A New Addressing Scheme for Discrimination Networks easing Development and Testing

Karl-Heinz Krempels, Fabian Ohler and Christoph Terwelp
Informatik 5, Information Systems and Databases, RWTH Aachen University, Aachen, Germany

Keywords: Rule-based System, Discrimination Network.

Abstract: Rule Based Systems and Databased Management Systems are important tools for data storage and processing. Discrimination Networks (DNs) are an efficient way of matching conditions on data. DNs are based on the paradigm of dynamic programming and save intermediate computing results in network nodes. Therefore, an efficient scheme for addressing the data in the used memories is required.
Currently used schemes are efficient but sophisticated in operation hindering the development of new approaches for structural and functional optimization of DNs. We introduce and discuss a new addressing scheme for fact referencing in DNs with aim to ease the development of optimization approaches for DNs. The scheme uses fact addresses computed from sets of edges between the nodes in a DN to reference data.

1 INTRODUCTION

Because of their ability to store, access, and process large amounts of data Database Management Systems (DBMSs) and Rule-based Systems (RBSs) are used in many information systems as information processing unit (Brownston et al., 1985) (Forgy, 1981). A basic function of a RBS and a function of many DBMSs is to match conditions on the available data. Checking all data again every time some data changes performs badly. It is possible to improve the performance by saving intermediate results in memory using the dynamic programming paradigm. Such condition matchers are often based on Discrimination Networks (DNs). Many DN optimization approaches are discussed in (Forgy, 1982), (Miranker, 1987), and (Hanson, 1993). Their implementation and evaluation is often hindered by the complexity of current DN implementations (Jamocha Team, 2013) (CLIPS Project Team, 2013) (JBoss Drools Team, 2013). To ease the implementation of new optimization approaches, we introduce a new method to address information inside a DN.

This paper is organized as follows: Section 2 introduces DNs and Section 3 discusses the state of the art for addressing facts in DNs. Section 4 gives a short description of the problem which the approach in this paper solves. Section 5 presents the new approach to address facts in an easier way. Section 6 describes an algorithm using the new address scheme in detail. In Section 7 the mode of operation of the algorithm is shown by an example run. Finally, Section 8 shows our plans to improve and implement the presented approach.

2 DISCRIMINATION NETWORKS

A DN consists of a set of nodes for conditional tests or join operations for facts. The objective of a DN is the verification of rule conditions with a minimum of test executions.

A DN represents the condition of every rule from the rule base by a hierarchic network of interconnected nodes. So the terminal node of every hierarchic network is visited only by facts or fact tuples that fulfill the corresponding condition. A rule's condition is divided into its atomic tests which are executed by network nodes. The network is constructed with respect to the syntax of the condition.

Every node has a test set, a memory, one or more inputs, and an output connection. The memory keeps a set of fact tuples received by the node through its input connection that have passed the test set. The output connection is used by successor nodes to access the node's memory and receive notifications about memory changes. In this way the network saves intermediate processing results and in this meets the dynamic programming requirements. Common DNs

DOI: 10.5220/0004642201180124
In *Proceedings of the 2nd International Conference on Data Technologies and Applications* (DATA-2013), pages 118-124
ISBN: 978-989-8565-67-9

consist of four different node types as shown below in Fig. 1:

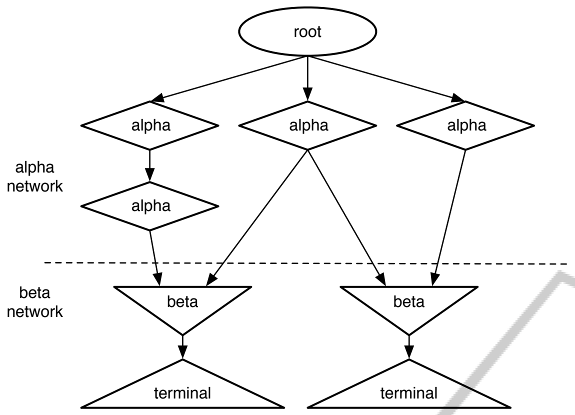


Figure 1: DN example.

Root Node. is a virtual unique node. Its input connection is connected to the working memory and receives all changes of facts. As a virtual node it is missing a memory because it would be equivalent to the working memory. The output connection provides access to a tuple based representation of the working memory elements (facts). The root node has an empty test set.

Alpha Node. has one input connection, where it receives tuples containing just one fact. The input is connected to another alpha node or to the root node. It checks if the tuples pass the test set and relays them to the output. It may have a memory or be virtual, acting as a simple filter.

Beta Node. has more than one input connection. It joins the tuples from the input connections and checks if the resulting tuple passes the test set. So it is required not only to react to incoming notifications but to request memory contents from its predecessors. The input connections can be connected to alpha or beta nodes. It saves all tuples which passed the test set in its memory and sends notifications about memory changes through its output connection. Beta nodes may have an empty test set to enable full joins.

Terminal Node. is a virtual node with one input connection, which is connected to a beta or alpha node. Its purpose is to collect the fact tuples which fulfill the whole rule condition. So it has an empty test set and its output is connected to the conflict set, to fill it.

Based on these node types the network consist of two parts. The alpha network containing the root node and all alpha nodes, and the beta network containing beta and terminal nodes.

3 STATE OF THE ART

Beta nodes of a DN apply filters to joined fact tuples. So, there is a need to identify which facts in the tuples the corresponding filters should be applied to. A simple solution to address a specific fact in a fact tuple is to give the beta node inputs an order. Therefore, the order of the facts inside the tuple is defined and the facts can be addressed by their order position inside the tuple. The order of the facts depends on the order of the beta node's inputs and the orders of the inputs of all preceding beta nodes. So, during the construction of the network the correct position of a fact has to be calculated, which gets the more complicated the larger the beta network gets. Current RBS implementations use rule-compilers which do these calculations during network construction. There are no publications addressing this topic in detail since (Forgy, 1982) although improvements would ease development of optimization algorithms a lot.

4 OBJECTIVE

Optimizations of DN construction algorithms and DN runtime optimization require an efficient way to address facts in filter conditions easily. Additionally, internal optimizations of memory usage and tuple processing often require flexibility in addressing facts not limited by a predefined order of facts in tuples. So, the aim is to develop an efficient referencing method for facts which on the one hand is easy to use for network construction and optimization algorithms, and on the other hand allows a computation of references which can be used efficiently during the DN runtime.

5 APPROACH

Observing the fact flow in a DN we recognize that it is possible to address an element of a fact tuple in the DN by the set of edges traversed by the fact since it left the alpha network (see Fig. 2).

If we would limit the network structure of one rule condition to a tree we could identify a fact tuple element with only the edge between the alpha and beta network. Since the output of one alpha node may be used multiple times in a rule condition network, we have to use more than one edge to address elements which originate from the same alpha node. Although these additional edges are only required in this particular cases we use the full set of edges traversed by the fact to obtain an easy to use addressing scheme.

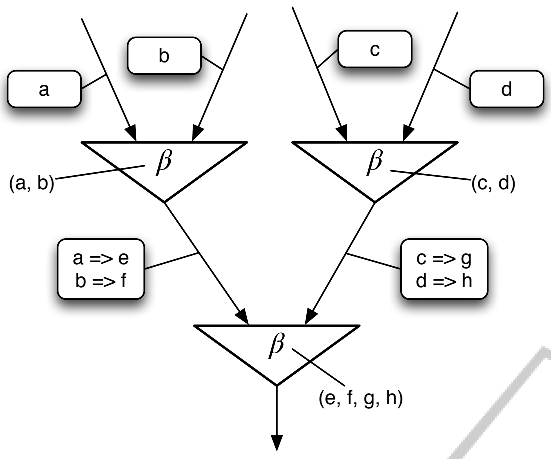
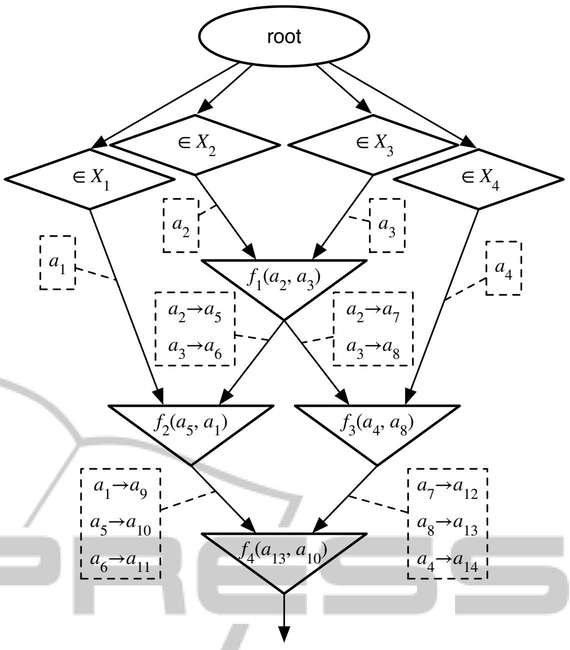


Figure 2: Address translation in the edges.

Because of the complexity of handling these sets of edges as addresses we reduce the sets to a unique identifier. Therewith, we simplify the comparison of addresses and the conversion of addresses into the positions of fact tuple elements. This is possible, because it is not necessary to determine which edges are in the set. The unique address identifiers are generated by constructing the corresponding edge set. The construction starts at the outputs of the alpha network. For edge sets consisting of only one edge, this edge must connect the alpha with the beta network. These edges are annotated with a unique address identifier. So, if the address for the edge set with only one edge is required, it can be acquired from the edge. For edge sets with more than one edge, we assign a map to every edge inside the beta network. It maps the addresses of all elements of the fact tuples of the source beta node (to whose output the edge is connected) to new addresses valid for fact tuples of the target beta node (to whose input it is connected). So, we retrieve the address of a tuple element by successive mapping of the addresses through all edges of the addressing set. That way we never have to compare sets but only identifiers. This satisfies the requirements during construction of the DN.

To process facts in the network some more information is required. On the one hand, the inputs of a beta node need to know their addresses to apply the node's filter to an incoming fact tuple. Because the input already has a mapping of addresses valid in the parent node to addresses valid in the child node, these information is available. On the other hand, a mapping of addresses to their corresponding input and the address in the parent node is required to be able to access the fact tuples in the parent node's memory, if a fact tuple has arrived on another input and a join has to be performed. To solve this problem, the parent's node memory and the corresponding address is attached to new addresses.



$$x_1 \in X_1 \wedge x_2, x_3 \in X_2 \wedge x_4, x_5 \in X_3 \wedge x_6 \in X_4$$
$$\wedge f_1(x_2, x_4) \wedge f_1(x_3, x_5) \wedge f_2(x_2, x_1) \wedge f_3(x_5, x_6) \wedge f_4(x_2, x_6)$$

Figure 3: DN, edges annotated with address mappings.

In the scope of object orientation we get a linked list. It starts with the address in the current node, which contains a reference to the parent nodes memory and an address which is valid in the parent nodes scope. This address again has a reference to its parent and so on until the alpha network is reached. Normally this list should only be used to access the memory of the direct parents. Address comparisons are possible by direct object reference comparison because address objects are unique.

The impact on the runtime of a DN would be linear at most, because only single addresses have to be accessed and no searches through the linked addresses are required. It could be reduced to no change to runtime behaviour by translating the addresses used during construction time into the previously used addressing scheme for runtime.

6 NETWORK CONSTRUCTION

In this Section the mode of operation of a DN construction algorithm is discussed. A DN is constructed based on rule conditions defined in a rule description language. The DNs derived from the same rule description may differ in the order of filter application

Algorithm 1: Function for creating addresses.

```
Input: The output $o$ for which the address is created.
    The address $a'$ which is mapped to this address
    by the output $o$.
Output: A new address $a$.
 1: function $new\_address(a', o)$
 2:     $a_{id} \leftarrow unique\ identifier$
 3:     $a_{output} \leftarrow o$
 4:     $a_{prev} \leftarrow a'$
 5:     return $a$
 6: end function
```

and optimization of the DN structure. As this is out of scope of this paper, an ordered list of the filters of a rule condition is expected as input for the construction of a DN. So, the order the filters are applied is defined in a preceding optimization phase and is not part of this construction algorithm.

Algorithm 2: Function for creating nodes.

```
Input: The variables $(v_1, ..., v_m)$ on which the filter
    $f$ should be applied. Current variable to address
    mapping $m$. The set $G$ of the tuples in which the
    variables are available.
Output: A new node $n$.
 1: function $new\_node((f, (v_1, ..., v_m)), m, G)$
 2:     $n_{outputs} \leftarrow \{\}$
 3:     $n_{inputs} \leftarrow \{\}$
 4:     for all $G' = \{v'_1, ..., v'_{m'}\} \in G$,
 5:                 with $\exists i \in \{1, ..., m\} : v_i \in G'$ do
 6:         $o \leftarrow m(v'_1)_{output}$
 7:         if $o_{connection} = undefined$ then
 8:             $o_{connection} \leftarrow n$
 9:             $n_{inputs} \leftarrow n_{inputs} \cup \{o\}$
10:         else
11:             $o' \leftarrow new\_output(o_{node})$
12:             $m \leftarrow$
                $\left( v \mapsto \begin{cases} o'_{translation}(m(v)_{prev}) & \text{, if } v \in G' \\ m(v) & \text{, otherwise} \end{cases} \right)$
13:             $n_{inputs} \leftarrow n_{inputs} \cup \{o'\}$
14:         end if
15:     end for
16:     for all $i \in \{1, ...n\}$ do
17:         $a_i \leftarrow m(v_i)$
18:     end for
19:     $n_{filter} \leftarrow (f, (a_1, ..., a_n))$
20:     return $(n, m)$
21: end function
```

The function given in Algorithm 1 creates a new address for an edge between two nodes using the current address of the fact a' and the edge o as parameters. The new address is assigned a unique identifier a_{id} and both parameters.

For a better understanding of the following algorithms we begin with introducing the variables used to keep the state of the network during construction. N is the set of nodes the DN is currently made of. m is a map, which maps the variables to their corresponding addresses they are currently available under. G keeps track of the variables which are combined in the tuples of the network.

Algorithm 3: Function for node outputs.

```
Input: The node $n$ for which a new output should be
    created.
Output: A new output $o$ of the node $n$.
 1: function $new\_output(n)$
 2:     $o_{node} \leftarrow n$
 3:     $n_{outputs} \leftarrow n_{outputs} \cup \{o\}$
 4:     $o_{address\_map} \leftarrow \{(a, new\_address(a, o)) \mid$
 5:                 $\exists n' \in n_{inputs} : (a', a) \in n'_{address\_map}\}$
 6:     $o_{translation} \leftarrow$
            $\left( a \mapsto \begin{cases} a' & \text{, if } (a, a') \in o_{address\_map} \\ undefined & \text{, otherwise} \end{cases} \right)$
 7:     $o_{connection} \leftarrow undefined$
 8:     return $o$
 9: end function
```

In Algorithm 2 a function is described which creates a new node in the DN. It requires the filter f and the variables $v_1, ..., v_m$ to which the filter should be applied. Additionally, the set G must be supplied to enable connecting the same node to two inputs of the new node. In the lines 4 to 15 all tuples containing variables the filter should be applied on are visited. In line 6 the output corresponding to the tuple is selected. Line 7 to 8 check if the output is already connected to a node. If it isn't it can be used as input for the newly created node. Otherwise in line 11 to 13 a new output is created and the address map is updated to match the new addresses of the variables of the tuple. This is especially required if a node should be connected to the new node more than once. The creation of the initial output of the node and the corresponding change of the set G and mapping m is not handled in this function because it is also required in the case of re-usage of an already existing node.

The function given in Algorithm 3 creates a new output for an existing node. The only parameter is the node n for which the output should be created. The output is assigned the source node and is added to the node's output set. An address map for the output is generated mapping all addresses of inputs of the source node to newly created addresses. Additionally a translation function is created which maps the addresses for easier use.

Algorithm 4: Function for calculating all possible addresses for variables in a filter.

Input: Variables $\{v_1,...,v_m\}$ of a filter to generate possible addresses for. Current variable-to-address mapping m. Current variable grouping per output G.

Output: Set of address mappings *return_mappings* which are relevant for searching a reusable node.

```
 1: function $possible\_addresses(\{v_1,...,v_m\}, m, G)$
 2:   $I \leftarrow \{1,...,m\}$
 3:   $V \leftarrow \{v_i | i \in I\}$
 4:   $V' \in G$, with $v_1 \in G'$
 5:   $I' \leftarrow \{i \in I | v_i \in G'\}$
 6:   $A' \leftarrow \{m(v_i) | i \in I'\}$
 7:   $O \leftarrow \{o | \exists a \in A' : o \in ((a_{output})_{node})_{outputs}\}$
 8:   $mappings \leftarrow \{m\}$
 9:   for all $o \in O \setminus m(v_1)_{output}$ do
10:     $new\_mapping \leftarrow \left( v \mapsto \begin{cases} o_{translation}(m(v)_{prev}) & \text{, if } v \in V' \\ m(v) & \text{, otherwise} \end{cases} \right)$
11:     $mappings \leftarrow mappings \cup \{new\_mapping\}$
12:   end for
13:   $return\_mappings \leftarrow \{\}$
14:   for all $a \in mappings, a' \in$
15:           $possible\_addresses(V - V', m, G)$ do
16:     $new\_mapping \leftarrow \left( v \mapsto \begin{cases} a(v) & \text{, if } v \in V' \\ a'(v) & \text{, otherwise} \end{cases} \right)$
17:     if $|\{new\_mapping(v) | v \in V\}| = |V|$ then
18:       $return\_mappings \leftarrow$
            $return\_mappings \cup \{new\_mapping\}$
19:     end if
20:   end for
21:   return $return\_addresses$
22: end function
```

Algorithm 4 describes a function to create a set of mappings of a given set of variables onto all possible addresses. This is required because in some cases variables are mapped to addresses of the wrong output of a node preventing node sharing. Without this function it would be possible to generate a valid network for every rule condition, but some constructs would not be possible, as we will see in the example below. This function widens the search space for nodes for node sharing and works in a recursive manner. In lines 9 to 12 it searches for all alternative outputs for the first variable tuple in G and creates an address mapping for the variables in the tuple for each output. In lines 13 to 20 each of these mappings is combined with each of the recursively generated mappings for all other variable tuples. Lines 17 to 19 are required to make sure that different variable tuples are not mapped to the same addresses.

Algorithm 5: Network construction algorithm, supporting node sharing.

Input: The list F of filters with their variables on which they should be applied in the order in which they should be implemented in the network. A function m mapping variable identifiers to addresses of edges from the alpha network.

Output: A discrimination network with the nodes in N.

```
 1: function $create\_network(F, m)$
 2:   $N \leftarrow \{\}$
 3:   $G \leftarrow \{\{v\} | \exists (f, V) \in F : v \in V\}$
 4:   for all $(f, (v_1,...,v_m)) \in F$ do
 5:     for all $possible\_mapping \in$
 6:         $possible\_addresses(\{v_1,...,v_m\}, m, G)$ do
 7:       for all $i \in \{1,...,m\}$ do
 8:         $a_i \leftarrow possible\_mapping(v_i)$
 9:       end for
10:       if $\exists n' \in N : n'_{filter} = (f, (a_1,...,a_m))$ then
11:         $n \leftarrow n'$
12:         $m \leftarrow possible\_mapping$
13:         $o$, with $o_{node} = n'$
14:         Break for loop.
15:       end if
16:     end for
17:     if $!isset(n)$ then
18:       $(n, m) \leftarrow$
            $new\_node((f, (a_1,...,a_n)))$
19:       $o \leftarrow new\_output(n)$
20:     end if
21:     $G' \leftarrow \{X \in G | \exists i \in \{1,...,m\} : v_i \in X\}$
22:     $m \leftarrow \left( v \mapsto \begin{cases} o_{translation}(m(v)) & \text{, if } v \in \bigcup_{g \in G'} g \\ m(v) & \text{, otherwise} \end{cases} \right)$
23:     $G \leftarrow (G - G') \cup \{\bigcup_{g \in G'} g\}$
24:   end for
25:   return $N$
26: end function
```

Finally, the function given in Algorithm 5 constructs the network using all functions discussed. Parameters to this function are the list of filters F in the order of implementation and a mapping m of all variables

Table 1: The mapping m after each filter is added to the network.

m()	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8	v_9
Step 0	a_2	a_3	a_2	a_3	a_2	a_3	a_1	a_4	a_4
Step 1	a_5	a_6							
Step 2			a_5	a_6					
Step 3					a_5	a_6			
Step 4	a_7	a_8					a_9		
Step 5			a_{12}	a_{13}				a_{14}	
Step 6					a_{12}	a_{13}			a_{14}

to the addresses of the outputs of the alpha network. Lines 2 and 3 are initializing the state of the DN. In lines 4 to 24 the filters of F are processed sequentially. Lines 5 to 16 search for existing nodes already implementing the filter. Using the function from Algorithm 4 all possible nodes for node sharing are found. If a node is found, it is used and the possible address mapping generated in Algorithm 4 becomes the current address mapping. In lines 17 to 20 a new node is created if no existing node to share was found. Lines 21 to 23 update the address mapping for all variables which are contained in the same tuples as the variables used by the filter. Set G is updated to match the tuples which are available at the new output.

7 EXAMPLE

An example run of the given algorithm is performed using the list of filter-variable tuples F and the mapping m given in Table 1, Step 0 as parameters.

$$\begin{aligned} F = (\quad & (f_1, v_1, v_2), \\ & (f_1, v_3, v_4), \\ & (f_1, v_5, v_6), \\ & (f_2, v_1, v_2, v_7), \\ & (f_2, v_3, v_4, v_8), \\ & (f_2, v_5, v_6, v_9) \quad) \end{aligned}$$

At the beginning of the network construction the set G is initialized as shown in Table 2, Step 0. We start with the construction of the network node for $(f_1, v_1, v_2) \in F$. The call of the *possible_addresses* function returns only the current mapping m because at this time every alpha node has only one output. Since there is no beta node in the current network, a new one is created. The output of the new node gets the mapping of a_2 to the new address a_5 and a_3 to a_6. The mapping m is updated, so it maps variables v_1 and v_2 to their new addresses in the output of the

Table 2: The values of the set G after each filter is added to the network.

	G
Step 0	$\{\{v_1\}, \{v_2\}, \{v_3\}, \{v_4\}, \{v_5\}, \{v_6\}, \{v_7\}, \{v_8\}, \{v_9\}\}$
Step 1	$\{\{v_1, v_2\}, \{v_3\}, \{v_4\}, \{v_5\}, \{v_6\}, \{v_7\}, \{v_8\}, \{v_9\}\}$
Step 2	$\{\{v_1, v_2\}, \{v_3, v_4\}, \{v_5\}, \{v_6\}, \{v_7\}, \{v_8\}, \{v_9\}\}$
Step 3	$\{\{v_1, v_2\}, \{v_3, v_4\}, \{v_5, v_6\}, \{v_7\}, \{v_8\}, \{v_9\}\}$
Step 4	$\{\{v_1, v_2, v_7\}, \{v_3, v_4\}, \{v_5, v_6\}, \{v_8\}, \{v_9\}\}$
Step 5	$\{\{v_1, v_2, v_7\}, \{v_3, v_4, v_8\}, \{v_5, v_6\}, \{v_9\}\}$
Step 6	$\{\{v_1, v_2, v_7\}, \{v_3, v_4, v_8\}, \{v_5, v_6, v_9\}\}$

newly created beta node (Table 1, Step 1). G is updated to show that v_1 and v_2 are joined and available in one tuple (Table 2, Step 1).

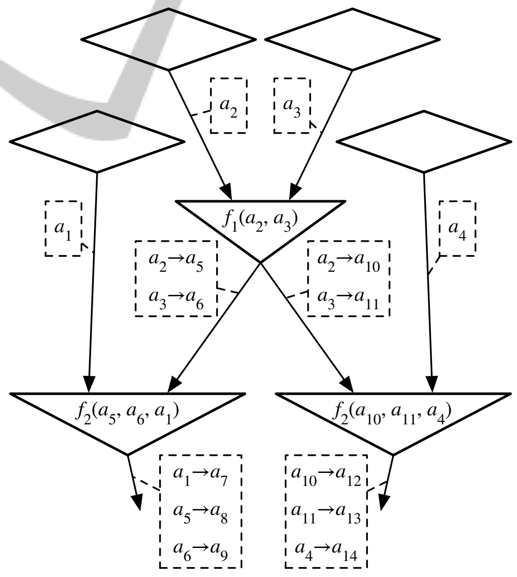


Figure 4: Example DN, constructed using the introduced algorithm.

The next tuple (f_1, v_3, v_4) is processed. Again the function *possible_addresses* returns only the current mapping m because all nodes only have one output. v_3 and v_4 are mapped by the mapping m to the addresses a_2 and a_3. The previously created beta node already provides this filter-address combination, so it is reused as a shared node. The mapping m is updated, so it maps variables v_3 and v_4 to their new addresses in the output of the reused beta node (Table 1, Step 2).

The set G is updated to show that v_3 and v_4 are joined and available in one tuple (Table 2, Step 2).

The next tuple (f_1, v_5, v_6) is processed in a similar way to the previous tuple (f_1, v_3, v_4). The mapping m and the set G are updated accordingly (Table 1 and Table 2, Step 3).

Next, the tuple (f_2, v_1, v_2, v_7) is processed. The function *possible_addresses* returns only the current mapping m because all nodes have only one output. The variables v_1, v_2, and v_7 are mapped to their corresponding addresses a_5, a_6, and a_1. Since no node with this filter-address combination exists, again a new network node is created. The new output of this node maps the existing addresses a_5, a_6, and a_1 to the new addresses a_7, a_8, and a_9. The mapping m is updated to map the variables to their new addresses (Table 1, Step 4). The set G is updated to show that v_1, v_2, and v_7 are available in one tuple now (Table 2, Step 4).

The next tuple (f_2, v_3, v_4, v_8) is processed. The function *possible_addresses* still returns only the current mapping m because all nodes have only one output. The variables v_3, v_4, and v_8 are mapped to their addresses a_5, a_6, and a_4. Since no node with this filter-address combination exists, a new network node is created. But because the output of the node created for the tuple (f_1, v_1, v_2) is already connected to another node, a new output is created for this node. This output maps the addresses a_2 and a_3 to the new addresses a_{10} and a_{11}. The newly created node is connected to this output. The mapping m and the set G are updated accordingly again (Table 1 and Table 2, Step 5).

The last tuple (f_2, v_5, v_6, v_9) is processed. The function *possible_addresses* returns now a set containing the current mapping m and a mapping identical to m except for the variables v_5 and v_6 which are mapped to the addresses a_{10} and a_{11}. For the first mapping of the variables to the addresses a_5, a_6 and a_4, no node implementing the filter-address combination is found. For the second mapping to the addresses a_{10}, a_{11} and a_4, the node created for (f_2, v_3, v_4, v_8) is found. This node is reused and the mapping m and the set G are updated accordingly again (Table 1 and Table 2, Step 6). The resulting DN is shown in Figure 4.

8 OUTLOOK

The described approach is currently in the process of implementation for the RBS Jamocha. It will be used to make fast prototyping of optimized construction algorithms and reconstruction algorithms possible. As an additional benefit of the clean and flexible interface it will be possible to exchange major parts of the DN (e.g. fact memory or join-optimization) with minimal code adaptation. Simplification of the algorithms especially the handling of shared network nodes is desirable and is objective of our future work.

REFERENCES

Brownston, L., Farrell, R., Kant, E., and Martin, N. (1985). *Programming expert systems in OPS5: an introduction to rule-based programming*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

CLIPS Project Team (2013). CLIPS Project Page. http://clipsrules.sourceforge.net/.

Forgy, C. L. (1981). OPS5 User's Manual. Technical report, Department of Computer Science, Carnegie-Mellon University.

Forgy, C. L. (1982). Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence*, 19(1):17 – 37.

Hanson, E. N. (1993). Gator: A Discrimination Network Structure for Active Database Rule Condition Matching. Technical report, University of Florida.

Jamocha Team (2013). Jamocha Project Page. http://www.jamocha.org, http://sourceforge.net/projects/jamocha.

JBoss Drools Team (2013). JBoss Drools Project Page. http://www.jboss.org/drools/.

Miranker, D. P. (1987). TREAT: A Better Match Algorithm for AI Production Systems; Long Version. Technical report, University of Texas at Austin, Austin, TX, USA.