Eindhoven University of Technology
Department of Mathematics and Computing Science

The Formal Model of a Pattern Browsing Technique

by

J. Hidders, C. Hoskens and J. Paredaens

95/30

ISSN 0926-4515

editors: prof.dr. J.C.M. Baeten
prof.dr. M. Rem

Computing Science Report 95/30
Eindhoven, October 1995

The Formal Model of a Pattern Browsing Technique

Jan Hidders Cora Hoskens Jan Paredaens

Abstract

In this report we introduce a general browsing model that describes intuitive ideas about browsing. In this model it is assumed that the database scheme, as well as the instance of the database are represented by graphs. The most important browsing step in this model is the pattern step. It is based upon finding subgraphs in the instance matching a pattern and supplied with a browsing condition that links it to previous steps. This allows the user to visually specify a browsing step based upon the results of previous steps. Other browsing steps and operators in the model allow the user to randomly select some subgraphs found by a step, replace an old browsing step with a new one or undo some of the last browsing steps. After presenting the model we compare its expressive power with that of the relational algebra.

1 Introduction

This paper introduces a browsing model. Browsing provides a means to investigate the contents of a database in a special way. It adds to querying the possibility to reuse former results. It is like moving around in the database by specifying intermediate results and using these to get more specific ones. In a sequence of steps the user tries to get closer to the information he wants to get.

What characterizes browsing, is that it is an interactive and iterative process of specifying queries and investigating the results of those queries in order to be able to state new ones. This is particularly useful when a user does not know exactly what he is looking for, or how to access the information he is looking for.

An example of a browsing facility is available in Smalltalk [8]. Smalltalk provides the possibility to wander around the class structure in order to find the particular class that the user is looking for. The classes are ordered in a tree structure and by means of browsing one can go from one class to one of its subclasses. Without this possibility it would be very hard to find the classes of interest.

Another way of browsing is available in hypertext documents [7]. In a hypertext, links are provided to other (parts of) documents which in some way or another are related to the current document. By choosing the right links, the user tries to find the document he is interested in. World Wide Web is a very nice example of a (world wide) hypertext [2]. Both examples show the characteristic of browsing: the use of intermediate results in order to get the required result.

This paper does not describe some new innovative browsing technique. It provides a general model to describe intuitive ideas about browsing [3], [4].

In this particular browsing model it is assumed that the database scheme, as well as the instances of the database are represented by graphs, such as in the GOOD model [1], [5], [6]. In this model, the nodes of the graph represent objects while the edges represent the properties of and the relationships between the objects.

A pattern is also a graph, and can be used to select subsets of an instance by the notion of pattern matching. This means that every subgraph of an instance that matches the pattern is selected. Such a subgraph is called an *embedding*.

With the pattern browsing technique, every possible action that the user can take is called a *browsing statement*. Browsing statements can be divided into *browsing steps* and *browsing operators*. There are two different steps in the model and two different operators. Actually, the steps are the elementary browsing statements. They provide the user with all necessary actions to enable him to browse. The operators, on the other hand, are meant to ease the task of the user. We first discuss the two possible browsing steps.

First of all, embeddings can be selected by specifying a pattern. As such it is a pattern matching step, or *pattern step* for short. The nodes in the pattern are labeled, and they can also be given a condition. Such a condition is called a *node condition*. By means of these node conditions, the user can restrict the subgraphs that are selected. The node conditions put an extra restriction on embeddings. Thus, an *embedding* is a subgraph that matches the pattern and fulfills the node conditions.

However, in the context of browsing a more general condition can apply to a pattern step, which is therefore called the *browsing condition*. This condition links the pattern to embeddings found in previous steps, and is therefore essential for the process of browsing. In such a way, it is possible to combine the results of several steps by combining conditions, that refer to different steps, into one condition.

Secondly, it is possible to select a set of embeddings amongst the ones that resulted from a previous step. Such a step is called the *selection step*. That way it is possible to select only the embeddings that seem interesting to the user. It narrows down future searches.

Besides these two steps, there are two operators. Each of these operators changes, in fact, a browsing program, which is a sequence of browsing steps. The first one, the *change operator* changes one step of the program, which may affect the result of this step and also of later steps

that refer to that step. The second one, on the other hand, the *rollback operator*, rolls back one or more browsing steps.

The operators are introduced because they add to the intuitive ideas about browsing. Browsing means investigating the contents of the database in an interactive manner. The possibility to backtrack (as provided by the change and rollback operator), and retrace steps (as with the change operator) can be part of that process. Including the operators makes it possible to have more user friendly browsings sessions.

It is possible to show the results of subsequent statements in a tree, as will be shown later in this paper. The embeddings that are found in the pattern and selection steps make up the nodes of this tree, while each edge is labeled with the step label. Each of these steps adds one layer to the browsing tree. The layer that contains the result of step p is called layer p. The browsing tree is a visual aid for the representation of the result. The browsing tree not only shows the results of each step, but also shows (by means of the edges) the dependencies between nodes in different steps.

Every node in a pattern has a node label. However, we presume that the nodes of a pattern are linearly ordered and that it is therefore also possible to speak about e.g. the third node of a pattern. Consequently, embeddings can be represented by unlabeled tuples. Also, an infinite countable set $L = \{\texttt{l0}, \texttt{l1}, \texttt{l2}, \ldots\}$ of step labels is presumed, with `l0` denoting step 0.

2 Definition of the Pattern Browsing Model

2.1 An example

First an informal example is given that illustrates the notion of browsing and the intended meaning of the different statements. In this example steps as well as operators are used. Note that each time an operator is encountered in the program, this operator can be replaced by rewriting the steps of the program according to this operator.

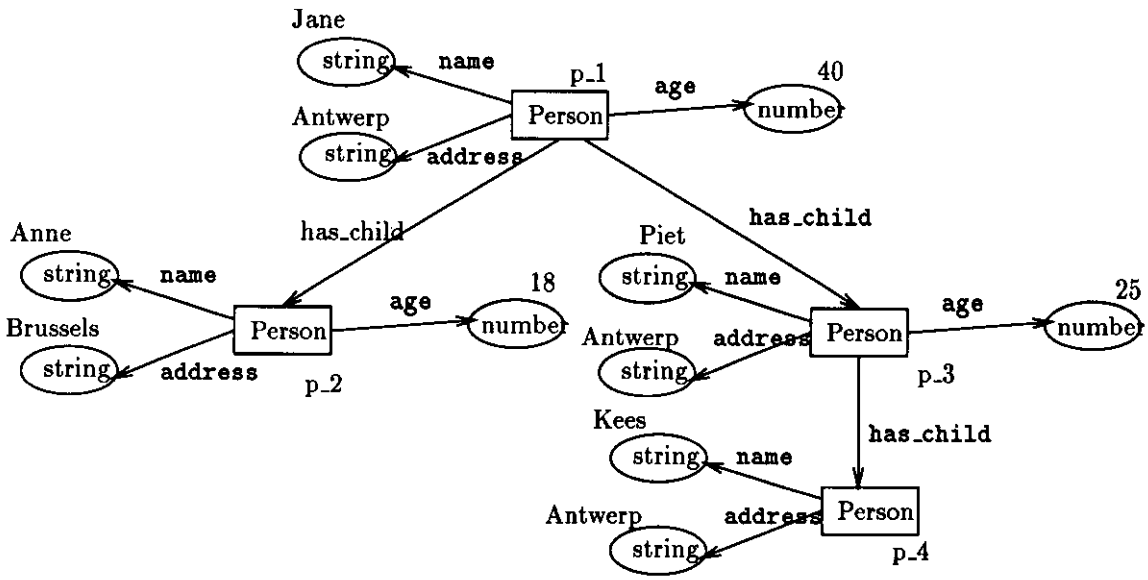
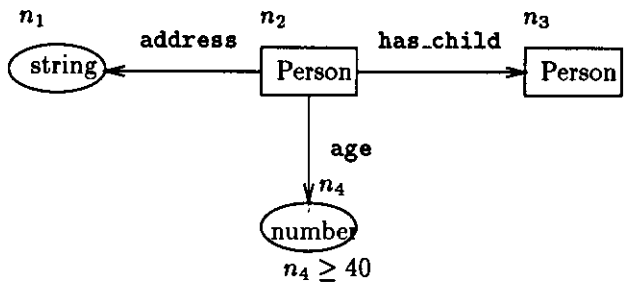


Figure 1: A simple instance

Example 1 *This example uses a small instance, given in Figure 1. In this instance the objects have an identification label, which makes it possible to distinguish between them.*

The instance represents four persons with their name, address and age (if those are known) together with their child–relationships.

Step 1 (pattern step): Select the nodes with pattern:

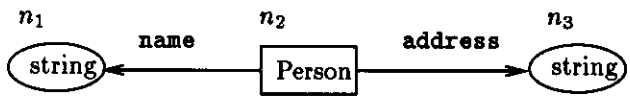


This query asks for all persons that are at least 40 years of age, together with their address and one of their children.

The result of this step is set of embeddings that match the pattern. Every embedding can therefore be represented by a (address, person, person, age)–tuple. The result can be represented by the following table:

n_1	*n_2*	*n_3*	*n_4*
Antwerp	*p_1*	*p_2*	*40*
Antwerp	*p_1*	*p_3*	*40*

Step 2 (pattern step with pattern condition): Select the nodes with pattern:



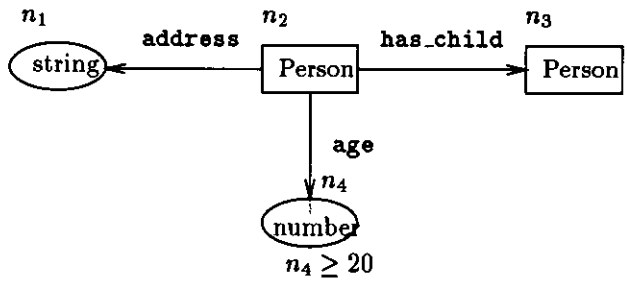
and condition: node n_2 matches node n_3 of step 1,
and node n_3 matches node n_1 of step 1.

Node n_2 of the instance (Person) has to match node n_3 of a tuple of step 1, which is the 'child' object of that pattern. In the same way, node n_3 of the instance (Address) has to match node n_1 of the same tuple, which is the 'address of the parent' of that pattern. This means that this query asks for all children who live at the same address of one of their parents, for those parents that were selected in step 1, together with their name and address.

This step will have the following table as result:

n_1	*n_2*	*n_3*
Piet	*p_3*	*Antwerp*

Step 3 (change operator): Change the pattern of step 1 into the following one:



This operator will change the original step 1. This adapted version of step 1 asks for all persons that are at least 20 years of age, together with their address and one of their children. Step 1 and 2 are executed again, which will now result in the following tables:

Step 1:

n_1	*n_2*	*n_3*	*n_4*
Antwerp	*p_1*	*p_2*	*40*
Antwerp	*p_1*	*p_3*	*40*
Antwerp	*p_3*	*p_4*	*25*

and step 2:

n_1	*n_2*	*n_3*
Piet	*p_3*	*Antwerp*
Kees	*p_4*	*Antwerp*

Step 4 (selection step): Select the following embedding from the result of step 2:

n_1	*n_2*	*n_3*
Kees	*p_4*	*Antwerp*

This query can be formulated as follows: select from the embeddings that where found in step 2, exactly the ones given in the table.

The result of this step is that the given embeddings are selected. This narrows down future searches as will be visualised when the results are presented in a browsing tree furtheron in this paper.

The result of this step is given by the following table:

n_1	*n_2*	*n_3*
Kees	*p_4*	*Antwerp*

Now imagine that the user finds that he made the wrong selection in step 4. What he can do now, is either change it with a change operator, or throw away step 4 altogether by means of a rollback operator. Imagine that he wants to do the latter. Step 5 would then look like:

Step 5 (rollback operator): Now roll back all actions from step 4 on.

This query can be translated as: throw away step 4.

The result of this step is that the result of step 4 is removed, and can no longer be used in subsequent steps.

This concludes the illustration of the different browsing steps.

2.2 Syntax of a browsing program

In this section, the syntax of a browsing program is described. First of all, the notion of a primitive browsing program is introduced. A primitive browsing program uses only (pattern and selection) steps. The syntax of the steps is given. Then, the notion of a non-primitive browsing program is introduced, in which also the change and rollback operators can be used. Also, the syntax of the operators is given.

The example given in the introduction is then worked out in the proper syntax. Finally, the result of the example is presented in a browsing tree.

2.2.1 Primitive statements: pattern step and selection step

Definition 1 *A* pattern step *has the following syntax:*

$l : (\underline{pat}\ p, \underline{cond}\ c);$

and is defined by

- *a label l*
- *a pattern p*
- *a pattern condition $c \in C$, where C is defined by:*

$C ::= (n_{i_1}, n_{i_2}, \ldots, n_{i_k}) ::= (\underline{\text{anc}}_{l'}\ (n_{j_1},\ n_{j_2},\ \ldots,\ n_{j_k}))\ |$
$(n_{i_1}, n_{i_2}, \ldots, n_{i_k}) ::= (\underline{\text{exist}}_{l'}\ (n_{j_1},\ n_{j_2},\ \ldots,\ n_{j_k}))\ |$ **NOT(** C**)** $|(C$ **AND** $C)|(C$ **OR** $C)|$**T**

where $n_{i_1}, n_{i_2}, \ldots, n_{i_k}$ *and* $n_{j_1}, n_{j_2}, \ldots, n_{j_k}$ *are node labels, and* l' *is a step label.*

This step is essentially a pattern matching step with a possibility to include a pattern condition c.

All nodes of the pattern are labeled with an identifier. In the pattern, each node may have a node condition. The syntax of the node condition of a node n_i is as follows:

$NC ::= n_i = e \mid n_i \leq e \mid n_i < e \mid n_i \geq e \mid n_i > e \mid$ **NOT(** NC**)** $\mid (NC$ **AND** $NC) \mid (NC$ **OR** $NC)$

where expression e is either a node label, a value, or an numerical expression (in which the binary operators +, −, *, / can be used) including a node label.

When a node is given a node condition, the value of this node is restricted. This means that each node in an embedding fulfills its condition. If e contains a node label n_j it means that the value of n_i is dependent on the value of n_j.
An example of a pattern with node-conditions is given in Figure 2. This pattern specifies persons who live in Antwerp and who are not older than 50 years, who have a child who is at least 30 years younger then him/herself. The result will consist of tuples of the form (name, address, person, age, person, age) for embeddings that match the pattern. They fulfill the following conditions: "Person n_3 lives in Antwerp, is less than 50 years old, and his/her stepchild n_5 is at least 30 years younger than him/herself.".

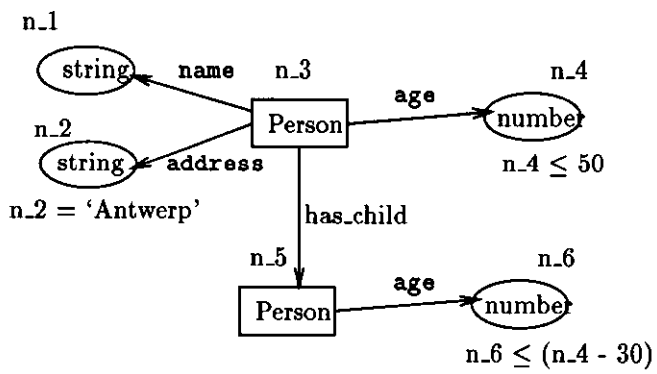


Figure 2: A pattern with node-conditions

In a pattern step $l : \langle \underline{pat}\ p, \underline{cond}\ c \rangle$, label l is used to identify the step within the program. Pattern condition c links the pattern to the selected embeddings of one or more preceding steps. It consists of subconditions of the form: $((n_{i_1}, n_{i_2}, \ldots, n_{i_k}) ::= (\underline{\text{anc}}_{l'}(n_{j_1},\ n_{j_2},\ \ldots,\ n_{j_k})))$ and $((n_{i_1}, n_{i_2}, \ldots, n_{i_k}) ::= (\underline{\text{exist}}_{l'}(n_{j_1},\ n_{j_2},\ \ldots,\ n_{j_k})))$. Each subcondition specifies for certain nodes of the pattern, which nodes of previously selected embeddings they have to match. There is a subtle difference between the meaning of the first and the second subcondition, which is explained later in the paper. Basically, they both say that nodes $n_{i_1}, n_{i_2}, \ldots, n_{i_k}$ have to match nodes $n_{j_1}, n_{j_2}, \ldots, n_{j_k}$ respectively, of an embedding of step l'. Later on in the paper, we will explain the semantics of anc and exist in more detail.

The result of a pattern step is the set of embeddings of the pattern that satisfy the pattern condition.

Definition 2 *A* selection step *has the following syntax:*

$l : \langle \underline{label}\ l', \underline{select}\ S \rangle$,

and is defined by

- *a label l*
- *a label l'*
- *a selection set S.*

The meaning of a selection step is that of the embeddings, found in step l', the ones in set S are selected. The idea behind a selection step is that the user selects the embeddings that he finds interesting and wants to continue with. Generally, it narrows down the search path for future steps. The pattern of selection step l is defined as the pattern of step l'. This means is that selection step l inherits the pattern of step l'. This pattern is needed when a future pattern step refers to step l in its pattern condition.

In the remainder of this paper we identify a step by its label, i.e. step l is short for the step with label l.

2.2.2 Primitive browsing programs

We introduce the notion of a closed primitive browsing program. A primitive browsing program is a sequence of browsing steps that fulfill certain conditions. These conditions make sure that each step encountered during the evaluation of the program has a defined meaning. A closed primitive browsing program is a browsing program that starts from scratch. An open primitive browsing program, on the other hand, takes an existing browsing program as input, and then adds one or more steps to it.

A pattern step with a non-empty pattern condition refers to a previously defined step in a browsing program. The embeddings of the pattern step then have to match an embedding of the referenced step. A selection step also refers to a previously defined step in a browsing program. The embeddings of its selection set are taken from the referenced step. To define the syntax of a primitive browsing program it is therefore necessary to know the set of steps that a pattern or selection step l can refer to. This is known as the set of referencable labels of step l.

Definition 3 *The* set of referencable labels of step l_i *in the context of a primitive browsing program* $[l_0, l_1, \ldots, l_{i-1}, l_i, \ldots, l_n]$, *is the set of labels that* l_i *may refer to in the program, and is defined as follows:*

$$srl([l_0, l_1, \ldots, l_{i-1}, l_i, \ldots, l_n], l_i) = \{\ l_0, l_1, \ldots, l_{i-1}\ \}$$

Definition 4 *A* closed primitive browsing program *is a finite list of browsing steps, for which the following holds:*

- *Every step in the list has a unique step label.*
- *If a pattern step l has a pattern condition containing a subcondition of the form* $(n_{i_1}, n_{i_2}, \ldots, n_{i_k}) ::= (\underline{\text{anc}}_{l'}(n_{j_1},\ n_{j_2}, \ldots,\ n_{j_k}))$, *or* $(n_{i_1}, n_{i_2}, \ldots, n_{i_k}) ::= (\underline{\text{exist}}_{l'}(n_{j_1},\ n_{j_2},\ \ldots,\ n_{j_k}))$ *then*
  - *l' has to be in the set of referencable labels of l*
  - *$n_{i_1}, n_{i_2}, \ldots, n_{i_k}$ are unique node labels of the pattern of step l*
  - *$n_{j_1}, n_{j_2}, \ldots, n_{j_k}$ are node labels of the pattern of step l'.*
- *For every selection step l : ($\underline{label}$ l', $\underline{select}$ S) it holds that*
  - *l' has to be in the set of referencable labels of l*
  - *selection set S only contains embeddings that are in the result of step l'.*

Definition 5 *An* open primitive browsing program *is a consecutive subsequence* $[l_i, \ldots, l_k]$ *of a closed primitive browsing program* $[l_0, \ldots, l_i, \ldots, l_k, \ldots, l_n]$.

2.2.3 Operators: change and rollback operator

Definition 6 *A* change operator *has the following syntax:*

change l *into* s;

where

s *is a pattern step* $l : \langle pat\ p, cond\ c \rangle$, *or a selection step* $l : \langle label\ l', select\ S \rangle$.

The meaning of the change operator is that it changes step l. The existing step l is replaced by the new step, identified by the change operator. The idea behind a change operator is, that the user can backtrack to a certain point in a program to change a move, and then retrace the rest of the steps.

Definition 7 *A* rollback operator *has the following syntax:*

rollback l;

where

l *is a step label.*

The meaning of a rollback operator is that it identifies a certain position in the program (by means of the step label). The program then rolls back all steps up to and including step l. It is like backtracking to the step that precedes step l.

Note that operators do not have a step label. They do not need one, because statements never refer to operators.

2.2.4 Non-primitive browsing programs

Non-primitive browsing programs are browsing programs in which steps as well as operators can be used. Therefore it is a sequence of statements. A non-primitive browsing program can be rewritten to a primitive browsing program. Therefore, primitive browsing programs have the same expressive power as non-primitive browsing programs.

Operators refer to browsing steps, which are identified by their label. Change operators refer to the step that they change, and rollback operators refer to the first step that has to be undone. To define the syntax of a non-primitive browsing program, it is therefore necessary to define a set of labels that an operator can refer to. This is known as the set of referencable labels of the operator. Since we already defined the set of referencable labels of a browsing step, we can now define the set of referencable labels of an arbitrary statement, in the context of a non-primitive browsing program.

Definition 8 *The* set of referencable labels of statement s_i, *in the context of a browsing program, is the set of labels that* s_i *may refer to in the program, and is defined as follows:*

$srl([l : \langle pat\ p, cond\ c \rangle] = \{l\}$,

$srl([s_0, \ldots, l : \langle pat\ p, cond\ c \rangle, s_i, \ldots, s_n], s_i) =$
$\quad srl([s_0, \ldots, l : \langle pat\ p, cond\ c \rangle], l : \langle pat\ p, cond\ c \rangle) \cup \{l\}$,

$srl([s_0, \ldots, l : \langle label\ l', select\ S \rangle, s_i, \ldots, s_n], s_i) =$
$\quad srl([s_0, \ldots, l : \langle label\ l', select\ S \rangle], l : \langle label\ l', select\ S \rangle) \cup \{l\}$,

$srl([s_0, \ldots, change\ l\ into\ s, s_i, \ldots, s_n], s_i) =$
$\quad srl([s_0, \ldots, change\ l\ into\ s), change\ l\ into\ s)$,

$srl([s_0, \ldots, step\ l, \ldots, rollback\ l, s_i, \ldots, s_n], s_i) =$
$\quad srl([s_0, \ldots, step\ l], step\ l)$,
$\quad where\ l \in srl([s_0, \ldots, step\ l, \ldots, rollback\ l], rollback\ l)$.

The set of referencable labels of a statement s_i in the context of a browsing program is determined by considering the previous statement s_{i-1}. If s_{i-1} is a step, then $srl(s_i)$ is equal to the union of $srl(s_{i-1})$ and the label of s_{i-1}. If, s_{i-1} is a change operator, then $srl(s_i)$ is equal to $srl(s_{i-1})$. The last possibility is that s_{i-1} is a rollback operator that rolls back all statements up and including s_l. Then, $srl(s_i)$ is equal to $srl(s_l)$. If, however, a statement has no predecessor it means that the statement itself is a pattern step, because a browsing program always starts with a pattern step. Therefore, the set of referencable labels of a pattern step in the context of itself, is defined.

Definition 9 *A* closed non-primitive browsing program *is a finite list of browsing statements, for which the following holds:*

1 *Every step in the list has a unique step label.*

2 *If a pattern step l has a pattern condition containing a subcondition of the form*
$(n_{i_1}, n_{i_2}, \ldots, n_{i_k}) ::= (\underline{\mathsf{anc}}_{l'}(n_{j_1}, n_{j_2}, \ldots, n_{j_k}))$, *or*
$(n_{i_1}, n_{i_2}, \ldots, n_{i_k}) ::= (\underline{\mathsf{exist}}_{l'}(n_{j_1}, (n_{j_2}, \ldots, n_{j_k}))$ *then*

- *l' is an element the set of referencable labels of l,*
- *$n_{i_1}, n_{i_2}, \ldots, n_{i_k}$ are unique node labels of the pattern of step l,*
- *$n_{j_1}, n_{j_2}, \ldots, n_{j_k}$ are node labels of the pattern of step l'.*

3 *For every selection step $l : \langle \underline{label}\ l', \underline{select}\ S \rangle$ it holds that*

- *l' is an element of the set of referencable labels of l,*
- *selection set S only contains embeddings that are in the result of step l'.*

4 *For every change operator $\underline{change}\ l$ into $l : \langle \underline{pat}\ p, \underline{cond}\ c \rangle$, it holds that*

- *l is an element of the set of referencable labels of this change operator,*
- *step l is a pattern step $l : \langle \underline{pat}\ p', \underline{cond}\ c' \rangle$ such that p' at most differs from p in its relationships between its nodes and in its node conditions,*
- *$l : \langle \underline{pat}\ p, \underline{cond}\ c \rangle$ conforms to rule* [2].

5 *For every change operator $\underline{change}\ l$ into $l : \langle \underline{label}\ l', \underline{select}\ S \rangle$ it holds that*

- *l is an element of the set of referencable labels of this change operator,*
- *step l is a selection, step $l : \langle \underline{label}\ l', \underline{select}\ S' \rangle$, which therefore also selects embeddings from step l',*
- *$l : \langle \underline{label}\ l', \underline{select}\ S \rangle$ conforms to rule* [3].

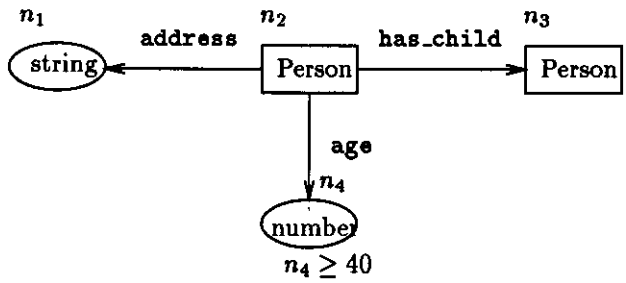
6 *For every rollback operator $\underline{rollback}\ l$ it holds that l is an element of the set of referencable labels of this rollback operator.*

Definition 10 *An* open non-primitive browsing program *is a consecutive subsequence $s_i, \ldots, s_k\rangle$ of a closed non-primitive browsing program $\langle s_0, \ldots, s_i, \ldots, s_k, \ldots, s_n \rangle$.*

Example 2 *Here, Example 1 is worked out in the notation of the pattern browsing technique. It is a closed, non-primitive browsing program.*

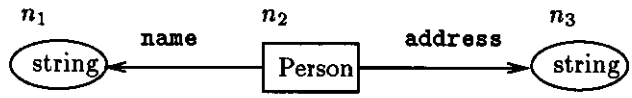
```
[

    11:
   <pat
    ---
```



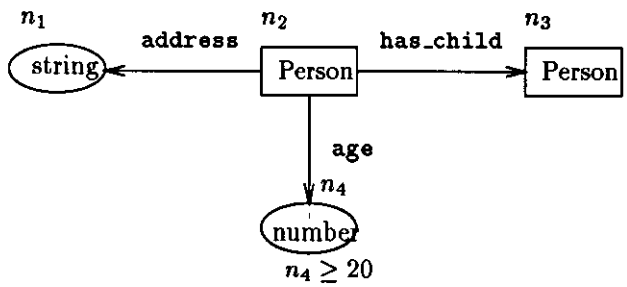

```
 , cond T>,
   ----

12:
<pat
 ---
```




```
 , cond (n2, n3) :: = (anc[l1](n3,n1))>,
   ----

change l1 into
------
  l1:
  <pat
   ---
```




```
   , cond: T>,
     ----
```

```
13:
<label 12, select: {(Kees, p_4, Antwerp)}>,
 ----      ------

rollback 13
--------
```

]

The resulting browsing trees of executing respectively the first to fifth statement of Example 2 are given in Figures 3, 4, 5, 6, and 7. The edges of each layer in a browsing tree are labeled with the corresponding step label. Executing the rollback operation deletes layer $l3$ as shown by the cutting edge. The result of a subsequent pattern or selection step, would then be drawn in level 3 of the browsing tree.

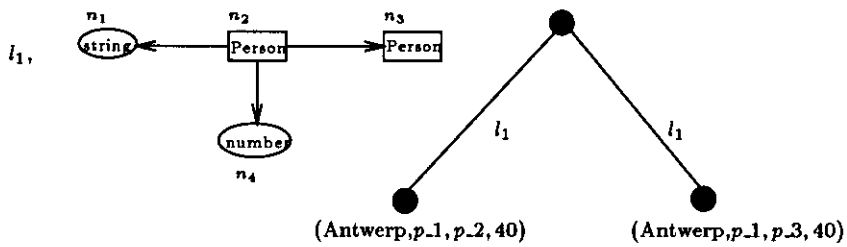


Figure 3: Browsing tree after executing the first statement

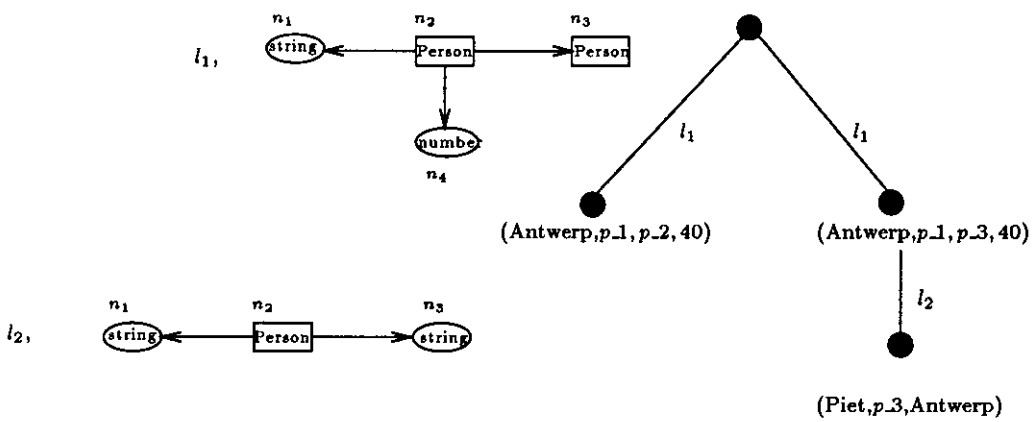


Figure 4: Browsing tree after executing the first and second statement

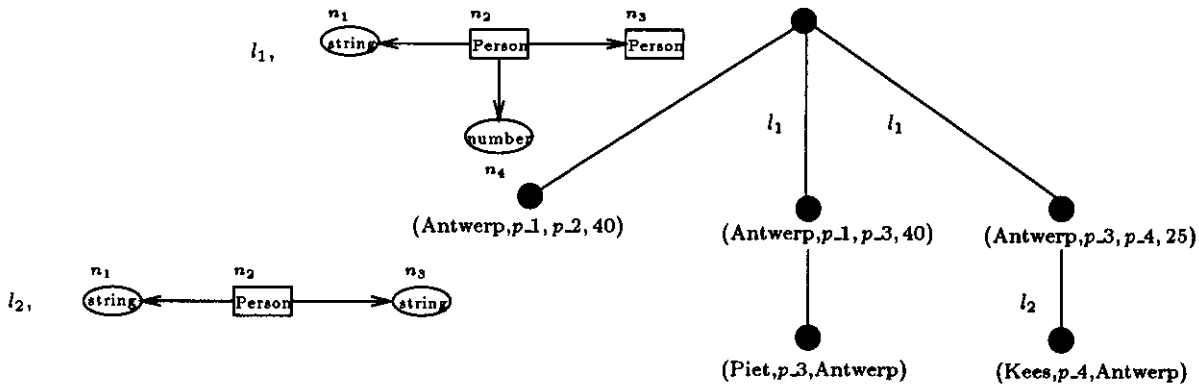


Figure 5: Browsing tree after executing the first to third statement

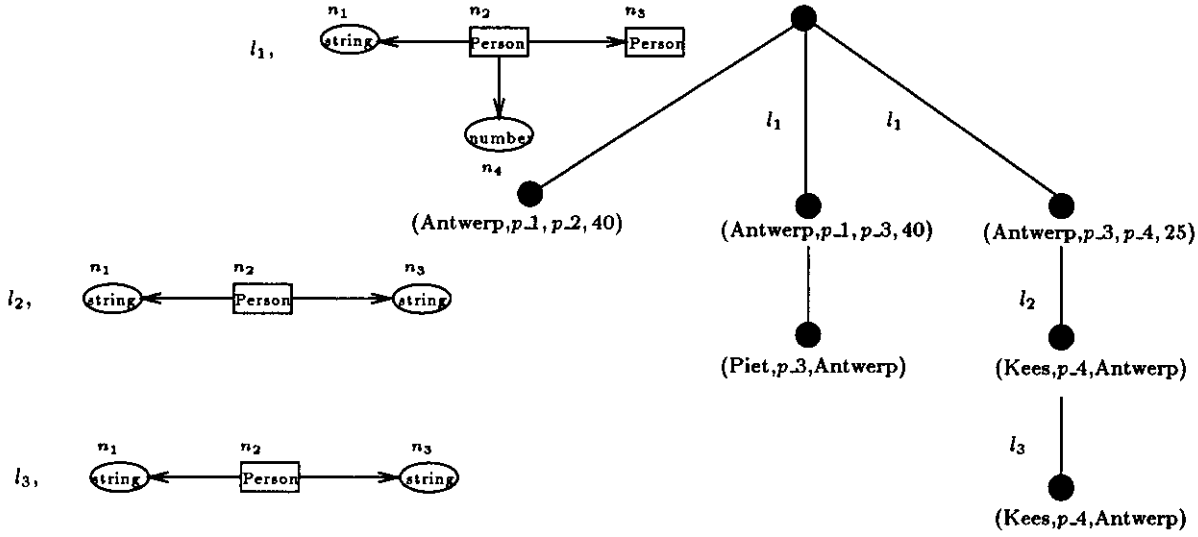


Figure 6: Browsing tree after executing the first to fourth statement

2.3 Semantics of a browsing program

In this section the exact meaning of a browsing program is given, by means of its semantics. First, the definition of a browsing tree is given. A browsing tree provides the possibility to visualize the result of a browsing program. There are different ways to represent the results in the browsing tree. For instance, edges can be labeled with a step label, with a step label and the pattern of the step, or with the complete step. Also, the resulting embeddings can be represented as a tuple or as a graph. In this model, the edges are label with a step label and the pattern of the step. This makes it possible for other steps to refer to a step (and its pattern) in the browsing tree.
A primitive browsing program is made up of individual browsing steps, therefore the semantics of these steps is given. The semantics of the different steps is given with respect to a certain instance and a browsing tree. Then the semantics of a browsing program is given in terms of the semantics of the individual steps. Further, it is shown how a non-primitive browsing program can be translated into a primitive browsing program.

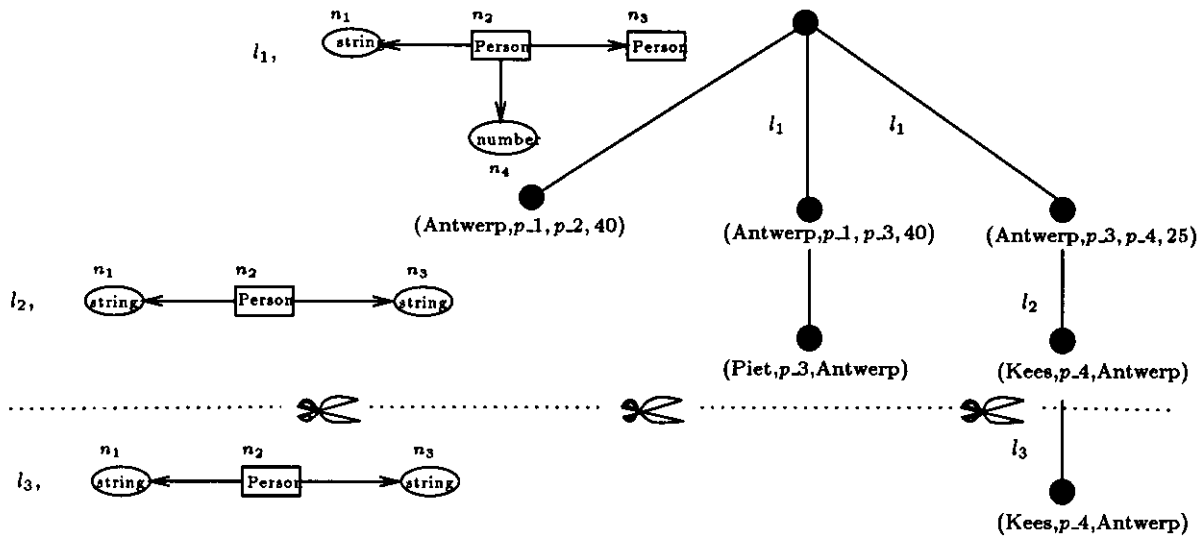


Figure 7: Browsing tree after executing the whole program

2.3.1 Browsing trees

The result of a browsing program can be shown in a tree: the browsing tree. Every step extends the browsing tree with a new layer. For each node in the bottom layer a (possibly empty) set of embeddings is added to the tree. Which embeddings are in this set, depends on the step itself, and of the embeddings in layers of steps that are referenced by in this step. A change operator backtracks to a certain layer, changes that step and then retraces its steps back to the bottom layer. A rollback operator rolls back a sequence of executed statements, up to and including the step indicated by the operator. In the browsing tree, this means that a rollback operator removes the layer of the indicated step, and all layers beneath it.

Definition 11 *A* browsing tree *is a labeled tree where each edge is labeled with the label and the pattern of a step, and where each node is labeled with an embedding. Furthermore, it must hold that:*

- *The root is labeled with the empty embedding.*
- *Edges are labeled with the same step label if and only if they are on the same level.*
- *Embeddings in a layer of the browsing tree are embeddings of the pattern of that layer.*
- *Every node of the tree can be uniquely identified by the list of edge and node labels that are encountered on the path from the root to that node.*

The browsing tree that consists of only the root is called the empty browsing tree or bt_0. The expression $result(bt, l)$ represents the set of nodes in the browsing tree bt that are the result of step l, except for $result(bt, 0)$ which represents the root of bt.

2.3.2 Semantics of the pattern and selection step

Definition 12 *The* semantics of a pattern step *is a partial function that given a pattern step, an instance and a browsing tree determines the updated browsing tree.*

$sps(l : \langle \underline{pat}\ p, \underline{cond}\ c \rangle, i, bt)$ *is defined whenever*

- *label l does not yet have a layer in bt, and*

- *for every subcondition in c of the form* $(n_{i_1}, n_{i_2}, \ldots, n_{i_k}) ::= (\underline{\text{anc}}_{l'}\ (n_{j_1},\ n_{j_2},\ \ldots,\ n_{j_k}))$ *or* $(n_{i_1}, n_{i_2}, \ldots, n_{i_k}) ::= (\underline{\text{exist}}_{l'}\ (n_{j_1},\ n_{j_2},\ \ldots,\ n_{j_k}))$
  - $n_{i_1}, n_{i_2}, \ldots, n_{i_k}$ *are unique labels of a node of pattern p,*
  - l' *is the label of a layer in the browsing tree,*
  - $n_{j_1},\ n_{j_2},\ \ldots,\ n_{j_k}$ *are labels of a node of the pattern of layer* l'.

The result of $sps(l : \langle \underline{pat}\ p, \underline{cond}\ c\rangle, i, bt)$ *is a browsing tree that is a super-tree of bt in the following way. For each node n in the bottom layer of bt, every embedding e of step pattern p into instance i is considered. If this embedding fulfills condition c a node m is added to bt, as well as an edge from node n to node m. Node m is labeled with embedding e, and the edge is labeled with label l and pattern p.*
The super-tree thus has an extra (possible empty) layer l, as opposed to bt.

Whether embedding e fulfills condition c is defined by induction on the condition:

- $(n_{i_1}, n_{i_2}, \ldots, n_{i_k}) ::= (\underline{\text{exist}}_{l'}\ (n_{j_1},\ n_{j_2},\ \ldots,\ n_{j_k}))$ *holds if and only if* $\exists$ *embedding* $e' \in result(bt, l')$ *such that* $\forall t \in \{1, 2, \ldots, k\}\ (e(n_{i_t}) = e'(n_{j_t}))$
- $(n_{i_1}, n_{i_2}, \ldots, n_{i_k}) ::= (\underline{\text{anc}}_{l'}\ (n_{j_1},\ n_{j_2},\ \ldots,\ n_{j_k}))$ *holds if and only if* $\exists$ *embedding* $e' \in result(bt, l')$ *such that*
  1. $\forall t \in \{1, 2, \ldots, k\}\ (e(n_{i_t}) = e'(n_{j_t}))$ *, and*
  2. e' *lies on the path from tree node n to the root of bt.*
- $(c_1$ `AND` $c_2)$ *holds if and only if* c_1 *holds and* c_2 *holds.*
- $(c_1$ `OR` $c_2)$ *holds if and only if* c_1 *holds or* c_2 *holds.*
- `NOT`(c) *holds if and only if c does not hold.*
- `T` *holds always.*

Definition 13 *The* semantics of a selection step *is a partial function that given a selection step and a browsing tree determines the new (extended) browsing tree.*

$scs(l : \langle \underline{label}\ l', \underline{select}\ S\rangle, bt)$ *is defined whenever*

- *label* l' *has a layer in bt, and*
- *label l does not yet have a layer in bt, and*
- $\forall e : e \in S : e \in result(bt, l')$.

The result of $scs(l : \langle \underline{label}\ l', \underline{select}\ S\rangle, bt)$ *is a browsing tree that is a super-tree of bt in the following way. For each node n in the bottom layer of bt node* n'_l *is considered, where* n'_l *is defined as follows:*

- n'_l *lies on the path from n to the root of bt, and*
- n'_l *is a node in layer* l'.

If the label of n'_l *is an embedding* $e' \in$ *selection set S, then a new node m is added, as well as an edge from node n to node m. Node m is labeled with embedding* e'*, and the edge is labeled with l and the pattern of layer* l'.
The super-tree thus has an extra (possible empty) layer l, as opposed to bt.

Semantics of a primitive browsing program

Definition 14 *The* semantics of an open primitive browsing program *is a partial function that given an open primitive browsing program, a browsing tree and an instance determines the new (extended) browsing tree.*

Whether the result of $sopb([s_1, \ldots, s_k], bt, i)$ *is defined, can be deduced by induction upon the structure of the browsing program:*

- $sopb([s_1], bt, i)$ *is defined if either* s_1 *is a pattern step and* $sps(s_1, i, bt)$ *is defined, or* s_1 *is a selection step and* $scs(s_1, bt)$ *is defined.*
- $sopb([s_1, s_2, \ldots, s_k], bt, i)$ *is defined if* $sopb([s_1], bt, i)$ *is defined and* $sopb([s_2, \ldots, s_k], sopb([s_1], bt, i), i)$ *is defined.*

This means that the function is defined if for every step s, the step semantics function sps or scs (depending on whether it is a pattern or selection step) is defined in the context of the tree that this step takes as input tree. This is the tree that is created if the steps that precede s, are executed with bt as original input tree.

The result of $sopb([s_1, \ldots, s_k], bt, i)$ *is defined as follows:*

$$sopb([], bt, i) = bt$$
$$sopb([s_1, s_2, \ldots, s_k], bt, i) = \begin{array}{l} sopb([s_2, \ldots, s_k], sps(s_1, i, bt), i) \text{ if } s_1 \text{ is a pattern step} \\ sopb([s_2, \ldots, s_k], scs(s_1, bt), i) \text{ if } s_1 \text{ is a selection step} \end{array}$$

Definition 15 *The* semantics of a closed primitive browsing program *is a function that given a closed primitive browsing program and an instance determines the browsing tree of the program.*

In fact, the semantics is equal to the open browsing program semantics if it takes the closed browsing program together with the empty browsing tree (bt_0*, which consists only of the root) as input. The only difference is, that the well defined structure of a closed browsing program ensures that the semantics function exists.*

Hence, $scpb([s_1, \ldots, s_n], i) = sopb([s_1, \ldots, s_n], bt_0, i)$.

Semantics of a non-primitive browsing program

Instead of defining the semantics of a non-primitive program in terms of the semantics of its individual statements, a translation is given to a primitive browsing program. The semantics of a primitive browsing program which has been given earlier in this paper, can then be applied. The reason we can do this, is that each operator is, in fact, a short cut for rewriting an existing primitive browsing program to another primitive browsing program.

Definition 16 *The* translation of a closed non-primitive browsing program $trans : CBP \rightarrow CPBP$ *is a function that given a closed non-primitive browsing program cbp translates it to an equivalent closed primitive browsing program cpbp.*

This function is defined by induction upon the structure of the browsing program. The result of $trans(obp)$ *is defined as follows:*

$trans([s_1, \ldots, l : \langle \underline{pat}\ p, \underline{cond}\ c \rangle, \ldots, s_k, \underline{change}\ l\ into\ \ l : \langle \underline{pat}\ p', \underline{cond}\ c' \rangle, s_t, \ldots, s_n])$,
where $s_1, \ldots, s_k$ *are steps,*
$= trans([s_1, \ldots, l : \langle \underline{pat}\ p', \underline{cond}\ c' \rangle, \ldots, s_k, s_t, \ldots, s_n])$,

$trans([s_1, \ldots, l : \langle \underline{label}\ l', \underline{select}\ s \rangle, \ldots, s_k, \underline{change}\ l\ into\ \ l : \langle \underline{label}\ l', \underline{select}\ S' \rangle, s_t, \ldots, s_n])$,
where $s_1, \ldots, s_k$ *are steps,*
$= trans([s_1, \ldots, l : \langle \underline{label}\ l', \underline{select}\ S' \rangle, \ldots, s_k, s_t, \ldots, s_n])$,

$trans([s_1, \ldots, s_i,\ step\ l, \ldots,\ s_k,\ \underline{rollback}\ l,\ s_t, \ldots,\ s_n])$
$\quad$ *where* $s_1, \ldots,\ s_k$ *are steps,*
$\quad = trans([s_1, \ldots, s_i, s_t, \ldots, s_n]),$

$trans([s_1, \ldots, s_n])$*, where* $s_1, \ldots, s_n$ *are steps,*
$\quad = [s_1, \ldots, s_n].$

Because closed non-primitive browsing programs have, by definition, a well defined structure, the above translation rules will always result in a closed primitive browsing program.

The translation function traverses to the first operator in the list of statements. If this is a change operator, it changes the indicated pattern or selection step, and then continues. The change operator is deleted from the list. If the first operator in the list is a rollback operator, the steps between the indicated step and the rollback operator are deleted from the list, as well as the rollback operator itself. The translation function is then repeated, with the new list of steps as input. This repeats itself until all steps in the list are primitive steps.

Definition 17 *The* semantics of a closed non-primitive browsing program *is a function that given a closed non-primitive browsing program determines the browsing tree of the program.*

The semantics can now be defined in terms of the translation function as follows:
$scb([s_1, \ldots, s_n], i) = scpb(trans([s_1, \ldots, s_n]))$.

This means that a closed non-primitive program is first translated to a closed primitive program. Then the semantics of a closed primitive program is applied to this primitive program.

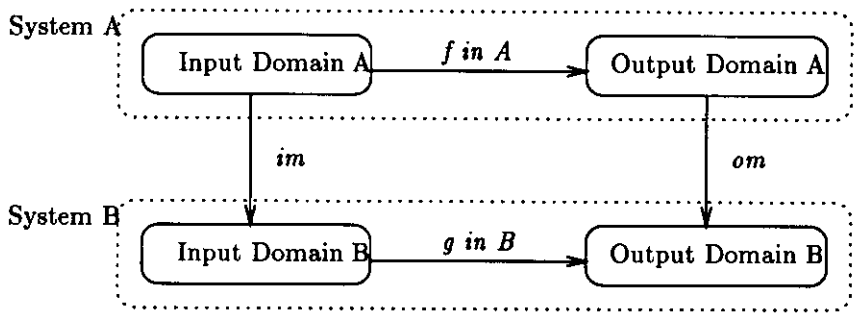


Figure 8: Simulation Diagram

3 The Expressiveness of the Model

In this section we will discuss the expressiveness of the browsing model. With expressiveness we mean the ability to calculate certain transformations. The browsing model will be compared with the relational algebra in several ways. We will consider only the model with primitive programs (consisting of steps only) since operations can be regarded as syntactic sugar. Also, we will not consider arithmetic in the node conditions because this would make the model incomparable with the relational model.

To be able to determine whether the browsing model is as expressive as the relational algebra and vice versa we have to establish what it means for one system of transformations to be as expressive as another.

Definition 18 *Let A and B be transformation systems and let im be a total mapping of the input domain of A to the input domain B and let om be a total mapping of the output domain of A to the output domain of B, as shown in Figure 8). A is* as expressive as *B* under input mapping *im* and output mapping *om if for every transformation g in B there is a transformation f in A such that $om \circ f = im \circ g$. If the reverse holds we say that B* can implement *A under input mapping im and output mapping om.*

Whether the browsing model is as expressive as the relational algebra depends upon the chosen mappings. Another important factor is how we choose the input and output domain of the browsing model and what we consider to be the transformations that are defined by the system.

For the relational model these choices are fairly obvious:

Input Domain A set of relations.

Output Domain A single relation.

Transformations Those expressed by an expression of the relational algebra.

For the browsing model they are less obvious. The first system we will consider is the following:

Input Domain A browsing tree.

Output Domain The embeddings of the nodes in the bottom layer of the resulting browsing tree.

Transformations Those expressed by a an open browsing program.

This system is called PBM_1 (Pattern Browsing Model 1). Now we know what the system is we can determine the mapping of the input and the output domains. The input mapping maps the layers of the browsing tree to tables containing the embeddings of the nodes of those layers. There will be a table for every layer and its name will the label of that layer. This mapping is

l_1			
n_1	n_2	n_3	n_4
Antwerp	p_1	p_2	40
Antwerp	p_1	p_3	40
Antwerp	p_3	p_4	25

l_2		
n_1	n_2	n_3
Piet	p_3	Antwerp
Kees	p_4	Antwerp

Figure 9: Input mapping example for im_1

called im_1. An example is given in Figure 9 where the result of im_1 applied to the browsing tree of Figure 7 (after the roll back operator) is shown.

The output mapping om_1 maps the embeddings of the nodes in the bottom layer to a table contain exactly these embeddings.

Theorem 1 *Under input mapping im_1 and output mapping om_1 it holds that PBM_1 is as expressive as the relational algebra.*

Proof: The proof is given by induction upon the algebra expression. We will show that the theorem holds for every individual algebra operator. Because we can concatenate them into one program it follows that any algebra expression can be simulated. It is presumed that every step receives a fresh step label that is not in the browsing tree. The exact order is not relevant but it should hold that the simulations of subexpressions precede the simulation of the comprising expression.

The translation of algebra operators in steps is demonstrated in Figure 10. Notice that in this proof we presume that there is only one node label A. It is possible to generalize the proof for more node labels but this would complicate the proof. Note also that we do not consider relational algebra operations such as $\sigma_{a<b}(R)$ but it is easy to see how they might be simulated with the help of node conditions. □

The expressiveness of PBM_1 as stated in Theorem 1 is slightly "crude". It says something about the manipulation of complete layers whereas in the browsing model with every step an extension is calculated for *every separate branch*. This explains why the anc-references were not really needed for the simulation, because they refer to the branch-specific embeddings. It is possible to define a more subtle expressiveness by extending the input domain of PBM_1 with a branch identification and define the output as the embeddings of the nodes at the bottom of the tree that was added to this branch. The relational algebra expression would then not only operate on tables containing the layers but also on tables containing the (single) embedding of every layer in the branch. The question whether the browsing model is as expressive as the relational algebra in this way, is still open.

Up to now we have regarded the expressiveness of the browsing model as a manipulation language for browsing trees. Yet, the user is only interested in the browsing tree as an intermediate result and, ultimately, is more interested in the final result (a set of embeddings) he can obtain starting with a certain instance. Therefore, it may be more interesting to see which sets of embeddings can be found given a certain instance and starting with the empty browsing tree. For this purpose we define PBM_2:

Input Domain An instance.

Output Domain The embeddings of the nodes in the bottom layer of the resulting browsing tree.

Transformations Those expressed by a closed browsing program.

The input mapping im_2 maps an instance (a labeled graph) to unary and binary tables. For every node label there will be an unary table of that name containing all the nodes in the instance

$l_a \Rightarrow$ pattern: $n_1 : [A] \ldots n_p : [A]$
condition: $(m_1, \ldots, m_p) ::= \underline{\text{exist}}_{l_a}(n_1, \ldots, n_p)$
where p is the arity of step l_a and $m_1, \ldots, m_p$ are the nodes of step l_a.

$\sigma_{n_i=v}(l_a) \Rightarrow$ pattern: $n_1 : [A] \ldots n_p : [A]$ with $n_i = v$ as node condition of n_i
condition: T

where v is a printable value and $n_1, \ldots, n_p$ are the nodes of step l_a.

$\pi_{n_1,\ldots,n_p}(l_a) \Rightarrow$ pattern: $n_1 : [A] \ldots n_p : [A]$
condition: $(n_1, \ldots, n_p) ::= \underline{\text{exist}}_{l_a}(n_1, \ldots, = n_p)$

$\rho_{n_i \mapsto m}(l_a) \Rightarrow$ pattern: $n_1 : [A] \ldots n_{i-1} : [A] m : [A] n_{i+1} : [A] \ldots n_p : [A]$
condition: $(n_1, \ldots, n_{i-1}, n_i, n_{i+1}, \ldots, n_p) ::= \underline{\text{exist}}_{l_a}(n_1, \ldots, n_{i-1}, m, n_{i+1}, \ldots, n_p)$
where $n_1, \ldots, n_p$ are the nodes of step l_a.

$(l_a \bowtie l_b) \Rightarrow$ pattern: $m_1 : [A] \ldots m_p : [A]$
condition: $(n_1, \ldots, n_q) ::= \underline{\text{exist}}_{l_a}(n_1, \ldots, n_q)$**AND**$(n'_1, \ldots, n'_r) ::= \underline{\text{exist}}_{l_b}(n'_1, \ldots, n'_r)$
where $n_1, \ldots, n_q$ and $n'_1, \ldots, n'_r$ are, respectively, the nodes of step l_a and step l_b, and $\{m_1, \ldots m_p\}$ is the union of these two sets.

$(l_a - l_b) \Rightarrow$ pattern: $n_1 : [A] \ldots n_p : [A]$
condition: $(n_1, \ldots, n_p) ::= \underline{\text{exist}}_{l_a}(n_1, \ldots, n_p)$**AND NOT**$((n_1, \ldots, n_p =) ::= \underline{\text{exist}}_{l_b}(n_1, \ldots, n_p))$
where $n_1, \ldots, n_p$ are the nodes of step l_a and step l_b.

$(l_a \cup l_b) \Rightarrow$ pattern: $n_1 : [A] \ldots n_p : [A]$
condition: $(n_1, \ldots, n_p) ::= \underline{\text{exist}}_{l_a}(n_1, \ldots, n_p)$**OR**$(n_1, \ldots, n_p) ::= \underline{\text{exist}}_{l_b}(n_1, \ldots, n_p)$
where $n_1, \ldots, n_p$ are the nodes of step l_a and step l_b.

Figure 10: Translation of relational algebra operators

with that label. For every edge label there will be a binary table of that name containing all the pairs of begin and end nodes of edges in the instance. The output mapping om_2 is the same as om_1 and maps the embeddings of the bottom layer nodes to a table containing exactly these embeddings.

Theorem 2 *Under input mapping im_2 and output mapping om_2 it holds that PBM_2 is as expressive as the relational algebra.*

Proof: The edges and nodes can be encoded in the layer as, respectively, binary and unary embeddings. The first steps of the (closed) program will therefore be $n_1 : [N]$ for every node label N and $n_1 : [N_1] \xrightarrow{\alpha} n_2 : [N_2]$ for every edge label α allowed between the node labels N_1 and N_2. The rest of the program is constructed according to Theorem 1, performing the algebraic expression upon these layers. □

An important difference between PBM_1 and PBM_2 is the injectivity of the input mapping im_2. This means that no information is lost during that mapping and that it is now a meaningful question to ask the dual question of Theorem 2, i.e., can the relational algebra implement PBM_2?

Theorem 3 *Under input mapping im_2 and output mapping om_2 it holds that the relational algebra can implement PBM_2.*

Before we can give the proof of this theorem we need a lemma that says that the relational algebra can implement a single pattern step of the browsing model. Therefore we define PBM_2':

Input Domain An instance and a browsing tree.

Output Domain The extended browsing tree.

Transformations Those expressed by a single pattern step.

The input mapping im_2' maps the instance to unary and binary tables as im_2. Furthermore, the browsing tree is mapped a set of relations with for every layer a relation containing all the embeddings of that layer and a single relation representing the browsing tree limited to its bottom nodes and their ancestors. The relations representing a layer are named with the label of the layer they represent. Their column names will be the the nodes of the step of that layer. The relation representing (a part of) the browsing tree is named BT and its column names will be all $l_i.n_j$ with l_i a label of a step of a layer in the browsing tree and n_j a node in the pattern of that step. This relation will contain a tuple for every node in the bottom layer of the browsing tree consisting of the joined embeddings that are encountered on the path from this node to the root of the browsing tree. An example is given in Figure 11 where the result is shown of applying im_2' to the browsing tree of Figure 11 (after the rollback operation). Note that the tables representing the layers are not simply projections of the table BT.

The output mapping om_2' maps the new browsing tree to a relation in the same way as im_2' maps the old browsing tree to BT.

Lemma 4 *Under input mapping im_2' and output mapping om_2' it holds that PBM_2' can be implemented by the relational algebra.*

Proof: The pattern step that is to be implemented is $l : (\underline{pat}\ p_l, \underline{cond}\ c_l)$. The relational algebra expression that will do this is constructed in 4 phases:

1. The first phase consists of construction the algebra expression *emb* that determines the embeddings of the pattern p_l in the instance without considering the node conditions. Let p_l be (N, E, λ) where λ is the function that gives the labels of nodes in N and edges in E. For every node n_i in N we construct $ne_{n_i} = \rho_{\#1 \rightarrow n_i}(\lambda(n_i))$. Note that $\lambda(n_i)$ denotes here the unary table that belongs to the label $\lambda(n_i)$ with the single column #1. For every edge

l_1			
n_1	n_2	n_3	n_4
Antwerp	p_1	p_2	40
Antwerp	p_1	p_3	40
Antwerp	p_3	p_4	25

l_2		
n_1	n_2	n_3
Piet	p_3	Antwerp
Kees	p_4	Antwerp

BT						
$l_1.n_1$	$l_1.n_2$	$l_1.n_3$	$l_2.n_4$	$l_2.n_1$	$l_2.n_2$	$l_2.n_3$
Antwerp	p_1	p_2	40	Piet	p_3	Antwerp
Antwerp	p_1	p_4	25	Kees	p_4	Antwerp

Figure 11: Input mapping example for im_2'

e_j in E we construct $ee_{e_j} = \rho_{\#1 \mapsto n_b}(\rho_{\#2 \mapsto n_e}(\lambda(e_j)))$ where n_b and n_e are, respectively, the begin and end node of e_j. Note that $\lambda(e_j)$ denotes here the binary table that belongs to the label $\lambda(e_j)$ with the two columns #1 and #2. Finally, the table of embeddings can be constructed by simply joining all the relations of the nodes and the edges: $emb = (ne_{n_1} \bowtie \ldots \bowtie ne_{n_p}) \bowtie (ee_{e_1} \bowtie \ldots \bowtie ee_{e_q})$ where $N = \{n_1, \ldots, n_p\}$ and $E = \{e_1, \ldots, e_q\}$.

2. The second phase constructs the algebra expression emb_{nc} that determines the set of embeddings that fulfill the node conditions. For this purpose we define the function $\mathcal{C} : AE \times NC \rightarrow AE$ where AE is the set of relational algebra expressions and NC is the set of node conditions, such that $\mathcal{C}(emb, c)$ is the algebra expression that gives all the embeddings that fulfill node condition c. It is defined with induction upon the condition:

   (a) $\mathcal{C}(ae, c) = ae$ if the condition c is empty.

   (b) $\mathcal{C}(ae, n_i\ \theta\ m) = \sigma_{n_i \theta m}(ae)$ where θ is either $=$, $\leq$, $<$, $\geq$ or $>$.

   (c) $\mathcal{C}(ae, \texttt{NOT}(c)) = ae - \mathcal{C}(ae, c)$.

   (d) $\mathcal{C}(ae, (c_1\ \texttt{AND}\ c_2)) = \mathcal{C}(ae, c) \cap \mathcal{C}(ae, c)$.

   (e) $\mathcal{C}(ae, (c_1\ \texttt{OR}\ c_2)) = \mathcal{C}(ae, c) \cup \mathcal{C}(ae, c)$.

   The complete expression emb_{nc} now has the form $\mathcal{C}(emb, c_1) \cap \ldots \cap \mathcal{C}(emb, c_p)$ where $c_1, \ldots, c_p$ are the node conditions of, respectively, the nodes $n_1, \ldots, n_p$.

3. The third phase constructs the expression nbt that renames the columns of the result of emb_{nc} to appropriate names for the browsing tree representation (by prefixing l, the label of the step) and joins the result with the original browsing tree representation (BT).
   $nbt = BT \bowtie (\rho_{n_1 \mapsto l.n_1}(\ldots \rho_{n_p \mapsto l.n_p}(emb_{nc}) \ldots))$

4. The fourth phase constructs the final expression that selects from the result of nbt those tuples that contain embeddings that fulfill the pattern condition c_l. As in the second phase we define a function $\mathcal{C}' : AE \times PC \rightarrow AE$ where PC is the set of pattern conditions, such that $\mathcal{C}'(emb_{nc}, c)$ selects from the result of emb_{nc} those tuples that contain embeddings that fulfill the pattern condition c. It is defined with induction upon the condition:

   (a) $\mathcal{C}'(ae, \texttt{T}) = ae$

   (b) $\mathcal{C}'(ae, (n_1, \ldots, n_r) ::= \underline{\text{anc}}_{l'}(m_1, \ldots, = m_r)) = \sigma_{l.n_1 = l'.m_1}(\ldots \sigma_{l.n_r = l'.m_r}(ae) \ldots)$

   (c) $\mathcal{C}'(ae, (n_1, \ldots, n_r) ::= \underline{\text{exist}}_{l'}(m_1, \ldots, m_r)) =$
   $\pi_{a_1, \ldots, a_s}(\sigma_{l.n_1 = m_1}(\ldots \sigma_{l.n_r = m_r}(ae \bowtie l') \ldots))$
   where $a_1, \ldots, a_s$ are the column names of the result of emb_{nc} and l' denotes the table containing the embeddings of the nodes in the layer with label l'.

   (d) $\mathcal{C}'(ae, \texttt{NOT}(c)) = ae - \mathcal{C}'(ae, c)$.

(e) $C'(ae, (c_1 \texttt{ AND } c_2)) = C'(ae, c) \cap C'(ae, c)$.

(f) $C'(ae, (c_1 \texttt{ OR } c_2)) = C'(ae, c) \cup C'(ae, c)$.

The final relational algebra expression that simulates step l is now given by $C'(nbt, c_p)$.

□

Just like the pattern step can be simulated in the relational algebra it is also possible to simulate the choice step in the relational algebra. Therefore we define PBM_2'' that is equal to PBM_2' except that the transformation are those expressed by a single choice step.

Lemma 5 *Under input mapping im_2' and output mapping om_2' it holds that PBM_2'' can be implemented by the relational algebra.*

Proof: The choice step that is to be implemented is $l : (\underline{label}\ l', \underline{set}\ \{\eta_1, \ldots, \eta_q\})$ where $\eta_1, \ldots, \eta_q$ are all embeddings.

We will construct for every embedding η_i an expression ae_i that selects those tuples that have that embedding for step l':
$ae_i = \sigma_{l'.n_1=\eta_i(n_1)}(\ldots \sigma_{l'.n_p=\eta_i(n_p)}(BT)\ldots)$ where $n_1, \ldots, n_p$ are the nodes of the pattern of step l'. The final result can then be computed by taking the union of the tables found per embedding: $ae_1 \cup \ldots \cup ae_q$ □

With the last two lemmas it is now easy to prove the previous theorem.

Proof: (of Theorem 3) It is easy to see how a complete closed browsing program can be simulated in the relational algebra by concatenating the simulations of the individual steps and starting with a table BT that has no columns and contains only the empty tuple.

However, this simulation fails if there is a step that has no result i.e. no nodes are added to the browsing tree. This is because in the browsing model the results of a step are always placed under the bottom layer nodes i.e. the result of the last step that was not empty, whereas in the simulation BT remains empty once it has become empty. This problem is solved by testing after every step if the result is empty. If it is not then the simulation of the following steps proceeds normally, but if it is then the simulation proceeds as if this step never happened. This is possible because the expression `if` e_1 `then` e_2 `else` e_3 with e_1, e_2 and e_3 algebra expressions, can be emulated in the relational algebra. □

An objection against PBM_2 might be that its transformations always act upon unary and binary relations. In the relational algebra the arity of the input relations is in general not limited. Furthermore, it is well known that relational tables of arbitrary arity can be coded in graphs as shown in Figure 12. Here the top node represents the relation with *element*-edges to the nodes that represent the tuples in the relation. The tuple nodes have edges that each represent a field of the tuple, labeled with the name of the field and ending in the value of the field. This leads us to the definition of PBM_3:

Input Domain An instance consisting of graphs of the form shown in Figure 12 with N the name of the p-ary relation, T a special tuple lable, $a_1, \ldots, a_p$ the (distinct) column names of N and $D_1, \ldots, D_p$ the corresponding (printable) domain types. Nodes with relation names are always uniquely identified by their label.

Output Domain The embeddings of the nodes in the bottom layer of the resulting browsing tree.

Transformations Those expressed by a closed browsing program.

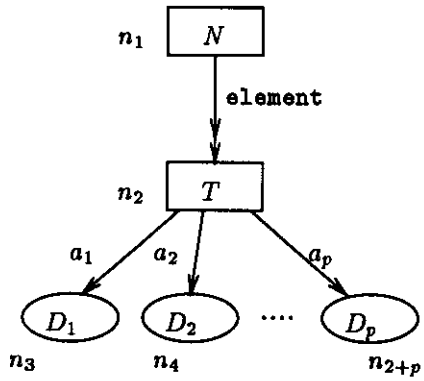


Figure 12: Pattern for representing simulated tables in layers

The input mapping im_3 maps an instance to a set of tables, one for every node with a relation name as its label. The relation will contain the tuples that are represented by the tuple nodes that they have *element*-edges to. These tuples contain exactly those fields that are labels of edges leaving the tuple node. The values of those fields are the values of the printable nodes that these edges end in. The output mapping om_3 is the same as om_1 and maps the embeddings of the bottom layer nodes to a table containing exactly these embeddings.

Theorem 6 *Under input mapping im_3 and output mapping om_3 it holds that PBM_3 is as expressive as the relational algebra.*

Proof: The proof is similar to that of Theorem 2. The first steps of the program will be the encoding of the relations in the layers. For this we can use the pattern that is given in Figure 12 followed by a step for projecting the relation node and tuple node away. This is done for every relation that is represented in the instance. Once the relations are encoded in the layers we can construct the rest of the closed program according to Theorem 1, for performing the algebraic expression upon these layers. □

References

[1] M. Andries, M. Gemis, J. Paredaens, I. Thyssens, and J. van den Bussche. Concepts for Graph-oriented Object Manipulation. volume 580 of *Lecture Notes in Computer Science*, Berlin, March 1992. Springer–Verlag. Proceedings of the third EDBT conference held in Vienna, Austria.

[2] T.J. Berners-Lee, R. Cailliau, J-F Groff, and B. Pollermann. World-Wide Web: The Information Universe. *Electronic Networking: Research, Applications and Policy*, 2(1):52–58, 1992.

[3] A. D'Atri and L. Tarantino. From Browsing to Querying. *Data Engineering*, 12(2):46–53, June 1989.

[4] A. D'Atri and L. Tarantino. A Browsing Theory and its Application to Database Naviagation. In J. Paredaens and L. Tenenbaum, editors, *Advances in database systems, implementations and applications*, number 347 in CISM Courses and Lectures, pages 161–179. Springer–Verlag, Wien, New–York, 1994.

[5] M. Gemis and J. Paredaens. An Object-Oriented Pattern Matching Language. volume 742 of *Lecture Notes in Computer Science*, pages 339–355, Berlin, 1993. Springer-Verlag. Proceedings of the international symposium on Object Technologies for Advanced Software; held in Kanazawa, Japan.

[6] M. Gyssens, J. Paredaens, J. Van den Bussche, and D. Van Gucht. A Graph-Oriented Object Database Model. *IEEE Transactions on Knowledge and Data Engineering*, 6(4):572–586, August 1994.

[7] J. Nielsen. *Hypertext & Hypermedia.* Academic Press, Inc., Dan Diego, CA, 1990.

[8] L. J. Pinson and R. S. Wiener. *An Introduction to Object-Oriented Programming and Smalltalk.* Addison-Wesley, Amsterdam, 1988.

Computing Science Reports

Department of Mathematics and Computing Science
Eindhoven University of Technology

In this series appeared:

93/01	R. van Geldrop	Deriving the Aho-Corasick algorithms: a case study into the synergy of programming methods, p. 36.
93/02	T. Verhoeff	A continuous version of the Prisoner's Dilemma, p. 17
93/03	T. Verhoeff	Quicksort for linked lists, p. 8.
93/04	E.H.L. Aarts J.H.M. Korst P.J. Zwietering	Deterministic and randomized local search, p. 78.
93/05	J.C.M. Baeten C. Verhoef	A congruence theorem for structured operational semantics with predicates, p. 18.
93/06	J.P. Veltkamp	On the unavoidability of metastable behaviour, p. 29
93/07	P.D. Moerland	Exercises in Multiprogramming, p. 97
93/08	J. Verhoosel	A Formal Deterministic Scheduling Model for Hard Real-Time Executions in DEDOS, p. 32.
93/09	K.M. van Hee	Systems Engineering: a Formal Approach Part I: System Concepts, p. 72.
93/10	K.M. van Hee	Systems Engineering: a Formal Approach Part II: Frameworks, p. 44.
93/11	K.M. van Hee	Systems Engineering: a Formal Approach Part III: Modeling Methods, p. 101.
93/12	K.M. van Hee	Systems Engineering: a Formal Approach Part IV: Analysis Methods, p. 63.
93/13	K.M. van Hee	Systems Engineering: a Formal Approach Part V: Specification Language, p. 89.
93/14	J.C.M. Baeten J.A. Bergstra	On Sequential Composition, Action Prefixes and Process Prefix, p. 21.
93/15	J.C.M. Baeten J.A. Bergstra R.N. Bol	A Real-Time Process Logic, p. 31.
93/16	H. Schepers J. Hooman	A Trace-Based Compositional Proof Theory for Fault Tolerant Distributed Systems, p. 27
93/17	D. Alstein P. van der Stok	Hard Real-Time Reliable Multicast in the DEDOS system, p. 19.
93/18	C. Verhoef	A congruence theorem for structured operational semantics with predicates and negative premises, p. 22.
93/19	G-J. Houben	The Design of an Online Help Facility for ExSpect, p.21.
93/20	F.S. de Boer	A Process Algebra of Concurrent Constraint Programming, p. 15.
93/21	M. Codish D. Dams G. Filé M. Bruynooghe	Freeness Analysis for Logic Programs - And Correctness, p. 24
93/22	E. Poll	A Typechecker for Bijective Pure Type Systems, p. 28.
93/23	E. de Kogel	Relational Algebra and Equational Proofs, p. 23.
93/24	E. Poll and Paula Severi	Pure Type Systems with Definitions, p. 38.
93/25	H. Schepers and R. Gerth	A Compositional Proof Theory for Fault Tolerant Real-Time Distributed Systems, p. 31.
93/26	W.M.P. van der Aalst	Multi-dimensional Petri nets, p. 25.
93/27	T. Kloks and D. Kratsch	Finding all minimal separators of a graph, p. 11.
93/28	F. Kamareddine and R. Nederpelt	A Semantics for a fine λ-calculus with de Bruijn indices, p. 49.
93/29	R. Post and P. De Bra	GOLD, a Graph Oriented Language for Databases, p. 42.
93/30	J. Deogun T. Kloks D. Kratsch H. Müller	On Vertex Ranking for Permutation and Other Graphs, p. 11.

93/31	W. Körver	Derivation of delay insensitive and speed independent CMOS circuits, using directed commands and production rule sets, p. 40.
93/32	H. ten Eikelder and H. van Geldrop	On the Correctness of some Algorithms to generate Finite Automata for Regular Expressions, p. 17.
93/33	L. Loyens and J. Moonen	ILIAS, a sequential language for parallel matrix computations, p. 20.
93/34	J.C.M. Baeten and J.A. Bergstra	Real Time Process Algebra with Infinitesimals, p.39.
93/35	W. Ferrer and P. Severi	Abstract Reduction and Topology, p. 28.
93/36	J.C.M. Baeten and J.A. Bergstra	Non Interleaving Process Algebra, p. 17.
93/37	J. Brunekreef J-P. Katoen R. Koymans S. Mauw	Design and Analysis of Dynamic Leader Election Protocols in Broadcast Networks, p. 73.
93/38	C. Verhoef	A general conservative extension theorem in process algebra, p. 17.
93/39	W.P.M. Nuijten E.H.L. Aarts D.A.A. van Erp Taalman Kip K.M. van Hee	Job Shop Scheduling by Constraint Satisfaction, p. 22.
93/40	P.D.V. van der Stok M.M.M.P.J. Claessen D. Alstein	A Hierarchical Membership Protocol for Synchronous Distributed Systems, p. 43.
93/41	A. Bijlsma	Temporal operators viewed as predicate transformers, p. 11.
93/42	P.M.P. Rambags	Automatic Verification of Regular Protocols in P/T Nets, p. 23.
93/43	B.W. Watson	A taxomomy of finite automata construction algorithms, p. 87.
93/44	B.W. Watson	A taxonomy of finite automata minimization algorithms, p. 23.
93/45	E.J. Luit J.M.M. Martin	A precise clock synchronization protocol,p.
93/46	T. Kloks D. Kratsch J. Spinrad	Treewidth and Patwidth of Cocomparability graphs of Bounded Dimension, p. 14.
93/47	W. v.d. Aalst P. De Bra G.J. Houben Y. Kornatzky	Browsing Semantics in the "Tower" Model, p. 19.
93/48	R. Gerth	Verifying Sequentially Consistent Memory using Interface Refinement, p. 20.
94/01	P. America M. van der Kammen R.P. Nederpelt O.S. van Roosmalen H.C.M. de Swart	The object-oriented paradigm, p. 28.
94/02	F. Kamareddine R.P. Nederpelt	Canonical typing and Π-conversion, p. 51.
94/03	L.B. Hartman K.M. van Hee	Application of Marcov Decision Processe to Search Problems, p. 21.
94/04	J.C.M. Baeten J.A. Bergstra	Graph Isomorphism Models for Non Interleaving Process Algebra, p. 18.
94/05	P. Zhou J. Hooman	Formal Specification and Compositional Verification of an Atomic Broadcast Protocol, p. 22.
94/06	T. Basten T. Kunz J. Black M. Coffin D. Taylor	Time and the Order of Abstract Events in Distributed Computations, p. 29.
94/07	K.R. Apt R. Bol	Logic Programming and Negation: A Survey, p. 62.
94/08	O.S. van Roosmalen	A Hierarchical Diagrammatic Representation of Class Structure, p. 22.
94/09	J.C.M. Baeten J.A. Bergstra	Process Algebra with Partial Choice, p. 16.

94/10	T. verhoeff	The testing Paradigm Applied to Network Structure. p. 31.
94/11	J. Peleska C. Huizing C. Petersohn	A Comparison of Ward & Mellor's Transformation Schema with State- & Activitycharts, p. 30.
94/12	T. Kloks D. Kratsch H. Müller	Dominoes, p. 14.
94/13	R. Seljée	A New Method for Integrity Constraint checking in Deductive Databases, p. 34.
94/14	W. Peremans	Ups and Downs of Type Theory, p. 9.
94/15	R.J.M. Vaessens E.H.L. Aarts J.K. Lenstra	Job Shop Scheduling by Local Search, p. 21.
94/16	R.C. Backhouse H. Doombos	Mathematical Induction Made Calculational, p. 36.
94/17	S. Mauw M.A. Reniers	An Algebraic Semantics of Basic Message Sequence Charts, p. 9.
94/18	F. Kamareddine R. Nederpelt	Refining Reduction in the Lambda Calculus, p. 15.
94/19	B.W. Watson	The performance of single-keyword and multiple-keyword pattern matching algorithms, p. 46.
94/20	R. Bloo F. Kamareddine R. Nederpelt	Beyond β-Reduction in Church's $\lambda\rightarrow$, p. 22.
94/21	B.W. Watson	An introduction to the Fire engine: A C++ toolkit for Finite automata and Regular Expressions.
94/22	B.W. Watson	The design and implementation of the FIRE engine: A C++ toolkit for Finite automata and regular Expressions.
94/23	S. Mauw and M.A. Reniers	An algebraic semantics of Message Sequence Charts, p. 43.
94/24	D. Dams O. Grumberg R. Gerth	Abstract Interpretation of Reactive Systems: Abstractions Preserving $\forall$CTL*, $\exists$CTL* and CTL*, p. 28.
94/25	T. Kloks	$K_{1,3}$-free and W_4-free graphs, p. 10.
94/26	R.R. Hoogerwoord	On the foundations of functional programming: a programmer's point of view, p. 54.
94/27	S. Mauw and H. Mulder	Regularity of BPA-Systems is Decidable, p. 14.
94/28	C.W.A.M. van Overveld M. Verhoeven	Stars or Stripes: a comparative study of finite and transfinite techniques for surface modelling, p. 20.
94/29	J. Hooman	Correctness of Real Time Systems by Construction, p. 22.
94/30	J.C.M. Baeten J.A. Bergstra Gh. Ştefanescu	Process Algebra with Feedback, p. 22.
94/31	B.W. Watson R.E. Watson	A Boyer-Moore type algorithm for regular expression pattern matching, p. 22.
94/32	J.J. Vereijken	Fischer's Protocol in Timed Process Algebra, p. 38.
94/33	T. Laan	A formalization of the Ramified Type Theory, p.40.
94/34	R. Bloo F. Kamareddine R. Nederpelt	The Barendregt Cube with Definitions and Generalised Reduction, p. 37.
94/35	J.C.M. Baeten S. Mauw	Delayed choice: an operator for joining Message Sequence Charts, p. 15.
94/36	F. Kamareddine R. Nederpelt	Canonical typing and Π-conversion in the Barendregt Cube, p. 19.
94/37	T. Basten R. Bol M. Voorhoeve	Simulating and Analyzing Railway Interlockings in ExSpect, p. 30.
94/38	A. Bijlsma C.S. Scholten	Point-free substitution, p. 10.

94/39	A. Blokhuis T. Kloks		On the equivalence covering number of splitgraphs, p. 4.
94/40	D. Alstein		Distributed Consensus and Hard Real-Time Systems, p. 34.
94/41	T. Kloks D. Kratsch		Computing a perfect edge without vertex elimination ordering of a chordal bipartite graph, p. 6.
94/42	J. Engelfriet J.J. Vereijken		Concatenation of Graphs, p. 7.
94/43	R.C. Backhouse M. Bijsterveld		Category Theory as Coherently Constructive Lattice Theory: An Illustration, p. 35.
94/44	E. Brinksma R. Gerth W. Janssen S. Katz M. Poel C. Rump	J. Davies S. Graf B. Jonsson G. Lowe A. Pnueli J. Zwiers	Verifying Sequentially Consistent Memory, p. 160
94/45	G.J. Houben		Tutorial voor de ExSpect-bibliotheek voor "Administratieve Logistiek", p. 43.
94/46	R. Bloo F. Kamareddine R. Nederpelt		The λ-cube with classes of terms modulo conversion, p. 16.
94/47	R. Bloo F. Kamareddine R. Nederpelt		On Π-conversion in Type Theory, p. 12.
94/48	Mathematics of Program Construction Group		Fixed-Point Calculus, p. 11.
94/49	J.C.M. Baeten J.A. Bergstra		Process Algebra with Propositional Signals, p. 25.
94/50	H. Geuvers		A short and flexible proof of Strong Normalazation for the Calculus of Constructions, p. 27.
94/51	T. Kloks D. Kratsch H. Müller		Listing simplicial vertices and recognizing diamond-free graphs, p. 4.
94/52	W. Penczek R. Kuiper		Traces and Logic, p. 81
94/53	R. Gerth R. Kuiper D. Peled W. Penczek		A Partial Order Approach to Branching Time Logic Model Checking, p. 20.
95/01	J.J. Lukkien		The Construction of a small CommunicationLibrary, p.16.
95/02	M. Bezem R. Bol J.F. Groote		Formalizing Process Algebraic Verifications in the Calculus of Constructions, p.49.
95/03	J.C.M. Baeten C. Verhoef		Concrete process algebra, p. 134.
95/04	J. Hidders		An Isotopic Invariant for Planar Drawings of Connected Planar Graphs, p. 9.
95/05	P. Severi		A Type Inference Algorithm for Pure Type Systems, p.20.
95/06	T.W.M. Vossen M.G.A. Verhoeven H.M.M. ten Eikelder E.H.L. Aarts		A Quantitative Analysis of Iterated Local Search, p.23.
95/07	G.A.M. de Bruyn O.S. van Roosmalen		Drawing Execution Graphs by Parsing, p. 10.
95/08	R. Bloo		Preservation of Strong Normalisation for Explicit Substitution, p. 12.
95/09	J.C.M. Baeten J.A. Bergstra		Discrete Time Process Algebra, p. 20
95/10	R.C. Backhouse R. Verhoeven O. Weber		Math∫pad: A System for On-Line Prepararation of Mathematical Documents, p. 15

95/11	R. Seljée	Deductive Database Systems and integrity constraint checking, p. 36.
95/12	S. Mauw and M. Reniers	Empty Interworkings and Refinement Semantics of Interworkings Revised, p. 19.
95/13	B.W. Watson and G. Zwaan	A taxonomy of sublinear multiple keyword pattern matching algorithms, p. 26.
95/14	A. Ponse, C. Verhoef, S.F.M. Vlijmen (eds.)	De proceedings: ACP'95, p.
95/15	P. Niebert and W. Penczek	On the Connection of Partial Order Logics and Partial Order Reduction Methods, p. 12.
95/16	D. Dams, O. Grumberg, R. Gerth	Abstract Interpretation of Reactive Systems: Preservation of CTL*, p. 27.
95/17	S. Mauw and E.A. van der Meulen	Specification of tools for Message Sequence Charts, p. 36.
95/18	F. Kamareddine and T. Laan	A Reflection on Russell's Ramified Types and Kripke's Hierarchy of Truths, p. 14.
95/19	J.C.M. Baeten and J.A. Bergstra	Discrete Time Process Algebra with Abstraction, p. 15.
95/20	F. van Raamsdonk and P. Severi	On Normalisation, p. 33.
95/21	A. van Deursen	Axiomatizing Early and Late Input by Variable Elimination, p. 44.
95/22	B. Arnold, A. v. Deursen, M. Res	An Algebraic Specification of a Language for Describing Financial Products, p. 11.
95/23	W.M.P. van der Aalst	Petri net based scheduling, p. 20.
95/24	F.P.M. Dignum, W.P.M. Nuijten, L.M.A. Janssen	Solving a Time Tabling Problem by Constraint Satisfaction, p. 14.
95/25	L. Feijs	Synchronous Sequence Charts In Action, p. 36.
95/26	W.M.P. van der Aalst	A Class of Petri nets for modeling and analyzing business processes, p. 24.
95/27	P.D.V. van der Stok, J. van der Wal	Proceedings of the Real-Time Database Workshop, p. 106.
95/28	W. Fokkink, C. Verhoef	A Conservative Look at term Deduction Systems with Variable Binding, p. 29.
95/29	H. Jurjus	On Nesting of a Nonmonotonic Conditional, p. 14