  1  0  0  0  0 -> 0  0  0  0  0  State: 1  Time: 0

# Another car gets visible in front
1  0  0  0  0  0  0 -> 0  0  0  0  0  State: 1  Time: 0
1  0  0  0  0  0  0 -> 0  0  0  0  0      Time: 2
1  0  0  0  0  0  0 -> 0  0  0  0  0      Time: 4

# Driving in a convoy
1  0  0  0  0  0  0 -> 0  0  0  0  0      Time: 6
1  0  0  0  0  0  0 -> 0  0  0  0  0      Time: 8

# Left lane exists, Overtaking allowed
# but volocity on the left lane to slow
1  0  0  0  0  0  0 -> 0  0  0  0  0      Time: 10

# Now velocity on the left lane is fast enough
1  1  0  0  0  0  0 -> 0  0  0  0  0  State: 2  Time: 0
1  1  0  0  0  0  0 -> 0  0  0  0  0      Time: 2

# Overtaking not allowed
1  1  0  0  0  0  0 -> 0  0  0  0  0  State: 1  Time: 0
1  1  0  0  0  0  0 -> 0  0  0  0  0      Time: 2

# Overtaking allowed again
1  1  0  0  0  0  0 -> 0  0  0  0  0  State: 2  Time: 0
1  1  0  0  0  0  0 -> 0  0  0  0  0      Time: 2
1  1  0  0  0  0  0 -> 0  0  0  0  0  State: 3  Time: 0
1  1  0  0  0  0  0 -> 0  0  0  0  0      Time: 2

# Changing to left lane, but then stopping the change
# because the left lane gets insecure
1  0  0  0  0  0  0 -> 0  0  0  0  0      Time: 2
1  1  0  1  0  0  0 -> 0  0  0  0  0  State: 2  Time: 0
```

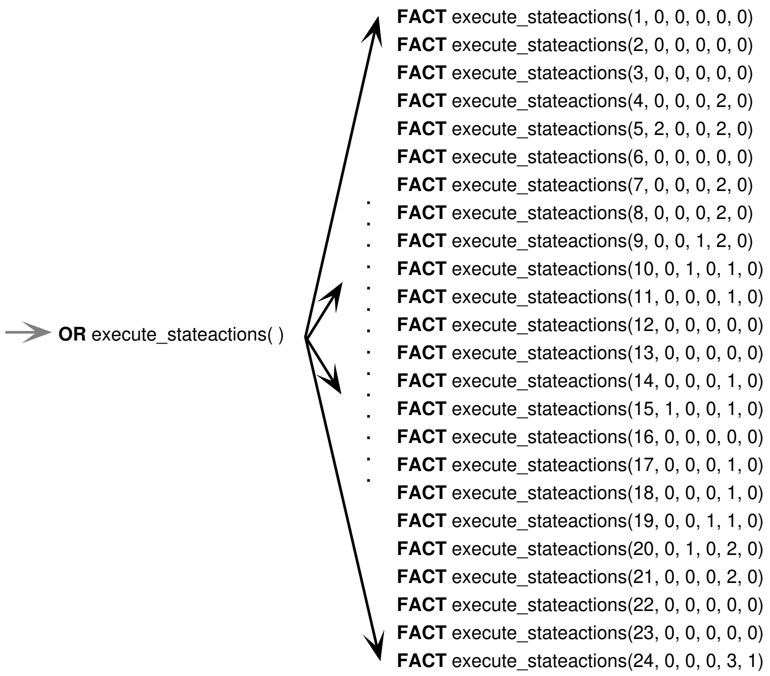
```
1  1  0  1  0  0  0 -> 0  0  0  0  0      Time: 2
1  1  0  1  0  0  0 -> 0  0  0  0  0  State: 3  Time: 0
1  1  0  1  0  0  0 -> 0  0  0  2  0  State: 4  Time: 0
1  1  0  1  0  0  0 -> 0  0  0  2  0      Time: 2
1  1  0  1  0  0  0 -> 0  0  0  2  0      Time: 4
1  1  0  1  0  0  0 -> 2  0  0  2  0  State: 5  Time: 0
1  1  0  1  0  0  0 -> 2  0  0  2  0      Time: 2
1  1  0  1  0  0  0 -> 2  0  0  2  0      Time: 4
1  1  0  1  0  0  0 -> 2  0  0  2  0      Time: 6
1  0  0  1  0  0  0 -> 2  0  0  2  0      Time: 8
1  1  0  1  0  0  0 -> 0  0  0  2  0  State: 7  Time: 0
1  1  0  1  0  0  0 -> 0  0  0  2  0      Time: 2
1  1  0  0  0  1  0 -> 0  1  0  1  0  State: 10 Time: 0
1  1  0  0  0  0  0 -> 0  1  0  1  0      Time: 2
1  1  0  0  0  0  0 -> 0  1  0  1  0      Time: 4
1  1  0  0  0  0  0 -> 0  0  0  1  0  State: 11 Time: 0
1  1  0  0  0  0  0 -> 0  0  0  1  0      Time: 2
1  1  0  0  0  0  0 -> 0  0  0  1  0      Time: 4

# End
```

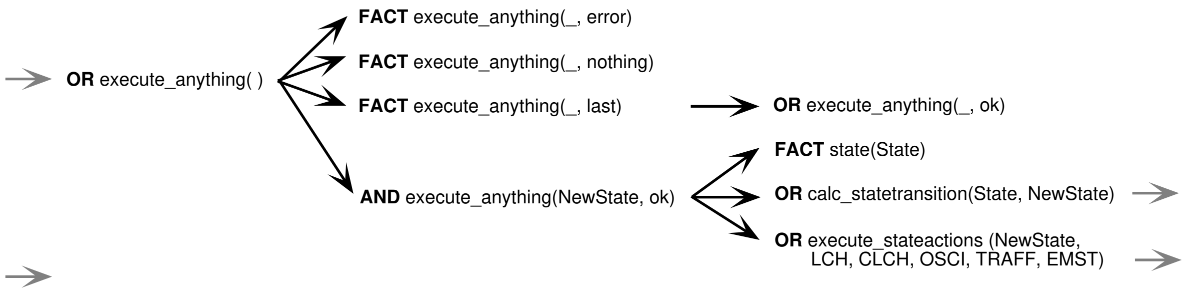
Appendix D

This Appendix contains the structure graph fragments according to the Prolog program above. Due to a better shape, the graphs are now drawn from left to right instead of top down.



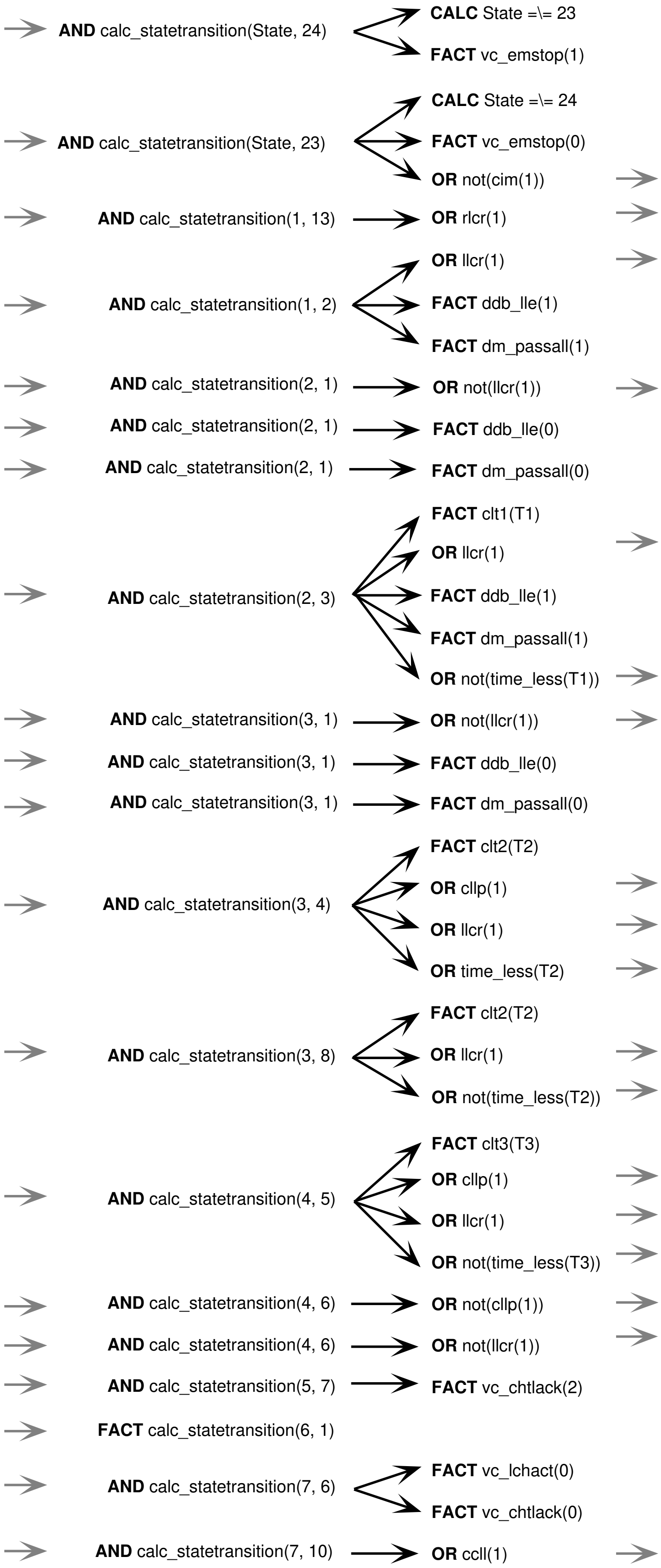


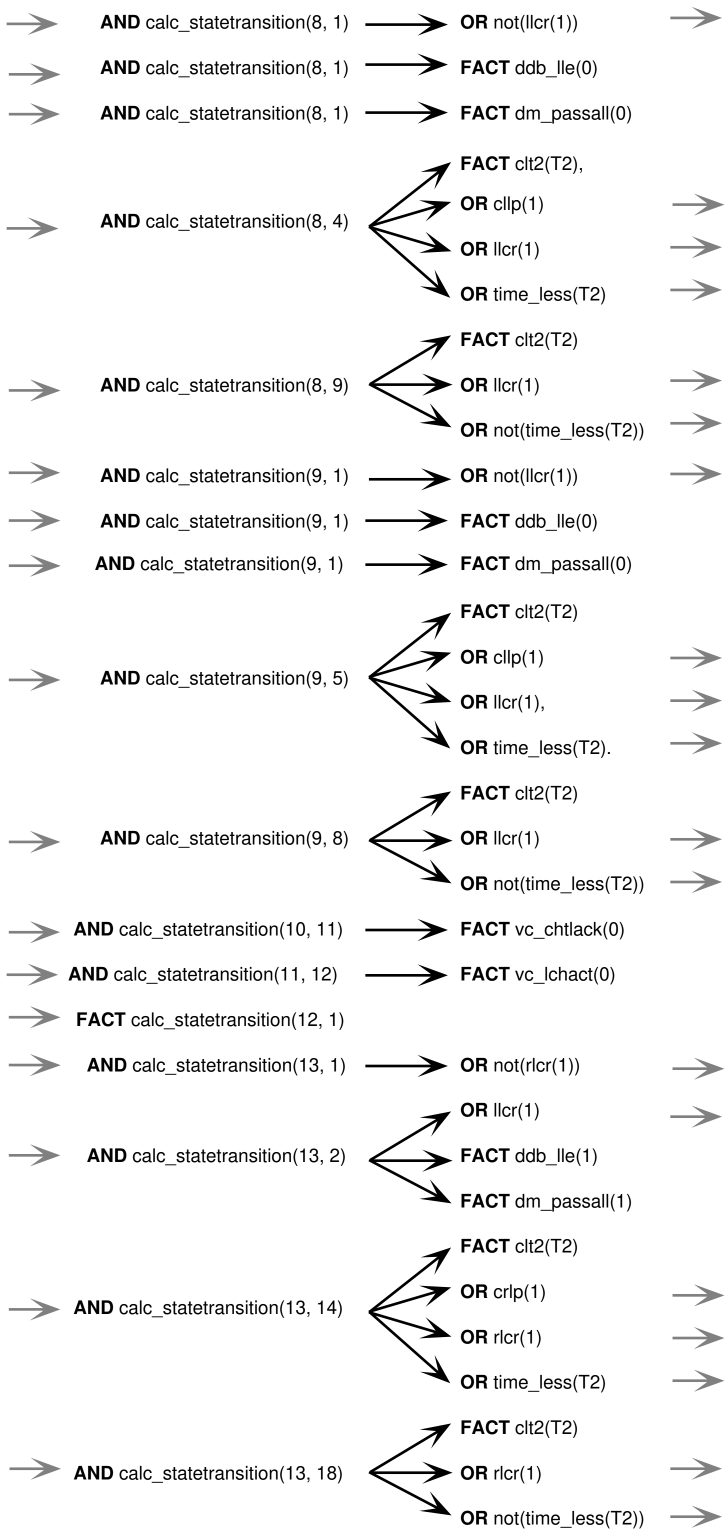

OR execute_stateactions( )
FACT execute_stateactions(1, 0, 0, 0, 0, 0)
FACT execute_stateactions(2, 0, 0, 0, 0, 0)
FACT execute_stateactions(3, 0, 0, 0, 0, 0)
FACT execute_stateactions(4, 0, 0, 0, 2, 0)
FACT execute_stateactions(5, 2, 0, 0, 2, 0)
FACT execute_stateactions(6, 0, 0, 0, 0, 0)
FACT execute_stateactions(7, 0, 0, 0, 2, 0)
FACT execute_stateactions(8, 0, 0, 0, 2, 0)
FACT execute_stateactions(9, 0, 0, 1, 2, 0)
FACT execute_stateactions(10, 0, 1, 0, 1, 0)
FACT execute_stateactions(11, 0, 0, 0, 1, 0)
FACT execute_stateactions(12, 0, 0, 0, 0, 0)
FACT execute_stateactions(13, 0, 0, 0, 0, 0)
FACT execute_stateactions(14, 0, 0, 0, 1, 0)
FACT execute_stateactions(15, 1, 0, 0, 1, 0)
FACT execute_stateactions(16, 0, 0, 0, 0, 0)
FACT execute_stateactions(17, 0, 0, 0, 1, 0)
FACT execute_stateactions(18, 0, 0, 0, 1, 0)
FACT execute_stateactions(19, 0, 0, 1, 1, 0)
FACT execute_stateactions(20, 0, 1, 0, 2, 0)
FACT execute_stateactions(21, 0, 0, 0, 2, 0)
FACT execute_stateactions(22, 0, 0, 0, 0, 0)
FACT execute_stateactions(23, 0, 0, 0, 0, 0)
FACT execute_stateactions(24, 0, 0, 0, 3, 1)

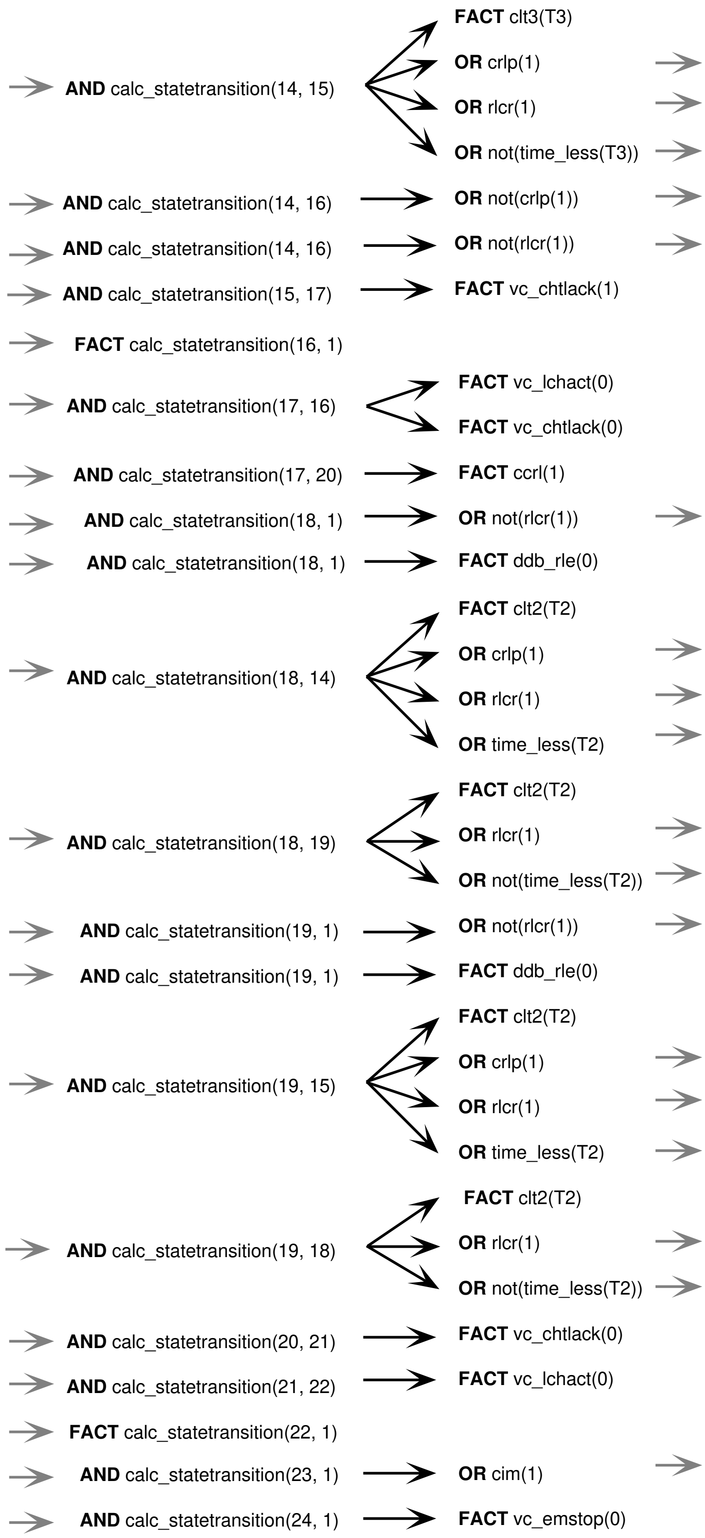

OR execute_anything( )
FACT execute_anything(_, error)
FACT execute_anything(_, nothing)
FACT execute_anything(_, last)
OR execute_anything(_, ok)
AND execute_anything(NewState, ok)
FACT state(State)
OR calc_statetransition(State, NewState)
OR execute_stateactions (NewState,
LCH, CLCH, OSCI, TRAFF, EMST)

OR calc_statetransistions( )

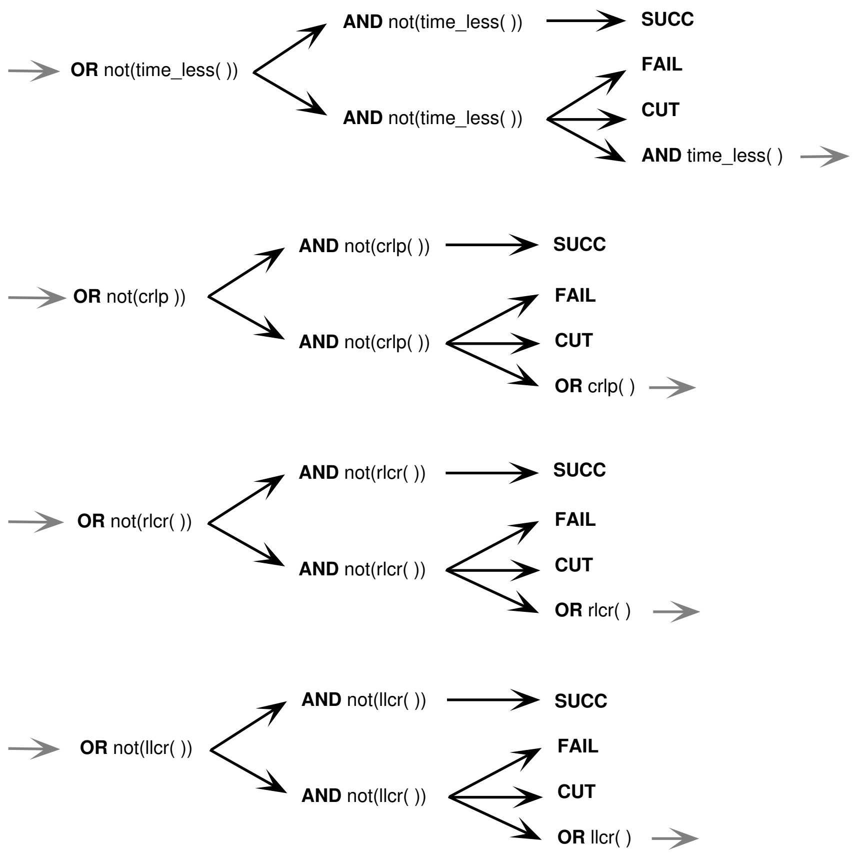
AND calc_statetransistions(State, 24)
AND calc_statetransistions(State, 23)
AND calc_statetransistions(1, 13)
AND calc_statetransistions(1, 2)
AND calc_statetransistions(2, 1)
AND calc_statetransistions(2, 1)
AND calc_statetransistions(2, 1)
AND calc_statetransistions(2, 3)
AND calc_statetransistions(3, 1)
AND calc_statetransistions(3, 1)
AND calc_statetransistions(3, 1)
AND calc_statetransistions(3, 4)
AND calc_statetransistions(3, 8)
AND calc_statetransistions(4, 5)
AND calc_statetransistions(4, 6)
AND calc_statetransistions(4, 6)
AND calc_statetransistions(5, 7)
FACT calc_statetransistions(6, 1)
AND calc_statetransistions(7, 6)
AND calc_statetransistions(7, 10)
AND calc_statetransistions(8, 1)
AND calc_statetransistions(8, 1)
AND calc_statetransistions(8, 1)
AND calc_statetransistions(8, 4)
AND calc_statetransistions(8, 9)
AND calc_statetransistions(9, 1)
AND calc_statetransistions(9, 1)
AND calc_statetransistions(9, 1)
AND calc_statetransistions(9, 5)
AND calc_statetransistions(9, 8)
AND calc_statetransistions(10, 11)
AND calc_statetransistions(11, 12)
FACT calc_statetransistions(12, 1)
AND calc_statetransistions(13, 1)
AND calc_statetransistions(13, 2)
AND calc_statetransistions(13, 14)
AND calc_statetransistions(13, 18)
AND calc_statetransistions(14, 15)
AND calc_statetransistions(14, 16)
AND calc_statetransistions(14, 16)
AND calc_statetransistions(15, 17)
FACT calc_statetransistions(16, 1)
AND calc_statetransistions(17, 16)
AND calc_statetransistions(17, 20)
AND calc_statetransistions(18, 1)
AND calc_statetransistions(18, 1)
AND calc_statetransistions(18, 14)
AND calc_statetransistions(18, 19)
AND calc_statetransistions(19, 1)
AND calc_statetransistions(19, 1)
AND calc_statetransistions(19, 15)
AND calc_statetransistions(19, 18)
AND calc_statetransistions(20, 21)
AND calc_statetransistions(21, 22)
FACT calc_statetransistions(22, 1)
AND calc_statetransistions(23, 1)
AND calc_statetransistions(24, 1)
FACT calc_statetransistions(State, State)

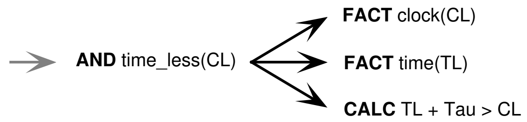

AND calc_statetransition(State, 24)
CALC State =\= 23
FACT vc_emstop(1)
AND calc_statetransition(State, 23)
CALC State =\= 24
FACT vc_emstop(0)
OR not(cim(1))
AND calc_statetransition(1, 13)
OR rlcr(1)
AND calc_statetransition(1, 2)
OR llcr(1)
FACT ddb_lle(1)
FACT dm_passall(1)
AND calc_statetransition(2, 1)
OR not(llcr(1))
AND calc_statetransition(2, 1)
FACT ddb_lle(0)
AND calc_statetransition(2, 1)
FACT dm_passall(0)
AND calc_statetransition(2, 3)
FACT clt1(T1)
OR llcr(1)
FACT ddb_lle(1)
FACT dm_passall(1)
OR not(time_less(T1))
AND calc_statetransition(3, 1)
OR not(llcr(1))
AND calc_statetransition(3, 1)
FACT ddb_lle(0)
AND calc_statetransition(3, 1)
FACT dm_passall(0)
AND calc_statetransition(3, 4)
FACT clt2(T2)
OR cllp(1)
OR llcr(1)
OR time_less(T2)
AND calc_statetransition(3, 8)
FACT clt2(T2)
OR llcr(1)
OR not(time_less(T2))
AND calc_statetransition(4, 5)
FACT clt3(T3)
OR cllp(1)
OR llcr(1)
OR not(time_less(T3))
AND calc_statetransition(4, 6)
OR not(cllp(1))
AND calc_statetransition(4, 6)
OR not(llcr(1))
AND calc_statetransition(5, 7)
FACT vc_chtlack(2)
FACT calc_statetransition(6, 1)
AND calc_statetransition(7, 6)
FACT vc_lchact(0)
FACT vc_chtlack(0)
AND calc_statetransition(7, 10)
OR ccll(1)

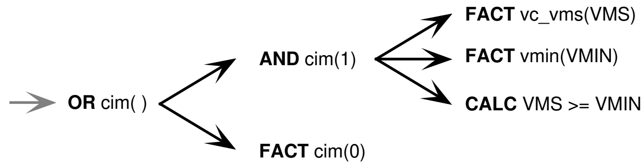

AND calc_statetransition(8, 1)
OR not(llcr(1))
AND calc_statetransition(8, 1)
FACT ddb_lle(0)
AND calc_statetransition(8, 1)
FACT dm_passall(0)
AND calc_statetransition(8, 4)
FACT clt2(T2),
OR cllp(1)
OR llcr(1)
OR time_less(T2)
AND calc_statetransition(8, 9)
FACT clt2(T2)
OR llcr(1)
OR not(time_less(T2))
AND calc_statetransition(9, 1)
OR not(llcr(1))
AND calc_statetransition(9, 1)
FACT ddb_lle(0)
AND calc_statetransition(9, 1)
FACT dm_passall(0)
AND calc_statetransition(9, 5)
FACT clt2(T2)
OR cllp(1)
OR llcr(1),
OR time_less(T2).
AND calc_statetransition(9, 8)
FACT clt2(T2)
OR llcr(1)
OR not(time_less(T2))
AND calc_statetransition(10, 11)
FACT vc_chtlack(0)
AND calc_statetransition(11, 12)
FACT vc_lchact(0)
FACT calc_statetransition(12, 1)
AND calc_statetransition(13, 1)
OR not(rlcr(1))
AND calc_statetransition(13, 2)
OR llcr(1)
FACT ddb_lle(1)
FACT dm_passall(1)
AND calc_statetransition(13, 14)
FACT clt2(T2)
OR crlp(1)
OR rlcr(1)
OR time_less(T2)
AND calc_statetransition(13, 18)
FACT clt2(T2)
OR rlcr(1)
OR not(time_less(T2))

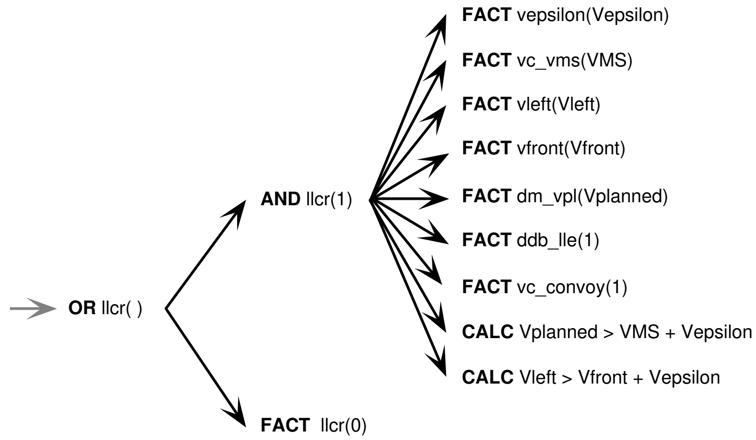


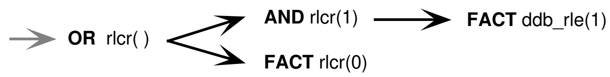
FACT clt3(T3)
AND calc_statetransition(14, 15)
OR crlp(1)
OR rlcr(1)
OR not(time_less(T3))
AND calc_statetransition(14, 16)
OR not(crlp(1))
AND calc_statetransition(14, 16)
OR not(rlcr(1))
AND calc_statetransition(15, 17)
FACT vc_chtlack(1)
FACT calc_statetransition(16, 1)
FACT vc_lchact(0)
AND calc_statetransition(17, 16)
FACT vc_chtlack(0)
AND calc_statetransition(17, 20)
FACT ccrl(1)
AND calc_statetransition(18, 1)
OR not(rlcr(1))
AND calc_statetransition(18, 1)
FACT ddb_rle(0)
FACT clt2(T2)
AND calc_statetransition(18, 14)
OR crlp(1)
OR rlcr(1)
OR time_less(T2)
FACT clt2(T2)
AND calc_statetransition(18, 19)
OR rlcr(1)
OR not(time_less(T2))
AND calc_statetransition(19, 1)
OR not(rlcr(1))
AND calc_statetransition(19, 1)
FACT ddb_rle(0)
FACT clt2(T2)
AND calc_statetransition(19, 15)
OR crlp(1)
OR rlcr(1)
OR time_less(T2)
FACT clt2(T2)
AND calc_statetransition(19, 18)
OR rlcr(1)
OR not(time_less(T2))
AND calc_statetransition(20, 21)
FACT vc_chtlack(0)
AND calc_statetransition(21, 22)
FACT vc_lchact(0)
FACT calc_statetransition(22, 1)
AND calc_statetransition(23, 1)
OR cim(1)
AND calc_statetransition(24, 1)
FACT vc_emstop(0)

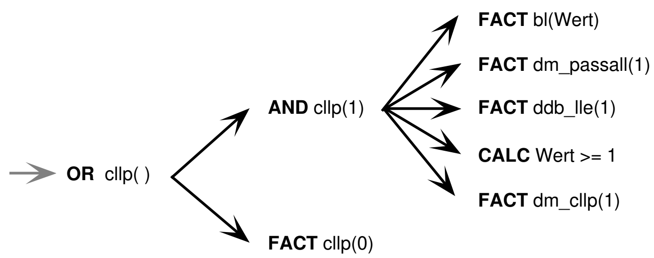

OR not(time_less( ))
AND not(time_less( ))
SUCC
AND not(time_less( ))
FAIL
CUT
AND time_less( )
OR not(crlp))
AND not(crlp( ))
SUCC
AND not(crlp( ))
FAIL
CUT
OR crlp( )
OR not(rlcr( ))
AND not(rlcr( ))
SUCC
AND not(rlcr( ))
FAIL
CUT
OR rlcr( )
OR not(llcr( ))
AND not(llcr( ))
SUCC
AND not(llcr( ))
FAIL
CUT
OR llcr( )

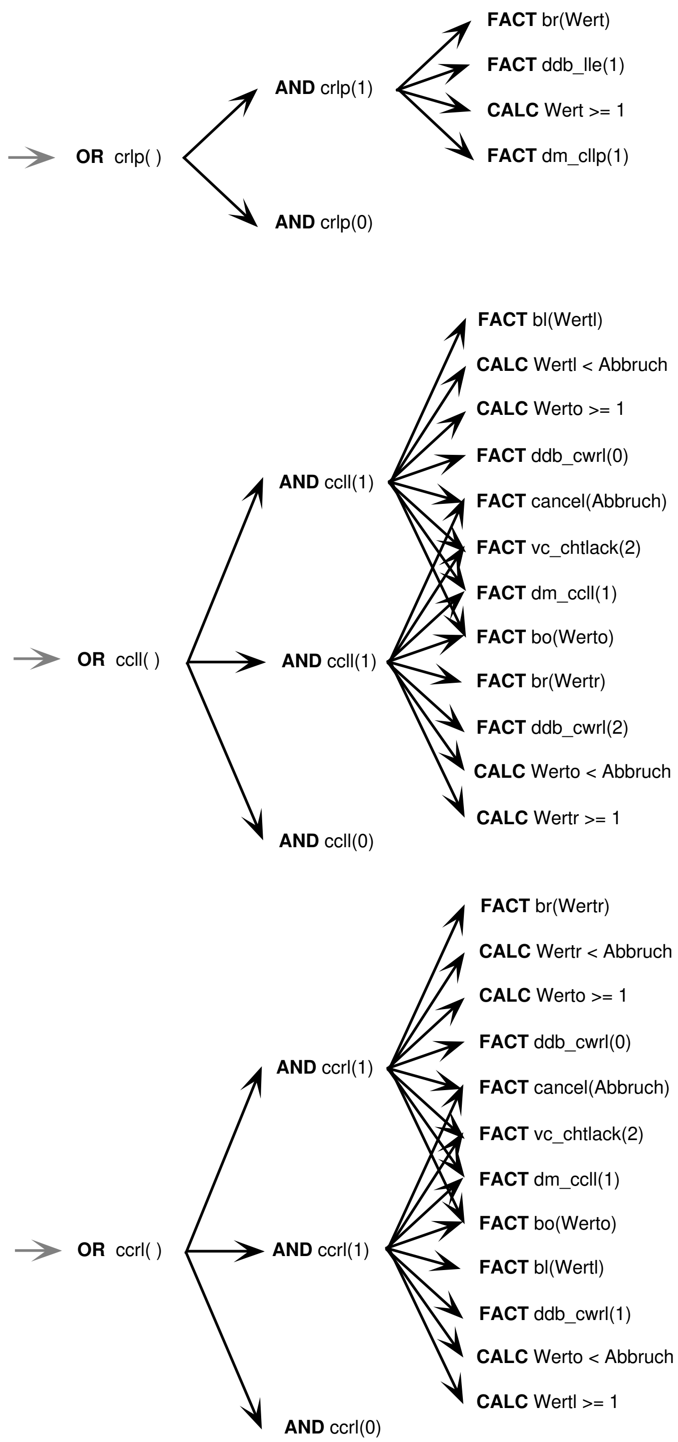

AND time_less(CL)
FACT clock(CL)
FACT time(TL)
CALC TL + Tau > CL


OR cim( )
AND cim(1)
FACT vc_vms(VMS)
FACT vmin(VMIN)
CALC VMS >= VMIN
FACT cim(0)


OR llcr( )
AND llcr(1)
FACT vepsilon(Vepsilon)
FACT vc_vms(VMS)
FACT vleft(Vleft)
FACT vfront(Vfront)
FACT dm_vpl(Vplanned)
FACT ddb_lle(1)
FACT vc_convoy(1)
CALC Vplanned > VMS + Vepsilon
CALC Vleft > Vfront + Vepsilon
FACT llcr(0)

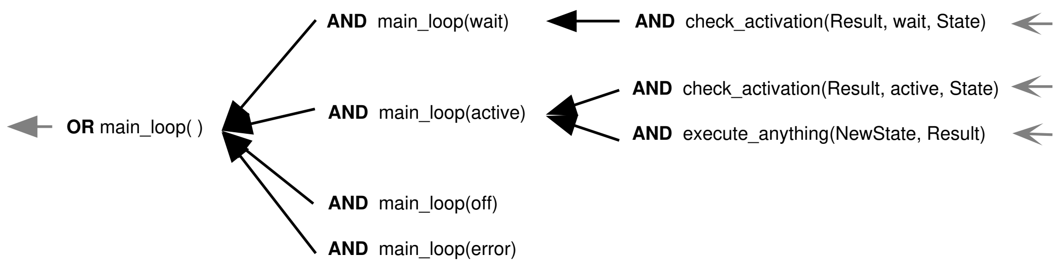

OR rlcr( )
AND rlcr(1)
FACT ddb_rle(1)
FACT rlcr(0)

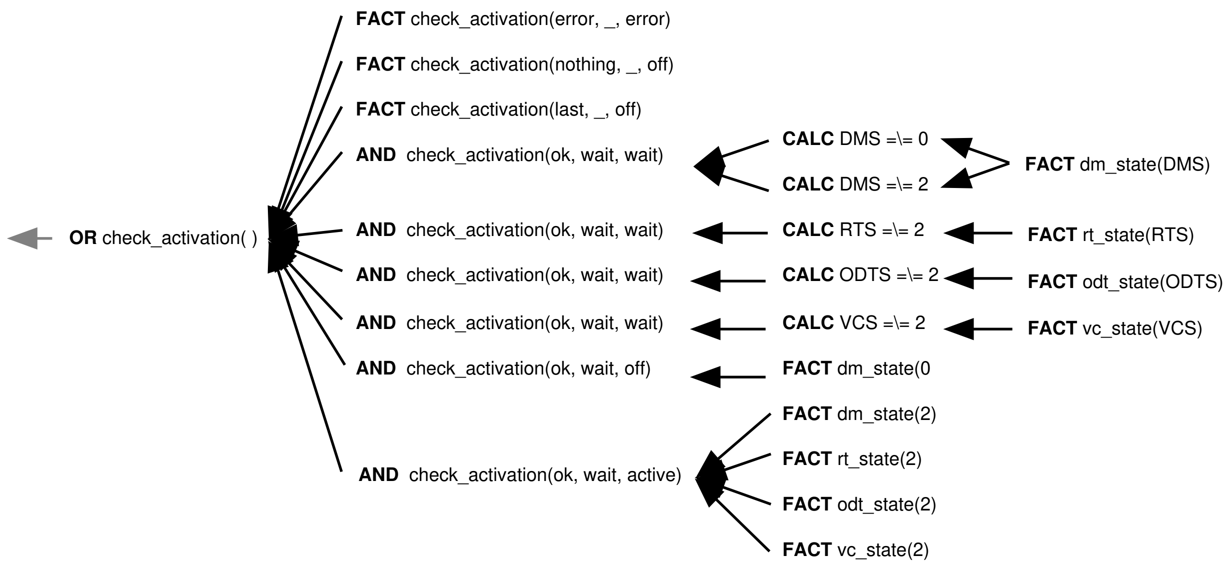

OR cllp( )
AND cllp(1)
FACT bl(Wert)
FACT dm_passall(1)
FACT ddb_lle(1)
CALC Wert >= 1
FACT dm_cllp(1)
FACT cllp(0)

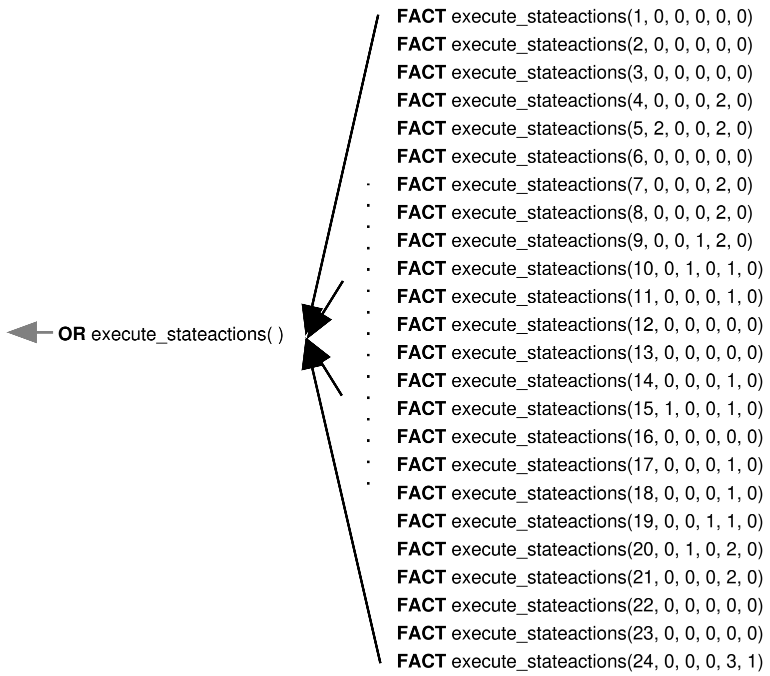

FACT br(Wert)
FACT ddb_lle(1)
CALC Wert >= 1
FACT dm_cllp(1)
AND crlp(1)
OR crlp( )
AND crlp(0)
FACT bl(Wertl)
CALC Wertl < Abbruch
CALC Werto >= 1
FACT ddb_cwrl(0)
FACT cancel(Abbruch)
FACT vc_chtlack(2)
FACT dm_ccll(1)
FACT bo(Werto)
FACT br(Wertr)
FACT ddb_cwrl(2)
CALC Werto < Abbruch
CALC Wertr >= 1
AND ccll(1)
OR ccll( )
AND ccll(1)
AND ccll(0)
FACT br(Wertr)
CALC Wertr < Abbruch
CALC Werto >= 1
FACT ddb_cwrl(0)
FACT cancel(Abbruch)
FACT vc_chtlack(2)
FACT dm_ccll(1)
FACT bo(Werto)
FACT bl(Wertl)
FACT ddb_cwrl(1)
CALC Werto < Abbruch
CALC Wertl >= 1
AND ccrl(1)
OR ccrl( )
AND ccrl(1)
AND ccrl(0)

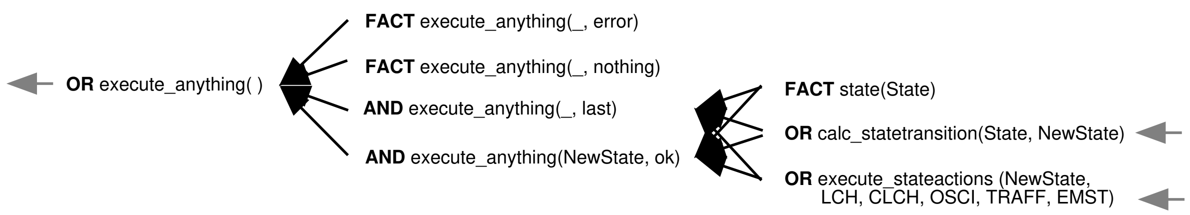
Appendix E

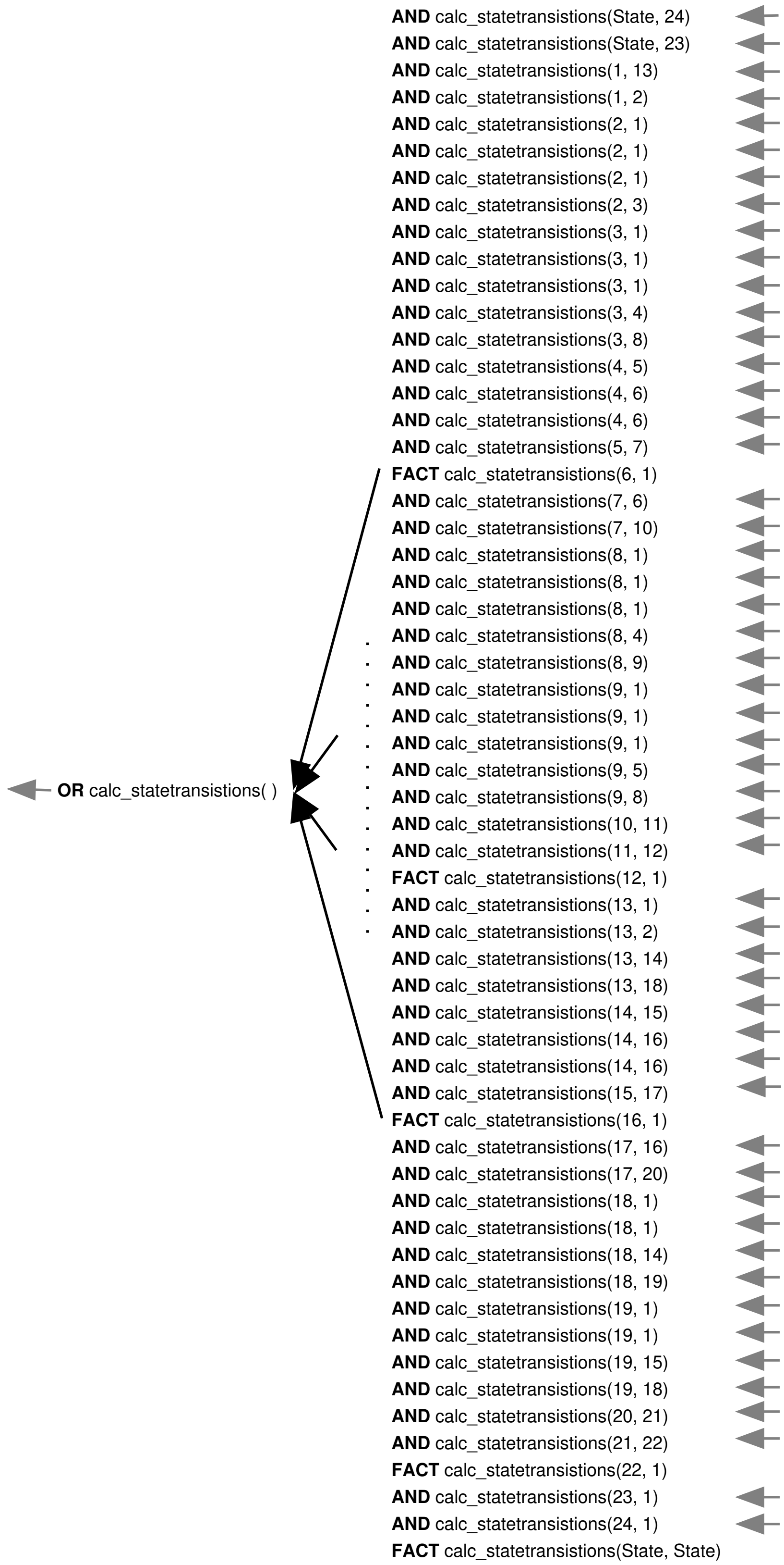
This Appendix contains the corresponding dataflow graph fragments are schown. Again, the graphs are drawn from left to right instead of top down.

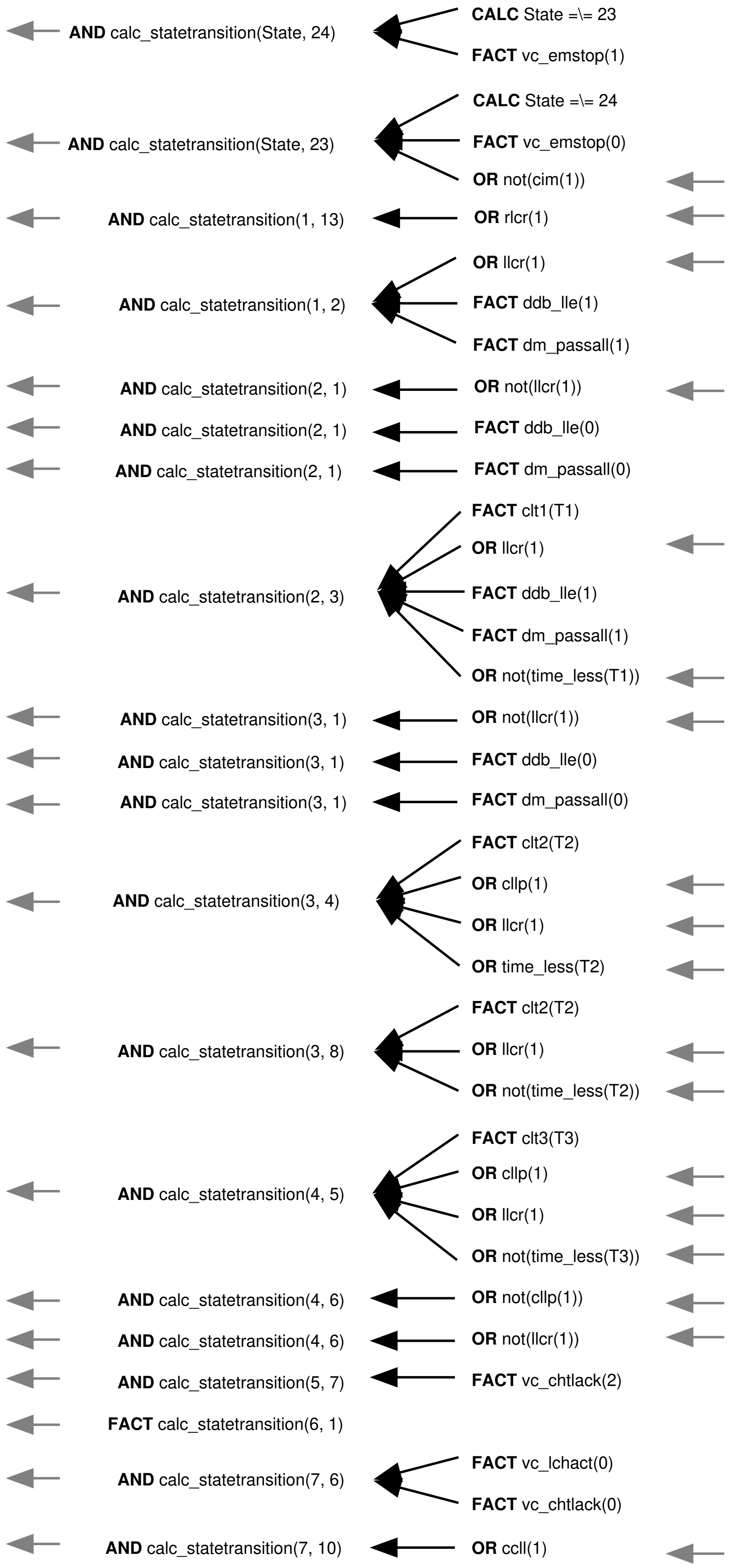




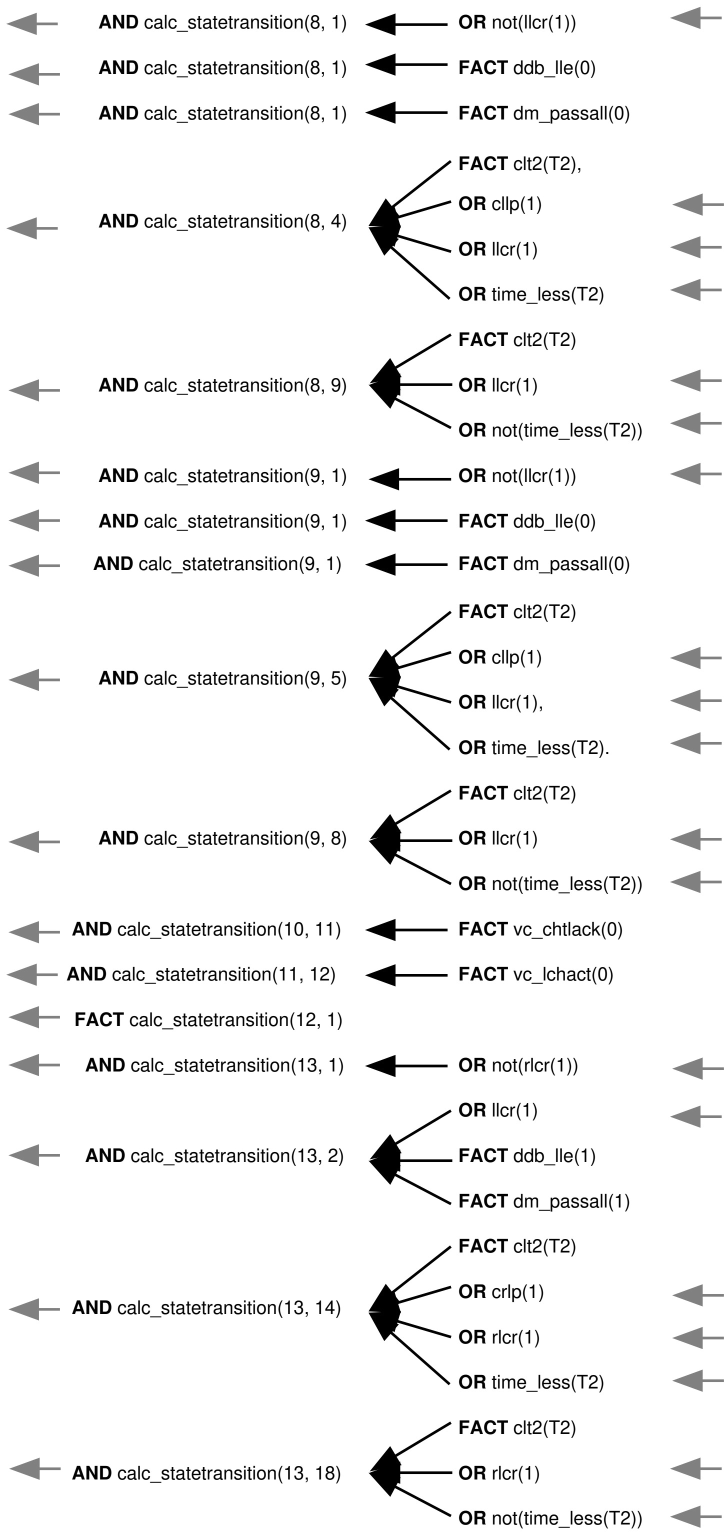

OR execute_stateactions( )
FACT execute_stateactions(1, 0, 0, 0, 0, 0)
FACT execute_stateactions(2, 0, 0, 0, 0, 0)
FACT execute_stateactions(3, 0, 0, 0, 0, 0)
FACT execute_stateactions(4, 0, 0, 0, 2, 0)
FACT execute_stateactions(5, 2, 0, 0, 2, 0)
FACT execute_stateactions(6, 0, 0, 0, 0, 0)
FACT execute_stateactions(7, 0, 0, 0, 2, 0)
FACT execute_stateactions(8, 0, 0, 0, 2, 0)
FACT execute_stateactions(9, 0, 0, 1, 2, 0)
FACT execute_stateactions(10, 0, 1, 0, 1, 0)
FACT execute_stateactions(11, 0, 0, 0, 1, 0)
FACT execute_stateactions(12, 0, 0, 0, 0, 0)
FACT execute_stateactions(13, 0, 0, 0, 0, 0)
FACT execute_stateactions(14, 0, 0, 0, 1, 0)
FACT execute_stateactions(15, 1, 0, 0, 1, 0)
FACT execute_stateactions(16, 0, 0, 0, 0, 0)
FACT execute_stateactions(17, 0, 0, 0, 1, 0)
FACT execute_stateactions(18, 0, 0, 0, 1, 0)
FACT execute_stateactions(19, 0, 0, 1, 1, 0)
FACT execute_stateactions(20, 0, 1, 0, 2, 0)
FACT execute_stateactions(21, 0, 0, 0, 2, 0)
FACT execute_stateactions(22, 0, 0, 0, 0, 0)
FACT execute_stateactions(23, 0, 0, 0, 0, 0)
FACT execute_stateactions(24, 0, 0, 0, 3, 1)

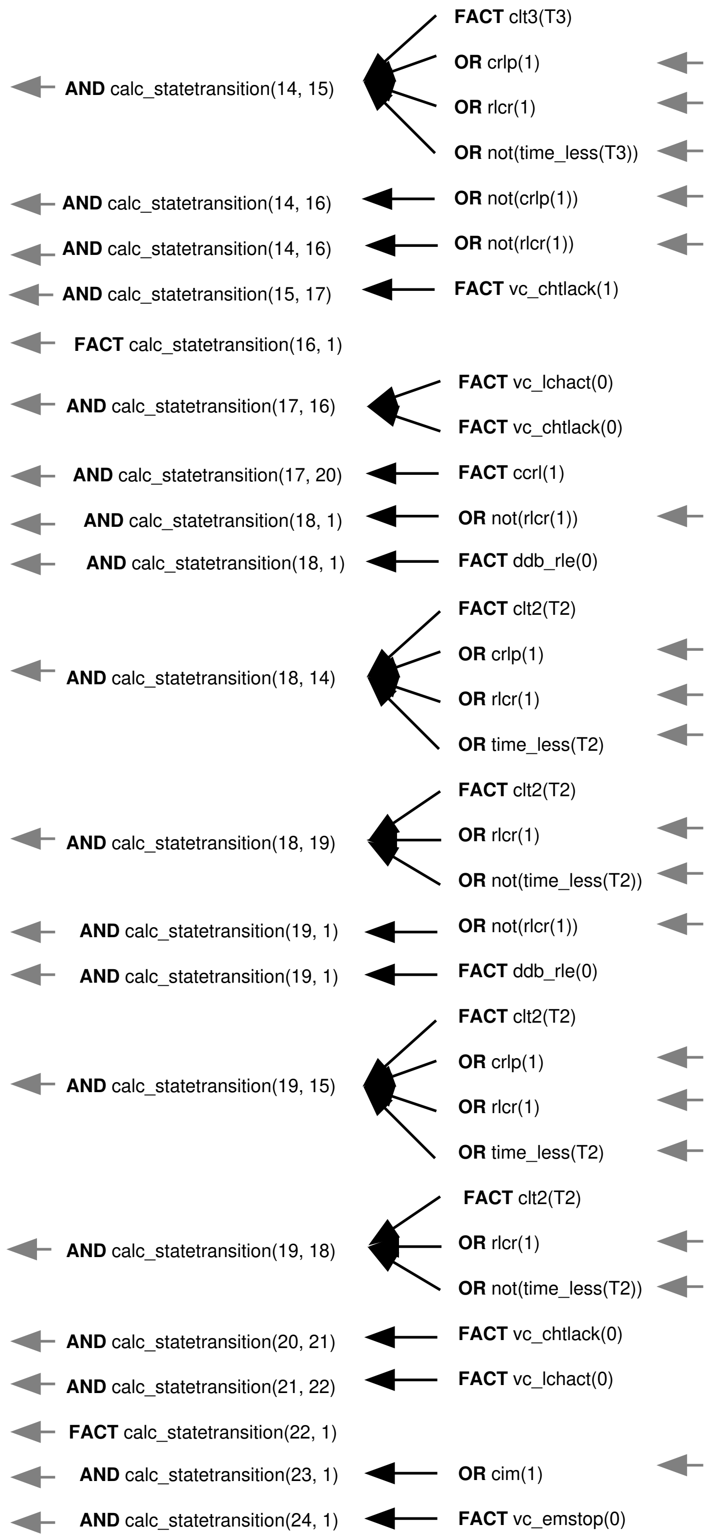

OR execute_anything( )
FACT execute_anything(_, error)
FACT execute_anything(_, nothing)
AND execute_anything(_, last)
AND execute_anything(NewState, ok)
FACT state(State)
OR calc_statetransition(State, NewState)
OR execute_stateactions (NewState, LCH, CLCH, OSCI, TRAFF, EMST)

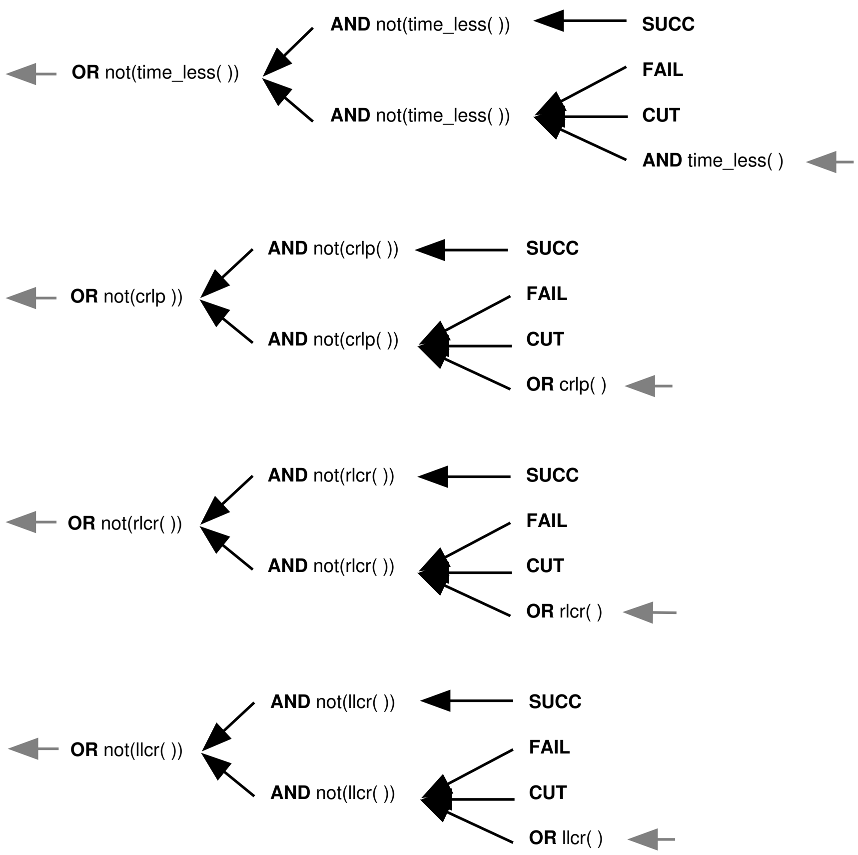

OR calc_statetransistions( )
AND calc_statetransistions(State, 24)
AND calc_statetransistions(State, 23)
AND calc_statetransistions(1, 13)
AND calc_statetransistions(1, 2)
AND calc_statetransistions(2, 1)
AND calc_statetransistions(2, 1)
AND calc_statetransistions(2, 1)
AND calc_statetransistions(2, 3)
AND calc_statetransistions(3, 1)
AND calc_statetransistions(3, 1)
AND calc_statetransistions(3, 1)
AND calc_statetransistions(3, 4)
AND calc_statetransistions(3, 8)
AND calc_statetransistions(4, 5)
AND calc_statetransistions(4, 6)
AND calc_statetransistions(4, 6)
AND calc_statetransistions(5, 7)
FACT calc_statetransistions(6, 1)
AND calc_statetransistions(7, 6)
AND calc_statetransistions(7, 10)
AND calc_statetransistions(8, 1)
AND calc_statetransistions(8, 1)
AND calc_statetransistions(8, 1)
AND calc_statetransistions(8, 4)
AND calc_statetransistions(8, 9)
AND calc_statetransistions(9, 1)
AND calc_statetransistions(9, 1)
AND calc_statetransistions(9, 1)
AND calc_statetransistions(9, 5)
AND calc_statetransistions(9, 8)
AND calc_statetransistions(10, 11)
AND calc_statetransistions(11, 12)
FACT calc_statetransistions(12, 1)
AND calc_statetransistions(13, 1)
AND calc_statetransistions(13, 2)
AND calc_statetransistions(13, 14)
AND calc_statetransistions(13, 18)
AND calc_statetransistions(14, 15)
AND calc_statetransistions(14, 16)
AND calc_statetransistions(14, 16)
AND calc_statetransistions(15, 17)
FACT calc_statetransistions(16, 1)
AND calc_statetransistions(17, 16)
AND calc_statetransistions(17, 20)
AND calc_statetransistions(18, 1)
AND calc_statetransistions(18, 1)
AND calc_statetransistions(18, 14)
AND calc_statetransistions(18, 19)
AND calc_statetransistions(19, 1)
AND calc_statetransistions(19, 1)
AND calc_statetransistions(19, 15)
AND calc_statetransistions(19, 18)
AND calc_statetransistions(20, 21)
AND calc_statetransistions(21, 22)
FACT calc_statetransistions(22, 1)
AND calc_statetransistions(23, 1)
AND calc_statetransistions(24, 1)
FACT calc_statetransistions(State, State)

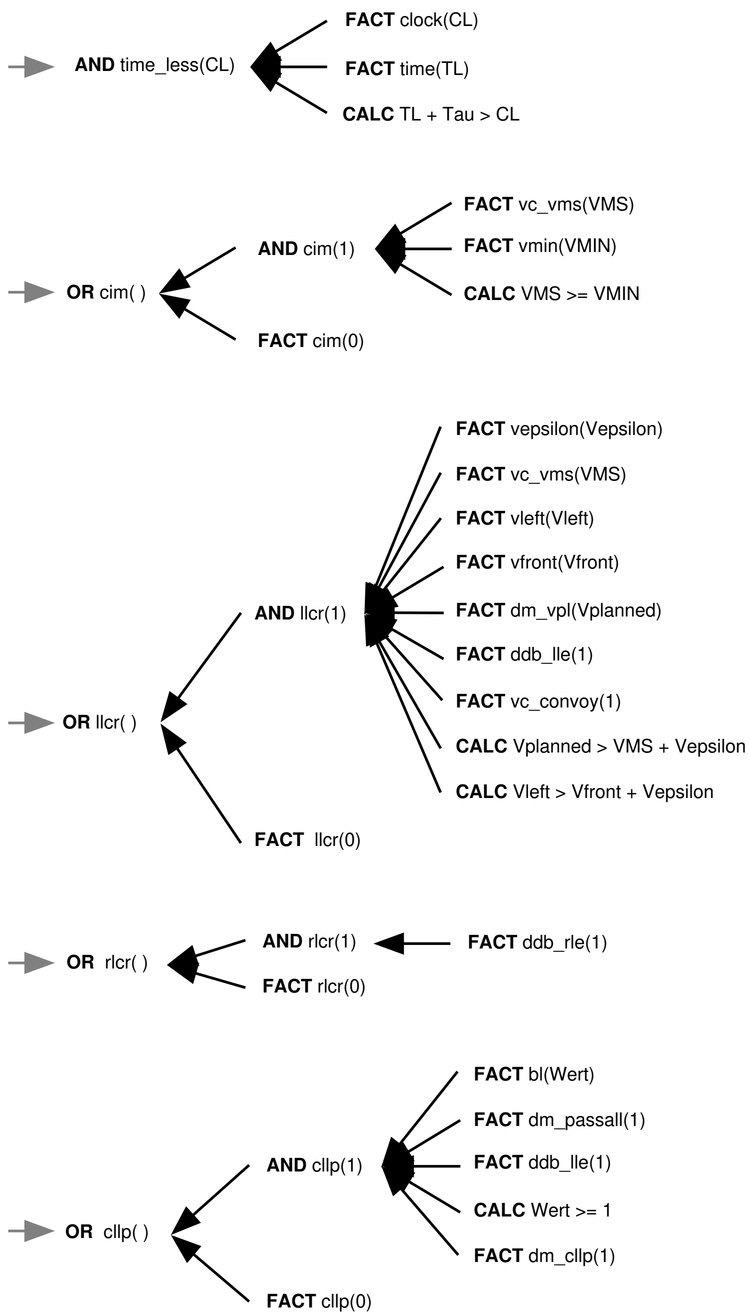


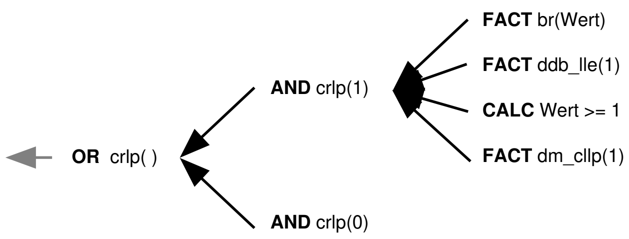
CALC State =\= 23
AND calc_statetransition(State, 24)
FACT vc_emstop(1)
CALC State =\= 24
AND calc_statetransition(State, 23)
FACT vc_emstop(0)
OR not(cim(1))
AND calc_statetransition(1, 13)
OR rlcr(1)
OR llcr(1)
AND calc_statetransition(1, 2)
FACT ddb_lle(1)
FACT dm_passall(1)
AND calc_statetransition(2, 1)
OR not(llcr(1))
AND calc_statetransition(2, 1)
FACT ddb_lle(0)
AND calc_statetransition(2, 1)
FACT dm_passall(0)
FACT clt1(T1)
OR llcr(1)
AND calc_statetransition(2, 3)
FACT ddb_lle(1)
FACT dm_passall(1)
OR not(time_less(T1))
AND calc_statetransition(3, 1)
OR not(llcr(1))
AND calc_statetransition(3, 1)
FACT ddb_lle(0)
AND calc_statetransition(3, 1)
FACT dm_passall(0)
FACT clt2(T2)
OR cllp(1)
AND calc_statetransition(3, 4)
OR llcr(1)
OR time_less(T2)
FACT clt2(T2)
AND calc_statetransition(3, 8)
OR llcr(1)
OR not(time_less(T2))
FACT clt3(T3)
OR cllp(1)
AND calc_statetransition(4, 5)
OR llcr(1)
OR not(time_less(T3))
AND calc_statetransition(4, 6)
OR not(cllp(1))
AND calc_statetransition(4, 6)
OR not(llcr(1))
AND calc_statetransition(5, 7)
FACT vc_chtlack(2)
FACT calc_statetransition(6, 1)
FACT vc_lchact(0)
AND calc_statetransition(7, 6)
FACT vc_chtlack(0)
AND calc_statetransition(7, 10)
OR ccll(1)

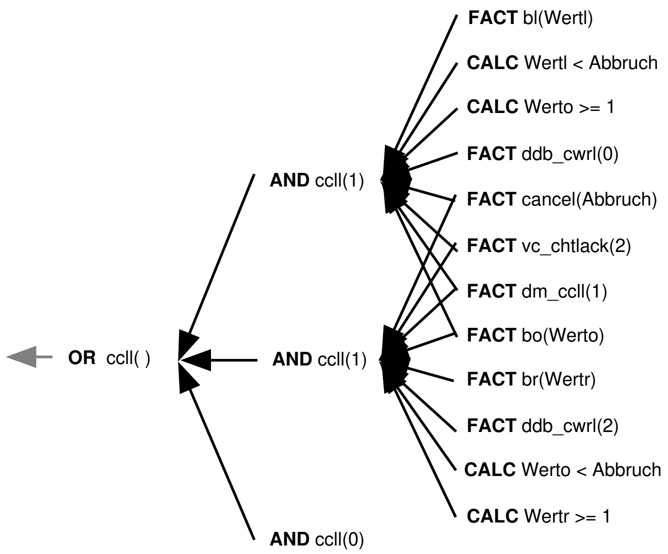

AND calc_statetransition(8, 1)
OR not(llcr(1))
AND calc_statetransition(8, 1)
FACT ddb_lle(0)
AND calc_statetransition(8, 1)
FACT dm_passall(0)
AND calc_statetransition(8, 4)
FACT clt2(T2),
OR cllp(1)
OR llcr(1)
OR time_less(T2)
AND calc_statetransition(8, 9)
FACT clt2(T2)
OR llcr(1)
OR not(time_less(T2))
AND calc_statetransition(9, 1)
OR not(llcr(1))
AND calc_statetransition(9, 1)
FACT ddb_lle(0)
AND calc_statetransition(9, 1)
FACT dm_passall(0)
AND calc_statetransition(9, 5)
FACT clt2(T2)
OR cllp(1)
OR llcr(1),
OR time_less(T2).
AND calc_statetransition(9, 8)
FACT clt2(T2)
OR llcr(1)
OR not(time_less(T2))
AND calc_statetransition(10, 11)
FACT vc_chtlack(0)
AND calc_statetransition(11, 12)
FACT vc_lchact(0)
FACT calc_statetransition(12, 1)
AND calc_statetransition(13, 1)
OR not(rlcr(1))
AND calc_statetransition(13, 2)
OR llcr(1)
FACT ddb_lle(1)
FACT dm_passall(1)
AND calc_statetransition(13, 14)
FACT clt2(T2)
OR crlp(1)
OR rlcr(1)
OR time_less(T2)
AND calc_statetransition(13, 18)
FACT clt2(T2)
OR rlcr(1)
OR not(time_less(T2))

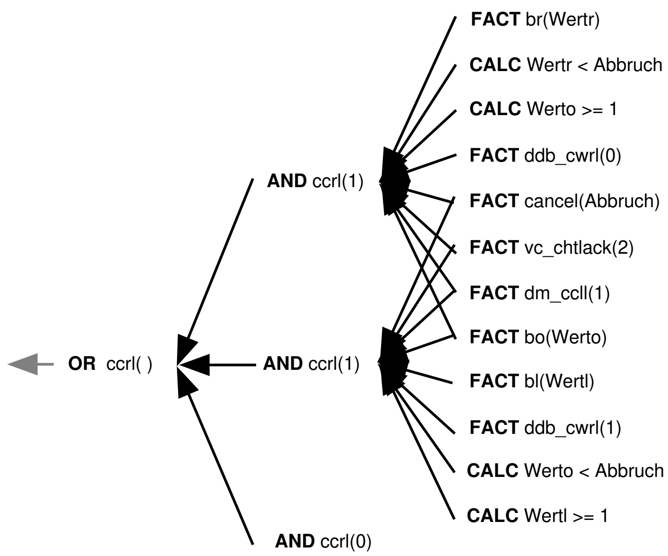

FACT clt3(T3)
OR crlp(1)
AND calc_statetransition(14, 15)
OR rlcr(1)
OR not(time_less(T3))
AND calc_statetransition(14, 16)
OR not(crlp(1))
AND calc_statetransition(14, 16)
OR not(rlcr(1))
AND calc_statetransition(15, 17)
FACT vc_chtlack(1)
FACT calc_statetransition(16, 1)
FACT vc_lchact(0)
AND calc_statetransition(17, 16)
FACT vc_chtlack(0)
AND calc_statetransition(17, 20)
FACT ccrl(1)
AND calc_statetransition(18, 1)
OR not(rlcr(1))
AND calc_statetransition(18, 1)
FACT ddb_rle(0)
FACT clt2(T2)
OR crlp(1)
AND calc_statetransition(18, 14)
OR rlcr(1)
OR time_less(T2)
FACT clt2(T2)
OR rlcr(1)
AND calc_statetransition(18, 19)
OR not(time_less(T2))
AND calc_statetransition(19, 1)
OR not(rlcr(1))
AND calc_statetransition(19, 1)
FACT ddb_rle(0)
FACT clt2(T2)
OR crlp(1)
AND calc_statetransition(19, 15)
OR rlcr(1)
OR time_less(T2)
FACT clt2(T2)
OR rlcr(1)
AND calc_statetransition(19, 18)
OR not(time_less(T2))
AND calc_statetransition(20, 21)
FACT vc_chtlack(0)
AND calc_statetransition(21, 22)
FACT vc_lchact(0)
FACT calc_statetransition(22, 1)
AND calc_statetransition(23, 1)
OR cim(1)
AND calc_statetransition(24, 1)
FACT vc_emstop(0)


AND not(time_less( ))
SUCC
OR not(time_less( ))
FAIL
AND not(time_less( ))
CUT
AND time_less( )
AND not(crlp( ))
SUCC
OR not(crlp))
FAIL
AND not(crlp( ))
CUT
OR crlp( )
AND not(rlcr( ))
SUCC
OR not(rlcr( ))
FAIL
AND not(rlcr( ))
CUT
OR rlcr( )
AND not(llcr( ))
SUCC
OR not(llcr( ))
FAIL
AND not(llcr( ))
CUT
OR llcr( )


AND time_less(CL)
FACT clock(CL)
FACT time(TL)
CALC TL + Tau > CL
OR cim( )
AND cim(1)
FACT vc_vms(VMS)
FACT vmin(VMIN)
CALC VMS >= VMIN
FACT cim(0)
OR llcr( )
AND llcr(1)
FACT vepsilon(Vepsilon)
FACT vc_vms(VMS)
FACT vleft(Vleft)
FACT vfront(Vfront)
FACT dm_vpl(Vplanned)
FACT ddb_lle(1)
FACT vc_convoy(1)
CALC Vplanned > VMS + Vepsilon
CALC Vleft > Vfront + Vepsilon
FACT llcr(0)
OR rlcr( )
AND rlcr(1)
FACT ddb_rle(1)
FACT rlcr(0)
OR cllp( )
AND cllp(1)
FACT bl(Wert)
FACT dm_passall(1)
FACT ddb_lle(1)
CALC Wert >= 1
FACT dm_cllp(1)
FACT cllp(0)


OR crlp( )
AND crlp(1)
AND crlp(0)
FACT br(Wert)
FACT ddb_lle(1)
CALC Wert >= 1
FACT dm_cllp(1)


OR ccll( )
AND ccll(1)
AND ccll(1)
AND ccll(0)
FACT bl(Wertl)
CALC Wertl < Abbruch
CALC Werto >= 1
FACT ddb_cwrl(0)
FACT cancel(Abbruch)
FACT vc_chtlack(2)
FACT dm_ccll(1)
FACT bo(Werto)
FACT br(Wertr)
FACT ddb_cwrl(2)
CALC Werto < Abbruch
CALC Wertr >= 1


OR ccrl( )
AND ccrl(1)
AND ccrl(1)
AND ccrl(0)
FACT br(Wertr)
CALC Wertr < Abbruch
CALC Werto >= 1
FACT ddb_cwrl(0)
FACT cancel(Abbruch)
FACT vc_chtlack(2)
FACT dm_ccll(1)
FACT bo(Werto)
FACT bl(Wertl)
FACT ddb_cwrl(1)
CALC Werto < Abbruch
CALC Wertl >= 1

References

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

[2] D. E. Altenkrüger, *Wissensdarstellung für Expertensysteme* B.I. Wissenschaftsverlag, 1987

[3] Arvind, K. P. Gostelow, "Dataflow Computer Architecture: Research and Goals", *UCI-ICS Report 113*, University of California, Irvine, ICS, 1978

[4] Arvind, V. Kathail, "A Multiple Processor Data Flow Machine that Supports Generalized Procedures", *Proc. of the 18th Int. Symp. on Computer Architecture*, Minneapolis, 1981, pp. 291-302

[5] Arvind, D. E. Culler, "The Tagged Token Dataflow Architecture", *Internal Report*, Massachusetts Institute of Technology, 1983

[6] L. Bic, "A Data-Driven Model for Parallel Interpretation of Logic Programs", *Proc. of the Int. Conference on fifth generation Computer Systems*, 1984, pp. 517-523

[7] L. Bic, C. Lee, "A Data-Driven Model for a Subset of Logic Programming", *ACM Transactions on Programming Languages and Systems*, Vol. 9, 1987, pp. 618-645

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

[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. B. Dennis, D. P. Misunas, "A Preliminary Architecture for a Basic Data-Flow Processor", *2nd Annual Symp. on Computer Architecture*,

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

[14] J. R. Gurd, C. C. Kirkham, I. Watson, "The Manchester Prototype Dataflow Computer", *Communications of the ACM*, Vol. 28, No. 1, 1980,

[15] W. Hahn, K. Fischer, "Music - an Event-Flow Computer for fast Digital Design Simulation", *22nd IEEE/ACM Design Automation Conference*, 1985, pp. 338-344

[16] W. Hahn, A. Hagerer, H. Anger, "Update-Dataflow Computing: A Way to Supercomputing in Discrete Simulation", *Volume I of Software Architecture*, Elsevier Science Publishers B. V. (North-Holland), 1992, pp. 509-517

[17] Z. Halim, I. Watson, "An OR-Parallel Data-driven Model for Logic Programs", *Proc. of International Workshop on High-Level Computer Architecture*, 1984, pp. 26-36

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

[19] K. Hiraki, S. Skiguchi, T. Shimada, "Status Report of SIGMA-1 Dataflow Supercomputer", *Advanced Topics in Dataflow Computing*, J.-L. Gaudiot, L. Bic, Prentice-Hall, 1991

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

[21] P. Kascuk, *Execution Models of Prolog for Parallel Computers*, MIT Press, Cambridge, Massachusetts, 1990

[22] P. Kascuk, "A Parallel Prolog Abstract Machine and its Multi-Transputer Implementation", *The Computer Journal*, Vol. 34, No. 1, 1991, pp 52-63

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

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

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

[26] G. M. Papadopoulos, K. R. Traub, "Multithreading: A Revisionist View of Dataflow Architectures", *Proc. of the 18th Int. Symp. on Computer Architecture*, Toronto, 1991

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

[28] T. Shimada, K. Hiraki, S. Sekiguchi, K. Nishada, "Evaluation of a Single Processor of a Prototype Data Flow Computer SIGMA-1 for Sientific Computations", *Proc. of the 13th Int. Symp. on Computer Architecture*, 1986, pp. 226-234

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

[30] 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

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

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

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

[34] M. J. Wise, *Prolog Multiprocessors*, Prentice-Hall, 1986