Discussion on Execution

09.03.2020, 03.09.2018 Arve Meisingset

This paper discusses execution of Existence logic within the Data transformation architecture. This means that we start with simple notions that are elaborated and modified throughout the paper.

This paper is not written as a tutorial. It contains a collection of discussions with myself. Hence, the paper contains suggestions, and not final conclusions. To summarize the results is left for another paper.

1 The role of schemata

In this paper, we will use the Data transformation architecture without presenting or explaining this architecture. See the book *Formal Description of Anything in the Universe*. We will focus on the Contents and the External Terminology layers.

This section outlines the overall execution of populations and schemata during search of data from the storage to the screen, and writing of data from the screen towards the internal storage. A section at the end of the paper will outline a practical implementation.

Any population of instances is controlled according to its schema. The schema reference from the population item is stated by its subordinate S, having a condition - <> - plus navigation. The Tip of the navigation refers to the schema item. This Tip is called the Original Tip, as we will insert more data - creating new Tips - under the Original Tip.

The schema contains the schema data, ie. the data classes that act as templates for the data instances in the population. The instances in the population are created by copying from the schema.

The S at the right hand side of the schema Condition, ie. the last of S<>S, acts as the root of the condition branch for the execution. The entire Contents schema is copied in under the Original Tip of this S. Data written by the end user, ie. the values, are put under the Tips of the copied schema data under this S. Anything written under the Original Tip of the S will be controlled according to what is found under the schema. Note the distinction between Original Tip and Tips.

Some Tips will contain copies of class labels – of entities, attribute groups, attributes, and roles. Other Tips will contain values. The values are leaves of the data tree. There will be functional dependencies between some of the values.

Immediately under the the Original Tip there may be a write Instruction which refers to the population. This write Instruction may be put in by default, and may not be written by some developer. The purpose of this Instruction is to rout the search results to the population.

Note that all Subordinate items are functionally dependent on their Superior item in the data tree. The values of the identifier attribute of a Subordinate entity are functionally dependent on the value of the identifier attribute of the Superior entity. This dependency may be stated by extra Conditions.

The entire contents under the rightmost S is used as a search specification through the External Terminology Population. We will use the same Contents schema to specify both the search and the form of the search result.

The Schema references allow for instantiating disjunctions of the options defined in the schema. The External Terminology Schema defines these options. The normalized statements in the External Terminology Schema are the options. Each Tip, including everything above in the data tree of the External Terminology Schema, is an option.

The Contents Schema will have to be supplied with additional information on constraints and derivations from the External Terminology Schema. Some of these constraints and derivations may conveniently be implemented as database procedures being referenced from the Distribution Schema. See a subsequent section on this.

We use the same Contents Schema both for search and responses.

For search, the end user adds values that have to be satisfied. All filled in values have to be satisfied simultaneously. The search specification is a data tree, which is executed from the root towards its leaves. The values are filled into the attributes, but will be perceived as conditions on their containing entities. If an entity satisfies its conditions, then this entity is included in the search result independently of its underlying entities in the tree. Hence, if an entity is included, then all its recursively superior entities in the tree are included. This is a kind of conjunction of everything in a path. However, we do not support commutability, ie. Not a AND b = b AND a. Our conjunction is an ordered sequence of conditions, ie. IF a, THEN IF b, THEN IF … etc. This means that if a condition is satisfied, then all the superior entities are included in the search result up to the previously conditioned entity. If there is no condition in a branch, then all entities in that sub-branch are included.

This means that if there are search values on several of the attributes of an entity, then they all have to be satisfied simultaneously. If Eye colour (Blue and Sex (Girl, then the Person with these characteristics is included in the search result. This is a classical conjunction, ie. a logical AND.

If several values are filled into one attribute, eg. Eye colour (Blue, Green, then they are considered to be alternatives. Hence, only one of the values need to be satisfied for each entity. Therefore, this is a disjunction. If the attributes are single valued, this is an Exclusive OR.

We observe that we search disjunctions of paths, and require conjunctions within each path. This is similar to a disjunctive normal form in classical logic.

The data tree for search, ie. when stating S<>, may consist of several branches. Each path is executed and satisfied independently of the other paths of entities. The data tree for simple Conditions, ie. <> without S-es, states that the entire conjunction of paths has to be satisfied. This is different from the S function S<>.

We will deal with negations in a subsequent section. We will not discuss other logical operations, but they may easily be implemented at the user interface, like in Query-By-Example.

During a write operation, the end user may fill in a whole screen with values.

During search, all constraints on an entity have to be satisfied, including the attribute values.

Hence, for search we need a two-phase execution.

- Find a complete path according to the Contents Schema – with sub-branches - that contain an entity instance that satisfies all its constraint values.
- Copy this entity, all its superiors, and all its sub-branches – according to the Contents Schema – that radiates out from this entity of this path. We may include property attributes of the superior entities, but not anything on their related entities. The sub-ordinates may be found by navigating via references in the External Terminology Population.

  Repeat this two-phase execution – from the current to the next path, until the entire population data tree has been parsed.

Note that many of the additional paths may only have separate sub-paths from the previous path, eg. a first path starts with a country at its root and then a person at its Tip; the next path uses the same country and finds another person, etc. The person sub-branches are different from each other, while the country root is common for several sub-paths.

For each path there may be many radiating sub-branches – according to the schema – that have no contents in the population, eg. some persons may have loved-persons, others have not.

A write instruction - >< - plus navigation will copy the found data into the Contents Population. If we want data about both the country and the persons, then the write Instruction must be stated directly under the Original Tip, ie. Above the country and persons..

The Contents Population is constructed by combining data from the search under the Contents S with data from the External Terminology Population.

The Contents Schema is used to create the form that the end user will see on the screen. The search data are filled into this screen, and after the execution, the screen will show the search result. The end user inserts data to the External Terminology Layer via the same screen combined with write instructions. The write instruction may come implicit with the state of the dialogue, or it may be stated explicitly by the end user.

The story on instantiation is more complex for the External Terminology Schemata and their External Terminology Population. This schema may contain conditions, write instructions, subordinate values and additional recursive schema references, eg. to value types. All these additional information will be validated and executed during insertion of data into the External Terminology Population. Also, the External Terminology Schema may contain data having ‘negation’ tags, which will hinder the corresponding data to be inserted into the External Terminology Population.

The constraints and derivations within the External Terminology Layer may go outside the scope of the Contents Population. The execution of the additional information will take place on the instances in the External Terminology Population as an impact of write Instructions. The navigation may go across branches in the External Terminology Population data tree and outside the scope of the data in the screen picture.

The Contents Schemata and Populations may include conditions and navigation as well, but these will appear as subordinate data in each branch, and will not navigate across branches. They specify navigation in the External Terminology Population. The Conditions in the Contents Layer specify navigation only, and do not instruct execution of the referenced Conditions in the External Terminology Layer.

The mapping between data in the population and data in the schema needs to be explained. The mapping is homomorphic, ie. many to one. This means that many instances in the population may map to the same class item in the schema. Hence, the Condition in the S reference is testing an n:1 correspondence. This means that many branches in the population may map to the same branch in the schema. The Condition tests if there is something in the schema that apply in each instantiation.

Ordinary Conditions, ie. non-S Conditions, require that everything - as a collective under the Condition branch - is found under the controlled branch. The entire sets of paths have to be satisfied from the Condition to the Tip-s. This is different from the S Condition, which only requires that each path contained under the conditioned item is found in the referenced schema.

The Contents Schema may navigate across and glue together many paths in the External Terminology Schema. Likewise, a path in the Contents Populations may connect many paths in the External Terminology Population.

The navigation in the Contents layer will hold all relevant information about this structure of the External Terminology Layer, and the used name tags are identical in both layers.

We will compare the two-phase execution with the programming construct IF ... THEN ... .

In the IF part we test if a condition is satisfied; in the THEN part we carry out the instruction. The data to be tested and written are stored somewhere else.

In Existence logic we also distinguish between Conditions and Instructions, but they are intervened in the same branches; and the data to be tested and written may also be found here.

The data tree in Existence logic is comparable to a Turing tape, where data and programme instructions are found on the same tape.

In Existence logic, instructions in the External Terminology Schema are copied into the External Terminology Population and executed among its data instances. In practical implementations, the instructions may not be physically stored among the data instances, but this is another story.

2 Execution architecture

The previous section tells that the execution requires

- A prescription, eg. found in a schema
- A data source, on which the execution operates, eg. a population
- A search and instruction store, eg. under the Tip of the schema reference S

The prescription can be found within a Condition and its Instructions, eg. under the schema reference. Hence, bullet one and three may have similar roles. Bullet three may be built by using bullet one with the addition of end user data for the search and editing..

The data source to be controlled, modified and extracted – bullet two -may be found in branches of the main data tree.

During a search, the control head may find data that partially satisfy the Condition. Only when fully complying with the Condition along a path, may the data be included into the search result. As an example, the found entity is a <u>Person</u>, but it has not the sought after Name John. Therefore, the non-John-s will have to be abandoned.

Sometimes, the condition branches will contain several Tip-s that all have to be satisfied for an Instruction to be carried out. Either we need a working memory to collect the Tip-s, or we have to execute the condition branch twice, first to control the contents, thereafter to collect the information from the controlled branch. We choose the last option with a two-pas execution. First we control that the entity satisfies the Condition; next we retrieve the entity and its contents.

If the Person with particular characteristics is contained in an unspecified Family that is contained in a specific Country, then we may first find the specific Country, then search all Family-s in this Country, and then find the Person-s with the particular characteristics in each Family of that Country. Hence, the search result will show one Country, all its Family-s and only those Person-s satisfying the requirements.

During the first phase of the execution, we need to return to a write Instruction or a Condition in order to know that the needed part of the condition branch has been covered. Alternatively, we execute per entity.

The second phase of the execution need only to cover the sub-branch beneath an Instruction or beneath the entity containing the last required value.

There may be more than one control branch that satisfy the Condition, eg. that the Condition applies for every Person. Then we have to execute the condition branch, or a sub-branch of it, repeatedly. If we move information from the controlled branch into the condition branch, we will have to tag and remove the extra information before a new run. Preferably, the search result is moved into another branch, eg. into an Instruction branch or write branch. We will choose to move the data into the write branch. Hence, we will need minimum one Instruction in every condition branch that may result in several result branches. These Instructions may be inserted by defaults, eg. for each entity.

Note that a simple Condition referencing an identifier, ie <>, is unique, ie points out one particular entity instance. In this case, no write Instruction is needed. We just control that the Condition is satisfied. However, when stating a search, ie. S<>, in the Contents Schema, we always need a write an Instruction in order to move the result to the Contents Population.

When executing a Condition,

- a read head is needed and it will be put into the condition branch
- a control head is needed to parse the controlled branches
- a second run may be needed if the Condition is satisfied from a write Instruction/entity down to a Tip
- in the second run, additional data are collected from the controlled head, eg. you fetch values that are not included in the condition branch
- The instruction head instructs a write head to step to the right position
- The Instruction may result in write operations
- The mapping from the condition branch to the instruction branch is often one-to-one from each controlled head to the write head,
- but may also be many-to-one, performing a function mapping

- The search result from the control head is transferred to the write head
- that is placed at the right data item under the write head before the transfer

In this outline of the execution, we have assumed

- that the control sub-branch is traversed twice, first doing test and then doing copying, in order to avoid transfer of not- valid data that would require a clean up
- that both the condition branch and the instruction branch are kept clean, and are not used for transfer of data
- that the condition and instruction branches together state a Function mapping
- that the instruction branch is executed, instructing the write head to the right position, before copying of data from the control branch can take place
- that data from the control branch are transferred directly to the write branch

Execution of Conditions require use of two synchronized read-write heads – ie. a read head and a control head.

Execution of Instructions requires use of two synchronized read-write heads, as well – ie. of the instruction head and the write head.

Copying requires use of three read-write heads – ie. of the read head in the condition branch, the control head in the control branch, and the write head in the write branch.

The read head in the condition branch controls the control head in the control branch. The read head controls the control head up to a Next data item that matches the Condition. This process can be explained the other way: the control head advances to Next data items until it finds a data item that matches with the data items under the read head.

If the control head in the second phase finds data subordinate to its Tip-s of satisfied data, it executes these and copy them to the write head. The condition may only refer to attributes, but this copying fetches the unknown values beneath the attributes.

The execution beneath the Tip-s is not controlled by the read head in the condition branch. Hence, even if three heads are involved above the Tip-s, only two synchronous heads are needed beneath the Tip-s.

One-to-one copying from the control to the write head corresponds to a search, where the write head produces the search result; it does not find the result.

The S Condition, ie. S<>, may produce many branches or sub-branches that matches the Condition. Therefore, the execution has to continue after a satisfying sub-branch is found, until all data are tested in the appointed control branches.

If the data item under the control head contains a schema reference to a function definition, ie. to a many-to-one mapping, a look up is made to the arguments found under the Tip-s in the control branch and the combination of these is replaced by the function value under the write head.

If the Condition is satisfied by many sub-branches, the Instruction in the condition branch will typically match each sub-branch, and produce a new instruction and write branch and function value for each.

Note that the write Instruction may not create a completely new branch. Parts of the Instruction or the satisfying control branch may match one existing write branch, and new data are copied into this. We have assumed that the copying is 1:1. Copying 1:n may be for further study. Also, there may be search operations in the write branch up to the right data item. This may need further study, as well. Maybe, we will need a two phase execution for the write Instruction, like for the condition branch. If the write Instruction is placed near the Tip-s of the condition branch, most of these challenges may disappear.

Before we go into the details of the execution, we will study the notions of negation, recursion and functions in more details.

3 Negation

We may try to introduce negation as a Condition that fails. We need no special symbol for negation. We can use anything that fails, but for convenience, we use an exclamation mark - ! -.

If we state

```
Colour (: <>  :
```

This condition will always be satisfied, because we have identical data items - : - on both sides of the Condition - <> -.

Also,

```
Colour (Red <>  :
```

is always satisfied, because any data item, eg. Red, has a colon - : - at its root.

PS: We will later abandon the deletion following from a Failed Condition, when a (sub-ordinate) Condition is immediately followed by an Instruction.

If we state

```
Colour (Red <> !
```

it will always fail when the Colour contains the value Red, because the exclamation mark - ! – is different from Red. The impact is that Red will be deleted.

Also,

```
Colour (: <>  !
```

will fail, because the colon - : - does not contain the entire exclamation mark - ! –. The colon will be deleted.

If we want to state that a Colour cannot be Red, we may state:

```
Colour (: <>  Red <> !
```

The colon indicates any value. The first Condition states that the value shall be Red, ie. it is satisfied if it is Red, which it is not. However, if the colon is replaced by Red

```
Colour (Red <>  Red
```

then the first Condition is satisfied.

The second Condition is not satisfied. Hence, the second Red is deleted, and the first Condition states that the left hand side shall be 'equal' to nothing, which it is not. Hence, this expression is not useful.

PS: If Failed Conditions do not result in deletion, this argument is no longer valid. Main Failed Conditions without Instructions may result in deletion.

The problem with the above formulations of negation is that the negating Conditions delete themselves. Therefore, we need a way to delete data items at another place in the data tree.

We already have a write Instruction to copy and insert data at another place. We need a tag stating that if the current data item is inserted, eg. : is overwritten (by/from the end User) by Red, it shall be deleted. We can add the tag at the appropriate place in the Condition statements.

In our fundamental (three-dimensional) language, we have two unused directions:

- Previous - ; - of the first item in a list
- Superior – " – of non-first items in a list

In order to ease the reading, we use the exclamation mark - ! – for this purpose.

If the first item of every list contains the name tag of the Superior data item, then it does not make any sense to negate the name tag. Hence, we are left with the directly Superior, which we write as

```
" ' ( "
```

on any non-first item in a list of Next items to state negation - ! -.

The expression ‘ (appears already at the Top of any condition branch. If it appears at any other item than such a Top, then ‘ will mean the Superior of the first item in this list, and the (will mean down to whatever appears after this parenthesis. We do not want the superior of the first, but the directly superior of any non-first item. This we indicate with the enclosing quotation marks.

The directly Superior of eg. the seventh item in a list of Next items will look as follows:

: : : : : : : “ ‘ (“ : :. This is equivalent to stating : : : : : : : ! : :.

In a three-dimensional notation, this list appears as

```
      :
: : : : : : : : :
```

The upper colon is a tag on the seventh item in the list. This tag appears within a condition branch, and is parsed by the read head controlling the control head.

When having tested everything contained in the seventh item under the control head, we have under the read head to test if this item has a previous item, ie. that it is non-first, and has a directly superior item. Note that this negation item will not appear under the control head.

We may additionally test if the negation item has no Superior, no Next and no Previous. This verifies that this is a negation tag. We will not discuss the option that the item is connected to a superior list, creating loops in the graph. We will not discuss lattices and knots in this paper.

The very first item in a list is always connected to a Superior item, except the Top of a condition branch that starts with navigation to Superiors. The test of negation on any other item includes a test of two connections, Previous and directly Superior. This is a test of a structure within the graph, and is therefore not a primitive operation. It is the introduction of a first letter - ! – in a specialized alphabet.

If the correct contents of the seventh item is found under the control head, and the negation item is found under the read head, then the Condition has Failed in this part of the control branch. Any other contents of the seventh item will pass as a success, ie. the Condition is Satisfied.

That the Colour cannot be Red is stated as

Colour (Red !

This means that if the control head finds Red under Colour, then the Condition – if not Red - is not satisfied, and Red will not be accepted.

The negation will apply when selecting a search result and when writing this result.

Note that there is no such thing as a double negation, as the negation does not function as a logical operator or connective. However, we may have Conditions within Conditions, and they will all have to be satisfied.

The negation in the condition branch results in removal of the tagged item in the search result.

The control branch may contain its own negation tags. These will be copied into the write branch.

4 Recursion

The recursion symbol - & - tells that the previous symbol may be repeated an unspecified number of times. Hence, & is a particular kind of wild card.

The schema notion may be used as a kind of Go-to that may be used to create recursion. You make a schema reference to a Superior or previous data item, eg. inside a condition branch. Then the data items between the referenced schema item and the schema reference may be repeated as many times as wanted in the actual instantiation. The contents of the schema are classes relative to the instances created during the recursion.

Note that the schema reference states an option, and may not be used. A Go-to in programming languages is a forced move. The use of the recursion, and how many times to be used, is up to the user or usage.

A Next data item - , - containing (() a schema reference - S<> - to its Previous item - ; - may be stated as follows:

```
    , (S <> S ‘ ;
```

The single quotation mark - ‘ - indicates the superior of the S. Comma means Next, and semi-colon means Previous. The above text explains how the recursion symbol functions for the Next data items.

In practical implementations we do not always need to store all the generated data items from recursions; we simply allow the execution to go into a loop. We do not deal with efficiency and clean up-s in this paper.

5 Functions

Let us define a list of Digit-s:

```
    Digit (0, 1, 2, 3, 4, 5, 6, 7, 8, 9
```

Note that we create a definition of Digit by listing its contents. In definitions, we do not do assignment by use of the equality sign, like Digit =. Digit is not a variable that is replaced by the values; Digit is an attribute that contains the values. We do not use variables.

The above list will be used as a class, and its constituent numbers are classes, as well. The whole or a part of the list may be instantiated into sub-lists or into an individual Digit, eg.

```
Digit (6
```

If the class Digit is defined anywhere, it may be referenced from anywhere - : - by the following expression:

```
: (S <> S ‘& (& Digit
```

The recursion operators indicate an unspecified number of levels up – ‘ – and down – (- to Digit.

Hence, the first colon - : - is an attribute that has the schema Digit, which contains the number 6 among other numbers. The value of the attribute is instantiated from 6 under Digit:

```
: (S <> S ‘& (& Digit 6
```

Note that the parenthesis in front of & needs no closing parenthesis. The blank in front of the 6, without a comma, indicate that the 6 is at the same level as the first S. This may be written as:

```
: (6, S <> S ‘& (& Digit
```

However, we may want the schema reference to be executes first, and therefore not use this expression. An alternative formulation is as follows:

```
: (S (<> S ‘& (& Digit), 6
```

The closing parenthesis after Digit corresponds to the opening parenthesis in front of <>.

The opening parenthesis after colon - : - tells that 6 exists at the same level as S, under the colon. The parenthesis in front of <> is not needed in the previous expressions, which are simplified notations. Note also that we do not always write closing parentheses.

We may use the recursion expression - & - in the schema, because we do not know exactly how the referenced class will be referenced in the population. When we know, the recursion will be replaced by a concrete path. This path may be stated by the developer, the end user or automatically at generation.

I have been wondering if the navigation, in some cases may be removed altogether. The effect would be that a data item may be connected via a Condition to a data item in some other branch. This would create tight connections and loops between branches. There may be some uses of this feature if we implement an automaton by using Existence logic itself. If we allow for such direct connections between any two nodes in the data tree, we may still need navigation to establish this connection.

We use the expression ‘& (& to refer to a data item anywhere. This is similar to using global names of constants or variables. However, global names indicate that the data items are observed from the outside. Our navigating references are made from a particular data item inside the data tree to another data item somewhere in the same tree.

A function is a mapping from a set of arguments, x1, x2, …, to a function value, y, eg.

y=f(x1, x2, …)

We will explain this idea using the new notation.

In a meta-schema, SCHEMA 5, we define a Digit, and the signature of a function IncOp. The function takes Arg as arguments and gives Func as a function value. Both the arguments and the function value takes Digit as their domain. We use a schema reference - S - to refer to a Digit in SCHEMA 5.

For assignment and placement of the function value, we use a write operator - >< - together with a navigation path. The write operator is the opposite of the condition operator - <> -.

```
:                         SCHEMA 5
  :                       Digit (0, 1, 2, 3, 4, 5, 6, 7, 8, 9
  :                       IncOp
       :     <>           Arg (S <> S ‘Arg ‘<> ‘IncOp ‘SCHEMA 5 (Digit
             ><           Func (S <> S ‘Func ‘>< ‘IncOp ‘SCHEMA 5 (Digit
```

We have an extra Condition within the condition branch, and use ‘<> to backtrack over this Condition. We have an extra Instruction within the instruction branch, and use ‘>< to backtrack over this Instruction.

In the above schema, the arguments Arg are found in a condition branch, and the function Func value is found in an instruction branch. This is not what we want. We want the arguments to be found somewhere in the main data tree, and the function value to be delivered somewhere else, as well. Furthermore, for the Condition to be satisfied, the Arg has to be found under the conditioned item in the main data tree. Likewise for the Func. Hence, we would have to write Conditions and Instructions to take and deliver the values to these data items.

The reason for the above difficulties, is that Arg and Func are constants having local name tags, and we do not support variables with global name tags that may reference by names across the data tree. Hence, we replace Arg and Func by arbitrary navigations, ie. ‘& (&. These include navigation over Conditions and Instructions.

```
:                         SCHEMA 6
```

```
      :                    Digit (0, 1, 2, 3, 4, 5, 6, 7, 8, 9
      :                    IncOp
           :    <>         '& (& (S <> S  '& (& '<> 'IncOp 'SCHEMA 6 (Digit
                ><         '& (& (S <> S  '& (& '>< 'IncOp 'SCHEMA 6 (Digit
```

We use ‘& to backtrack over (&, and (& to backtrack over ‘&.

In a second meta-schema, SCHEMA 7, we instantiate the IncOp function, where the values are taken from Digit. We will use ‘: in the schema reference to refer to the common root above SCHEMA 5 and SCHEMA 7.

```
 :                         SCHEMA 7
      :                    IncOp
           S    <>         S 'IncOp 'SCHEMA 7 ': (SCHEMA 5 (IncOp
           :    <>         Arg (0
                ><         Func (1
           :    <>         Arg (1
                ><         Func (2
           :    <>         Arg (2
                ><         Func (3
           ….
           :    <>         Arg (8
                ><         Func (9
           :    <>         Arg (9
                ><         Func (0
```

Note that Arg (0 is the value of Arg. We may replace 0 by empty, ie. Arg (.

Now, we want to refer to SCHEMA 6 rather than SCHEMA 5.

```
 :                         SCHEMA 8
      :                    IncOp
           S    <>         S 'IncOp 'SCHEMA 8 ': (SCHEMA 6 (IncOp
           :    <>         '& (& (0
                ><         '& (& (1
           :    <>         '& (& (1
                ><         '& (& (2
           :    <>         '& (& (2
                ><         '& (& (3
           ….
           :    <>         '& (& (8
                ><         '& (& (9
           :    <>         '& (& (9
            ><          ‘& (& (0
```

As you will see in SCHEMA 9, ‘& (& will be replaced by specific references during instantiation. Use of recursion may appear at many places, and makes the references unspecific. They may be replaced by a kind of variable, like x and backtracking via ‘x. A practical language may do exactly this. However, we want to explore what we can achieve without variables.

Replacing different instances of ‘& (& by x, y etc. will be used of variables as wild cards, and is not the same as replacing values of attributes by a variable, which also may be introduced. We use (: for an unspecific value. Use of variables of both kinds will imply use of a rewriting grammar. We want to explore use of attachment grammars only.

In the schema of the application, ie. SCHEMA 9, we define a Product. If the Product is inserted (by the end user), this is represented as a Condition with the insertion command I on the Product. The Condition is on the Product; hence, the Product and everything in it is executed only when the Product is inserted.

The same schema has a global attribute, # products, that counts the number of Products in the application. This number is incremented by one, whenever a new Product instance is inserted. This update is performed by the Inc function.

The Inc function takes SCHEMA 9 (IncOp as its schema. Note that we write ‘: for backtracking over SCHEMA 9, because this backtracking will be instantiated into its population, which will have another name tag than of this schema.

The Inc function takes its value via the Condition from # products and gives its function value via an Instruction to # products.

The Inc function looks up in its schema, and finds the matching component of IncOp, ie. the matching argument value. It looks up the assigned function value, and instantiates this into the function value of Inc, which refers to # products.

products takes Digit as its domain. See its schema reference.

```
:                       SCHEMA 9
    :                   Product
        <>  I           (nserted)
        :               Inc
            S   <>      S ‘Inc ‘Product ‘: ‘: (SCHEMA 8 (IncOp
            :   <>      : ‘Inc ‘Product ‘: (# products
                ><      : ‘Inc ‘Product ‘: (# products
    :                   # products
        S               <> S ‘# products ‘: ‘: (SCHEMA 6 (Digit
```

We observe that despite having had some clutter with stating the meta-schemata, the schema of the application becomes simple. SCHEMA 9 is what an application developer typically will write.

The S reference refers to the definition of IncOp in SCHEMA 8. The colon - : - beneath the S is an instance of the colons beneath the S in SCHEMA 8. In SCHEMA 8 the references use wild cards. In SCHEMA 9 they are replaced by concrete references to # products.

The developer should be cautious about using wild cards in application schemata, like SCHEMA 9, as he may refer to unintended data satisfying the Condition.

We may now define a POPULATION at the same level as SCHEMA 9, and state a schema reference to SCHEMA 9. The users may insert a list of Product instances, and the above calculations will be executed on each.

We have shown the principles of specification and execution. In practical applications, we may defer the specification of logic – of the Inc function - to the Distribution layer, which will not be addressed here.

In the above discussion, we have defined function applications between number symbols. We have not defined the 'meaning' of these symbols. This we may do by defining the number symbols to be name tags of lists:

```
1 (:, : !
2 (:, :, : !
…
9 (:, :, :, :, :, :, :, :, :, : !
```

In the definition of each number, we have added an extra data item - : - with a negation - ! -. This ensures that the last item of each list shall not appear, eg. that 2 contains exactly 2, and not 3 data items. The definitions may be used to test if a list has exactly n items, and not n+1 items.

6 Arrangement of data

We assume that the lists at every level of the data tree are arranged as follows:

- The first data item of any list contains the name tag of its superior data item
- The next data item contains the identifier attribute of the superior data item/entity
- The subsequent data items contains the other attributes of their superior data item/entity
- Finally, the subordinate entities of the superior data item/entity are listed

The above arrangement of data items is used in every condition branch after navigation to a common superior.

This arrangement will have the following effects on the execution of any condition branch:

- The name tags will be tested first
- Second, the identifier attribute is tested at every level
- If a list contains any subordinate entities, they will appear last in a list of Next items.

Note that several data items may have no Next data item in a sub-branch.

Any sub-branch may contain Subordinate values that will constrain the search.

Often, the navigation from a Condition starts by ascending recursively via Superior data items to a Top that is common for the conditioned item and the control item. There is always only one Superior. When descending from the Top in the condition branch, there may be several Subordinate data items under the control head that satisfy the navigation path under the read head in the condition branch. When backtracking from a Top in the condition branch, the execution will be descending via Subordinates to the Condition.

If the condition branch is only on Subordinate data of the conditioned data item, then the Top of the condition branch is the data item having the Condition as its Instruction, ie. the Root of the condition branch.

If a Function is involved, then it may take its arguments from all the Tip-s. However, some of the Tip-s may only contain values used for the navigation. The Function defines which Tip-s to be used as arguments. These arguments are typically the Ultimate tip and its previous Tip-s. So, arguments are placed last in the lists, while identifiers come first. Some functions may take arguments from several branches.

7 Execution of search

The Schema references allow for instantiating disjunctions of the options defined in the schema. The External Terminology Schema defines these options.

We do not distinguish between questions and answers; we use the same form for both. We take a Contents Schema and add search values. The same Contents Schema tells what data classes will appear in the answer.

A typical Condition - <> - looks as shown within the following expression:

1. <u>Country </u>(Name (NORWAY), <u>Person</u> (Name (JOHN), <u>Loved-person</u> (<> <u>Loved-person</u> /‘ ‘ (<u>Person</u> (Name (MARY

This Condition states that JOHN has a <u>Loved-person</u> that refers to MARY. The reference to <u>Person</u> (Name (Mary is shown, while the referenced <u>Person</u> (Name (Mary is not shown. The left hand parenthesis in front of the Condition is optional.

Each Contents Schema will be used as a Condition. The Conditions within the Contents Schema itself will not state new constraints; they will navigate over Conditions within its External Terminology Schema.

- This search will find only one <u>Country</u> (Name (NORWAY, only one of its <u>Person</u>-s (Name (JOHN, and only one of its <u>Loved-person</u>-s that refer to <u>Person</u> ()Name (MARY

Any Contents Schema will be copied under the Original tip of a Condition. The Contents Schema may look as follows:

2. <u>Country (</u>Name, <u>Person</u> (Name, <u>Loved-person</u> (<> <u>Loved-person</u>
/' ' (<u>Person</u> (Name

- This search will find all <u>Country</u>-s, all <u>Person</u>-s of each, and all of their <u>Loved-person</u>-s

The end user inserts some values into the search specification, eg. like this:

3. <u>Country (</u>Name (NORWAY), <u>Person</u> (Name, <u>Loved-person</u> (<> <u>Loved-person</u>
/' ' (<u>Person</u> (Name (MARY

Every Contents Schema is a tree. The example <u>Country</u> is Subordinate to the root of the tree.

- This search will find only one <u>Country</u> (Name (NORWAY, all of its <u>Person</u>-s, and only those of its <u>Loved-person</u>-s that refer to <u>Person</u> (Name (MARY

The two single quotes - ' ' - refer directly up to the current <u>Person</u> and the <u>Country</u> (Name (NORWAY) without going via Previous items. The subsequent expression (<u>Person</u> (Name (MARY)) in the same <u>Country</u> is going down to <u>Person</u>, Name and MARY without indication of intermediate Next items within each list. This search may go via several <u>Person</u>-s in the list under <u>Country</u>, before arriving at MARY. Also, the search may pass <u>Dog</u>-s and <u>Car</u>-s in the same list, before arriving at the right Person. Finally, the search may find several items that satisfy the search.

We will indicate recursion of an operation by the & symbol. ;& means repeating Previous. ,& means repeating Next. Hence, a longer version of the first formula may read as follows:

4. <u>Country (</u>Name (NORWAY), ,& <u>Person</u> (Name, ,& <u>Loved-person</u> (<> <u>Loved-person</u>
/;& ' ;& ' (,& <u>Person</u> (Name (MARY

- This search is the same as of 3.

Note that the additional operators to example 3 give instructions to the control head. Example 3 shows that these may be omitted, giving the same result.

In the Condition, in the first line, the control head arrives at the current Loved-person. This may not be the first Loved-person under Person. Hence, we step via Previous items ;& to the first in the list. Then we go up to Person by ‘. Then to the first in this list by ;&. Thereafter to the Top by ‘. Then down by (. Thereafter to Next items by ,& until we find a Person (Name (MARY.

Here we have assumed that each Name appears as the first attribute under each Person. Hence, we do not need to use recursion to find the first attribute. Except for this, we have to iterate to the Next items - ,& - in every sub-list, and iterate via Previous items - ;& - to the first item before going up.

In practical implementations, we may use direct access without use of these recursions. We will come back to this in a final section.

In addition to the shown iteration to any Previous item - ;& - and any Next item - ,& -, there in some cases will be a need of iteration to any Superior item – ‘& - and any Subordinate item – (& -. In the above formulas, we have assumed that navigation to Next and Previous items always involves iteration - & -, while navigation to Superior and Subordinate never involves iteration, if not explicitly stated.

The implicit iterations as of the first formula in this section, and explicit iterations as of the last formula, will result in the Condition requesting homomorphism, ie. one Condition may be satisfied by data in several control branches, as of the responses to the following query:

5. Country (Name (NORWAY), Person (Name (JOHN), Loved-person (<> Loved-person
   /‘ ‘ (Person (Name

- This search will find only one Country (Name (NORWAY, only one of its Person-s (Name (JOHN, and all of its Loved-person-s

The External Terminology Schema may look as follows:

6. Country (Name, Person (Name, Loved-person (<> Loved-person
   /‘ ‘ (Person

Here the Condition refers to a unique class Person, so no reference to its Name is needed in the schema, but will be needed in its populations.

The developer adds the Name-s in the Contents Schema:

7. Country (Name, Person (Name, Loved-person (<> Loved-person
   /‘ ‘ (Person (Name

- This search is the same as of 2.

The Name in the Condition, ie. the last entry above, is found under Person in the External Terminology Schema.

The end user may formulate the following searches:

8. Country (Name (NORWAY), Person (Name, Loved-person (<> Loved-person
/' ' (Person (Name (MARY

- This search is the same as of 3.

9. Country (Name (NORWAY), Person (Name, Loved-person (<> Loved-person
/' ' (Person (Name

- This search will find only one Country (Name (NORWAY, all its Person-s, and all the Loved-person-s of each.

10. Country (Name, Person (Name

- This search will list all Country-ies and all the Person-s in each.

11. Country (Name (NORWAY), Person (Name, Sex (MALE), Category (CHILD)

- This search will list only one Country (Name (Norway, ONLY its Person-s having the Sex value MALE and the Category value CHILD.

When adding a write insertion, eg. associates I to a value, into a Contents Population, this Population will be transformed to a write Instruction - >< - into the External Terminology Population, and the value will be attempted to be inserted.

In these examples, the External Terminology Population will look exactly like the Contents Population. The permissible values, NORWAY, JOHN and MARY are found under the respective Name-s in the External Terminology Schema. They may be found via extra meta-schema references to value types that state the permissible values.

When we in a Condition from an arbitrary item X write – X ' -, we navigate from the X item to its Superior. - X ' ' -means the Superior of the Superior, etc. The blank between the two '-s means an unnamed entity. We could as well have written - X ':': -. The colon - : - indicates any entity, which here is the Superior. There is only one Superior item. Hence, the specification is unique, despite that the name tag is not indicated. The colons - : - may be replaced by the appropriate name tags.

The Superior item may be found by repeated Previous moves - ;& -, to the (first) item, and then to the direct Superior - ‘ -. An alternative to these back steps is to log all Superiors when we were navigating down to X. We will come back to the logs in the subsequent section.

When, for the first time, navigating down via a path – (Y (Z (U … (W – etc., we have no log from previous runs, but we may create logs when going down.

We are going down to the leaf item W first, by taking Subordinate-s – (- before Next-s - , -. However, since Z is local to Y, U is local to Z etc., there may be many W-s. Therefore, we will put the name tags first in any list. Hence, we will check the name tag at each level before going to a Subordinate level. This way, we find a Y before we find a Z, a Z before we find a U, etc.

There may be many Y-s. Hence, we know that we have reached the right Y when its subordinate data also match. At any level, we will need to check the subordinates to ensure that the current item is right.

When a mismatch is found, we recursively have to do Previous - ; - and Superior -) -, then find the Next - , -, and go down again, recursively, until the Control head finds a full match at all levels given by the Condition.

There may be more than one path satisfying the Condition. Hence, the Control head is forced through all these paths. The same applies for the Instructions. In principle, the control head has to parse the entire data tree in order to find all branches satisfying a search. By sorting and ordering the lists in the data tree, we may reduce the search. We may also use hashing techniques to create efficient searches.

As an example, the Instruction - >< - may state

Country (Name (NORWAY), Person >< Person (Age (Adult

This will insert the tag – Age (Adult – on all Person-s in Country (Name (NORWAY.

8 Extended search

Rather than filling in values into the Contents Schema. Search may be stated as added Conditions:

1. Country (Name (NORWAY), Person (Name, Loved-person <> Loved-person
   /‘ ‘ (Person (Name (MARY

- This search is the same as of 3 in the previous section.

   may be stated as

2. Country (Name (<> : (NORWAY)), Person (Name, Loved-person <> Loved-person

/' ' (<u>Person</u> (Name <> : (MARY

- This search is the same as of 3 in the previous section.

Note that both the added Conditions - <> : (NORWAY) and <> : (MARY) - appear in the Contents Layer only, and they do not appear in the External Terminology Layer. The colons after the Conditions refer to the conditioned attributes.

Note the extra parenthesis in front of the first Condition, that is needed to show that the <u>Person</u> appears at the same level as Name.

We may also combine the styles:

3. <u>Country</u> (Name (NORWAY), <u>Person</u> (Name, <u>Loved-person</u> <> <u>Loved-person</u>
   /' ' (<u>Person</u> (Name <> : (MARY

- This search is the same as of 3 in the previous section.

Above, we have stated the Conditions on the attributes. They may alternatively be stated on the entities:

4. <u>Country</u> (<> (Name (NORWAY))) (Name, <u>Person</u> (Name, <u>Loved-person</u> <> <u>Loved-person</u>
   /' ' (<u>Person</u> (<> (Name (MARY))) (Name

- This search is the same as of 3 in the previous section.

Note that in the last formula we have omitted the colon in : after the Conditions, before the (Name, as the expression is clear without stating it, ie. the (must be contained in something.

The blank in front of (Name, after NORWAY))), indicates that the Name is contained in <u>Country</u>, and is not a part of the Condition. Similarly for the very last (Name.

The above examples show 'conjunctions' of search requirements.

The following expression states a conjunction.

5. <u>Country</u> (Name (NORWAY), <u>Person</u> (Name, Sex (MALE), Category (CHILD)

- This search is the same as of 11 in the previous section.

We may state a disjunction by introducing a schema reference

6. <u>Country</u> (Name (NORWAY), <u>Person</u> (S<> S '<u>Person</u> (Sex (MALE), Category (CHILD)) Name, Sex, Category

- This search lists only <u>Country</u> (Name (NORWAY, and ONLY its <u>Person</u>-s having Sex (MALE OR Category (CHILD

The schema reference refers only to the current Person in the Contents Layer. It does not refer to any other schema in the Contents or Terminology layers. Note that the first S appears at the same level as Name, Sex and Category.

> This schema reference states that <u>Person</u> may contain the options Sex (MALE or Category (CHILD. It does not state that there has to be minimum one of them. Hence, the expression allows for <u>Person</u>-s for whom there is registered no Sex value or no Category value, or the <u>Person</u> may have both Sex (MALE and Category (CHILD. The expression does not allow for <u>Person</u>-s being non-MALE or non-CHILD, ie. not <u>Person</u>-s that are FEMALE and/or GROWN-UP.

We are now ready to add negations, first to the conjunction:

7. <u>Country</u> (Name (NORWAY), <u>Person</u> (Name, Sex (MALE), Category (CHILD !)

- The search will only show <u>Country</u> (Name (NORWAY, and only its <u>Person</u>-s who are both MALE AND not CHILD.

We may add a negation to a disjunction, as well:

8. <u>Country</u> (Name (NORWAY), <u>Person</u> (S<>S ‘<u>Person</u> (Sex (MALE), Category (CHILD !)) Name, Sex, Category

- The search will only show <u>Country</u> (Name (NORWAY, and only its <u>Person</u>-s who are MALE OR not CHILD.

Empty values of Sex and Category may be prohibited by the definitions in the External Terminology Schema. If not, they may be excluded in the formulation of the search:

9. <u>Country</u> (Name (NORWAY), <u>Person</u> (S<>S ‘<u>Person</u> (Sex (MALE, !), Category (CHILD !, !)) Name, Sex, Category

- The search will only show <u>Country</u> (Name (NORWAY, and only its <u>Person</u>-s whose Sex is MALE OR not empty OR whose Category is not CHILD OR not empty.

The search result will satisfy any of the alternatives listed. The negation symbol -! - after a blank means not empty. This example shows how we may require a disjunction of values of each attribute, and disallow some.

The fine thing with the formulations presented in this section is that they do not require any categorization of data into attributes, values or entities, other than the added Conditions and their locations.

If the Conditions are stated on the entities, they will be executed first, and there is no need for the arrangement of data as being described in a previous section. One difficulty is that the Condition will in the three-dimensional notation be executed before checking the name tag of the entity.

When using the two-dimensional notation, the name tag will appear at the same line as of the colon representing the entity; hence, the name tag will be checked before executing the

Condition. Therefore, the mentioned arrangements of data will be as wanted when using the two-dimensional notation.

9 Search execution

A search specification is found via the schema reference to a Contents Schema.

We have the following rules for a search:

1. When there is no value - added by the end user or software - of an entity, all entities of this class (under the superior entity) are listed in the search result.
2. If a searched identifier attribute value is found, only this entity of this class (under the superior entity) is listed.
3. If all searched attributes values of an entity are found, then only these entities of the class (under the superior entity) are listed. This is a kind of conjunction.
4. If some of the searched values defined via a schema reference from the Contents schema are found, any of these entities of the class are listed. This is a kind of disjunction.

The rules are formulated per entity. Hence the execution must be stated per entity. This may be achieved either by:

a. Tagging each data item as entity, attribute, value etc.

b. Stating all conditions by using the Condition symbol at entity, attribute, value level etc.

This tagging may be done by software, and need not be seen explicitly by the end user at his screen, but implicitly he will see it, eg. by values appearing in variable fields.

When executing a condition branch, for each entity instance,

i. Any execution of a data item starts with checking if it has got a Condition, and first executes its condition branch: there may be more condition (sub-)branches within a condition branch; the conditions may state rules for the bullets above

ii. Next, check if the data item contains a schema (sub-)reference; the schema references may state additional rules for bullet 4 above

iii. If an entity in the controlled branch is found to satisfy the search rules 1-4, then after the control,

iv. If the data item contains a write Instruction, execute this branch to the Tip of the instruction branch, while creating the Tip of the write branch

v. If having executed an instruction branch, fetch all requested data within the controlled branch; this is the second run in chapter 2,

vi. If all required data to a write Instruction are fetched, then map these to the write branch, while possibly using a function mapping

vii. and continue the execution to the contained items of bullet iii

viii. If the entity in the controlled branch does not satisfy the search rules 1-4, then the execution continues to the Next item, bullet i

ix. If there is no Next item, return to the first Previous, then Superior, then the execution continues to the Next item, bullet i

x. If having backtracked to the Condition, and some controlled data satisfy the entire condition branch, then continue to the subordinate of the conditioned item

xi. If having backtracked to the Condition, and not any controlled data satisfy the entire condition branch, then the conditioned item - and its condition branch, subordinates and write Instruction - is deleted.

10 Primitive execution

The Primitive execution explains the fundamental execution of Existence logic. In practical applications, we will apply other forms of execution.

The Primitive execution deals with execution of a Condition. There may be Instructions and more Conditions within the condition branch, but we do not here address the particular operations on each.

First, we assume that there is only one sub-branch of the control branch that satisfies the Condition. This is typical for references within the External Terminology Layer. We assume that entity identifier attributes with values are added to the reference.

Second, we control the data beneath the conditioned item, and not data that first require navigation to superior items.

Third, we execute one Condition only, not nested Conditions and not write Instructions, but we allow for existence of nested Conditions and Instructions.

Fourth, as a consequence of the third assumption, we do not discuss implementation of functions and write Instructions.

Fifth, some items will only partially satisfy the Condition branch; hence, we need to support partial backtracking.

Sixth, we will not support negation.

Seventh, we assume that all navigation is made without use of name tags.

Eigth, the controlled branch may contain Next (and Previous) items that do not appear in the condition branch.

Note that the name tags may be defined by colons in three dimensions. Hence, name tags may be included, and their appearances may be checked, but we do not formulate the execution to be about name tags. Practical implementations will though check the name tags.

When a Condition is found, then a Read head and a Control head are created. One of these may be the already existing Read head before we reach the Condition. The Read head traverses the condition branch.

Whatever is found under the Read head is accompanied with a parallel move of the Control head – in their separate branches.

We may imagine that the condition branch creeps along the controlled branch, the Read head and Control head are synchronized through a common axle and rotate the same way. The heads will have to rotate in three dimensions – subordinate, next and condition -, and may reverse.

As an alternative to the axel, there may be a communication line between the two heads, sending an instruction from the Read head to the Control head. Preferably, the communication is synchronous, creating one step at a time. The Control head returns a confirmation that the move has Succeeded or Failed.

If the move at the Control head Fails on all attempts, then the Conditioned item is Deleted, and the extra head is removed. This means that there is no controlled branch satisfying the condition branch.

If an item is the n-th item in a list, then in order to come back to the Superior item, the head will have to make n-1 moves via Previous items to the first item of the list, and then make a Superior move.

Note that in the Primitive execution, each item has only one list of Subordinates. Hence, if both Person-s and Dog-s are Subordinate to the same item, then these will appear in the same list.

If the heads are automata that remember all their Superior items, the Previous moves are not needed. In this case, somehow, the automaton will need to hold a 'finger' at each Superior item. This may be achieved by creating a separate pair of read-write heads for every Contained list, ie. one pair for each list. When the execution of a list is finished, the control is given to the directly superior read-write pair, which holds the 'fingers' on the right data items. This alternative requires that the read-write heads communicate with their immediate Subordinates and Superior read-write heads.

The threads within the Contents Population – of the Contents Layer - may provide this log of all Superior items. The Contents Population consist of threads that are parsing through the External Terminology Population. The Contents Population acts as a Condition, and maybe each data item in any condition branch can be considered to be a read-write pair. These pairs communicate with each other, without having to parse intermediate data items. However, the head in the controlled branch will have to parse and test intermediate data items that do not fully satisfy the Condition.

Each data item may have three dimensions, and if used, have branches in 2*3 directions:

• Condition	<>
• Subordinate	(
• Next	,
• Previous	;
• Instruction	><
• Superior	‘ or)

These instructions may be written in

- One dimension
- Two dimensions
- Three dimensions

The three-dimensional notation is the most fundamental one. The one-dimensional notation is easiest to execute. The two-dimensional notation, combined with a one-dimensional notation for the details, eg. sub-conditions and sub-instructions, gives the best overview and is easiest to read.

The one-dimensional notation is written and executed from left to right, going into each parenthesis, until coming to its right end.

In the one-dimensional notation, right hand parentheses -) – are indicating the end of a list, thus if finding no Subordinate, the parenthesis is instructing the Read head to go to the Superior item of this list, then proceed to its Next item, its Subordinate etc. We see that when coming up to the Superior item, then the Next item is prioritized, then the Subordinate of the Next etc.

The entire Condition branch is executed towards bottom first.

The Control head is following the Read head all the way. When a Tip is reached, the Read head is backtracking in the reverse order of the above navigation.

During backtracking, we first go to the Previous, then to the Superior item. If we allow for negation, we first have to go to the direct Superior – and down again -, before we do the move to the Previous.

Before moving to the Previous, we first have to test if there is a Next item. If so, we turn from backtracking to forward moves.

We conclude the discussion on backtracking:

- If we come to a Superior item, we make tests in the following sequence: Next, Superior, Previous.
- If we come to a Previous item, we make tests in the following sequence: Previous, Superior.

Detailed backtracking is not needed if the Read head has a memory of all Superior items. The memory can be implemented by having a separate read-write head for every level of the condition branch, or for every item of the condition branch. The backtracking then takes place to the Superior read-write head. This arrangement requires a communication to directly Superior and Subordinate read-write heads, or even to the Next and Previous head. We may have a read-write head for each data item in the condition branch.

When backtracking over the Condition, the Condition is validated. If the Condition in the population is not satisfied, then the conditioned item is deleted.

The rest of this section is an Extension of the above:

When the Read head finds a Tip, ie. an item having no Subordinate and no Next item, then in a second run, the Control head checks if it has any Subordinate item that does not appear under the Read head; if so, the Control head functions as another Read head that copies all the Subordinate contents. This will become an argument of an Instruction. See later. The Control head may find several arguments, and functions as one Read head for each argument.

When the backtracking comes to an item containing an Instruction, the Instruction is executed. A Write head is created. The Write head is following the Read head (in the instruction branch) in the same way as the Control head. However, if the Read head finds an item that does not appear under the Write head, the Write head inserts this item. This way, the Write head is copying from the Read head in the Instruction.

Above, we have described a function application, ie. a mapping from constant arguments to a constant function value.

It is possible to state the Instruction on the original Conditioned item, but this is inconvenient, as the Write head will typically use the navigation of the Control head, and start where the Control head is positioned when the read head finds the Instruction. Therefore, we place the Instructions inside the condition branch.

When the Read head comes to a Tip of the Instruction, it will look up all the Read heads containing the arguments in the control branch, and copy all the arguments under the position of the Write head. The copying takes place in the sequence that the Tip-s were found by the Read head. The Read and the Write heads use synchronous communication. In addition, there needs to be a link between the Tip-s containing the copy items. The Write head creates a concatenation of the copies.

The last paragraph describes concatenation of variable, ie. unknown, structures into a common result. We have not here explored functions from variables to variables. See the function section.

A variant of the above, is to copy the entire branch under the Control head from where the Instruction is found under the Read head. This requires that the Instructions are placed on the right places. We will go for this solution.

A variant of the execution may copy from the Control head to the Read head, from the Read head to the Instruction head, and from the Instruction head to Write head. However, this way, we will modify the condition and instruction branches such that they will not be reusable. Therefore, we will not explore this option.

11 Navigation above the Condition

Most condition branches will contain navigation above the Condition. This is not covered in the previous section.

The navigation from the Condition will first go via Superiors to a Top. We call this navigation first-forward.

The first-forward will test Previous, and then Superior. This way, we avoid negations.

This is the same approach as for backtracking from Previous.

In the condition branch, there may be no Previous, so we may go directly to Superior. In the controlled branch, there may be many Previous. However, we are ascending through the path which led down to the Condition. Hence, they may have a recollection of the Superiors.

If we have one separate read-write head for each list of Next items, the control head will have Superior read-write heads from before we reached down to the Condition. Hence, they will have a recollection of the Superiors.

The Top within the condition branch has no Previous and no Superior. After having reached the Top, we may turn to forward processing.

There is a special case at the second level beneath the Top in the condition branch:

- The first Subordinate entity is the one that we passed when ascending
- The Next of the first Subordinate entity is the one that we want to descend through in forward execution

This difficulty about ascending and descending only appears in the three-dimensional notation; it disappears in the two- and one-dimensional notations.

Note that the name tag and the attributes of the Top – under the control head - may appear before the first Subordinate entity. What is mentioned here implies that the Top and its immediately Subordinates need to be given special treatment in the three-dimensional notation..

When doing backtracking, this may proceed to the Top. Then we have to turn to last-backward. Now we descend to the Condition. The descending takes place via the first Subordinate entity. This is the first Subordinate item under the read head. A search via Next items is needed under the control head. Alternatively, a control head is left from the ascending, and is now used as a finger on the right entity. We may even skip the ascending, and go right to the Conditioned data item.

The condition branch may be split into several sub-branches. The control head needs to find minimum one branch where all the sub-branches of the condition branch are satisfied, for the entire Condition to be satisfied.

12 Addition on navigation

If the condition branch includes one or more Instructions, then the instruction branch together with its write branch may create a partial search result, without the entire Condition being satisfied.

The sub-branches of a Condition may be split at the Condition, at the root of the condition branch. If so, the backtracking from a sub-branch shall proceed to forward processing of the next sub-branch until all sub-branches are processed before concluding the Condition.

One or more sub-branches may navigate down the data tree. One sub-branch may navigate up the data tree. For example, the various sub-branches may be used to access the arguments of a function application.

13 Various notations

In the three-dimensional notation, we have only one letter - : -. These colons are organized in a three-dimensional tree. The dimensions are

- Subordinate,
- Next and
- Condition.

In the two-dimensional notation, indentations are used to show Subordination. Line shifts are used to show Next items. Conditions are indicated by indented <> symbols. This means that opening (and closing) parentheses, as of the one-dimensional notation, are not needed.

In the one-dimensional notation, parentheses are used to indicate Subordination. Commas are used to indicate Next items. Conditions are indicated by the <> symbol. The end of a condition may be indicated by a blank before a Next item. Hence, the blank indicates Subordination to the item in front of the Condition. The Condition acts as an opening parenthesis. The reading is clearer if the blank is not used, an opening parenthesis is placed before the Condition, and a closing parenthesis is placed after the condition branch.

Two- and one-dimensional notations are often combined, as statement of the condition branches would require many lines and indentations in the two-dimensional notation. Therefore, the condition and instruction branches are most often expressed in the one-dimensional notation. In this combined two- and one-dimensional notation, closing parentheses are most often not needed. This combined notation is the most practical and useful notation.

If the one-dimensional notation exceeds a line, it continues after a backslash in the next line.

14 Execution of one-dimensional expressions

The one-dimensional notation is written from left to right. The left hand side contains the root of the data tree. Schemata and populations make up sub-branches of the tree. The sub-branches are enclosed in parentheses. The sub-branches and their data items are arranged in sequences of Next items. Next items are separated by commas. We prefer that schemata are written before populations. Schemata are typically static, while populations may change during execution.

In the one-dimensional notation, we put a blank behind each comma, and behind each apostrophe. We may put a blank behind Conditions and Instructions, and recursion symbols as well. We do not put blanks behind parentheses and negation symbols. We put a blank behind a condition or instruction branch before a Next item under a conditioned item. This blank serves as a right hand parenthesis. A sequence of Next items is terminated by a closing parenthesis.

In the one-dimensional notation, we use full name tags only, and we normally do not use colons. There are exceptions to this, when a reference shall apply between different schemata or between populations and schemata. Then we use colons in order to avoid specific schema names within the reference to be instantiated into a different population name.

The one-dimensional notation is executed from left to right. The data items are arranged such that this means execution towards the bottom first.

When backtracking to Superior-s, the options are tested in the sequence Next, then Previous.

If a Next item is found, then the processing is turned to forward execution.

If a searched data item is found in the control branch, followed by a negation symbol - ! - on the corresponding item in the condition branch, then the Condition has Failed. If a negation is found in the condition branch, then the corresponding data item in the control branch shall not be copied to the write branch. A negation in the control branch shall be copied to the write branch.

Note that we do not support double negation or any kind of truth-value calculations.

A data type is

ColourType (Red, Blue, Green

This data type is referenced in a schema from the attribute Colour

Colour (S <> '& '(ColourType

The same Colour in the schema has a value : with a Condition Red !, ie.

Colour (S <> '& '(ColourType (: <> Red !

This is equivalent to

Colour (S <> '& '(ColourType)), (: <> Red !

Here we state that the Colour contains a reference (S) to a schema named ColourType, and that the Coulor contains an element (:) having a Condition Red !.

We try to insert Red into the population:

Colour (Red

This Red value will be deleted, due to the Condition Red ! being copied into the population and being applied on the Colour (Red.. Red is permissible according to the ColourType, but not according to the Colour. Note that Red will be inserted at the user interface, but is deleted in the External Terminology Layer.

We may insert the value Blue

Colour (Blue

which will be accepted.

From the example, we see that the value within the Condition together with the negation tag must be treated as a unit. Blue is not literally identical to Red !. Hence, Blue should have been deleted. Blue is not deleted, due to the special treatment of negations. This has created doubt in me on the treatment of negations, but I have no other solution.

15 Contents execution

In the Contents Population, we want to formulate a look up of a Person JOHN and to list his Loved-person MARY and her Age.

The Contents Population is stated as follows:

> Person (Name (JOHN), Loved-person <> ‘ ‘ (Person (Name (MARY), Age (27))))

This is a compound statement navigating from the Person JOHN to the Person MARY and her properties. The Condition - <> - is not to be executed as a test – as when written within the External Terminology Population. The Condition just states the navigation through the External population. Hence, the Condition will not result in a Deletion of Loved-person if the Person is not found.

If the end user operation is a write Instruction, rather than a read, and the Condition is expressed in the External Terminology Layer, then the Loved-person will be deleted if the referenced Person is not found.

The entire Contents Population may be understood as a Condition on the External Terminology Population. We assume that this Condition is populated with all data to be found under it, including subordinate Conditions, write Instructions and schema reference. This way, the Condition serves as an import function.

The Contents Population is presented on the end user screen, where the user adds or modifies data, which through a write Instruction are exported to the External Terminology Population.

We see that Conditions serve as import functions – if the Condition is satisfied. The write Instruction is stated within the Condition. All data gathered under the write Instruction are exported to the place indicated by the write Instruction - which is not shown here.

If some of the Conditions and Instructions under the write Instruction fails, then these sub-Conditions and sub-Instructions are deleted, and are not carried out.

Note that it is the developer who writes data into the Contents Schema. This Contents Schema may automatically be supplemented by additional Conditions and Instructions from the External Terminology Schema. In most practical application, we may leave the Contents

Schema and Contents Population dumb, and do all constraint enforcement and derivations in the External Terminology layer, or even in the Internal layers.

When finding a referenced item in a search operation, we have to navigate up and down in the data tree. After having found the item, I have been wondering if we could establish a direct link between the reference and the referenced item. Through use of the Internal layers, this is exactly what we do. Copying all required data to the Condition is a means to avoid the direct link.

16 Primitive automaton

The automaton has an axle between two synchronized read-write heads. Data under one head may be used to control data under the other head, and data from the other head may be copied to the first head, or vice versa.

Rather than having the read-write pair separated from the data, it will be of interest to associate the head functionality with the data - : - themselves. Each item in the condition branch may correspond to a read-write pair. Hence, each item needs to be capable of communicating with its directly superior and subordinate item, and its next and previous item.

The data item will have

- 3*2 directions, ie. superior-subordinate, next-previous, condition-instruction

The coupling between the data items will be

- bi-polar, ie. superior-to-subordinate, next-to-previous, condition-to-instruction

The data items communicate via their bi-polar connections, and they orientate themselves according to these directions. I am wondering if the data items can rotate during execution, and rotate back to the bi-polar direction, being the stable position.

The rotation may create signals to the neighbour data items. Rotation between 3 directions may give 6 signals. Rotation between 6 directions may give 2*(5+4+3+2+1)=30 signals. We may as well have two or more moves in a signal sequence, eg. for 3 directions, we for each direction may move in one out of two directions and back again. This topic requires further analysis.

In the initial analyses, we assume that each data item holds

- one of four states, ie. forward, backward, first-forward, last-backward

I am uncertain if it is possible to reduce the number of states. I have found it convenient to introduce these states, but maybe some can be removed, when we know the direction of the incoming connection signal.

In addition, we need

- signals on Satisfied and Failed

These are transferred through the backward and last-backward states. The Satisfied signal needs to be transferred to the Instruction, for it to know if the Instruction is to be executed. The Failed signal needs to go back to the Condition.

The read function requires a read functionality. The control function additionally requires a comparison function with what is found under the read head. The comparison function finds sub-branches that produce the signal Satisfied from the last backward data item of that sub-branch.

The copying function additionally requires

- a write head

The write head is initiated by a write Instruction. If the read head in the Instruction branch does not find a corresponding data item within the write branch, it will be created. Hence, a write signal has to be transferred between the read-write heads. The initial parsing will though just be a control.

When the Tip of the instruction branch is reached, a read-write pair between the control branch and the write branch is initiated, and data are copied between these branches.

We have mentioned signaling between read-write pairs in the condition branch and control branch. We will create and delete pairs when coming

- To sub-conditions, but may use the same
- Back to instructions, between the instruction and write branch
- To the Tip of the instruction branch,
    - First, re-parsing the Satisfied condition sub-branch and control branches
    - Second, copying from the Tip-s of the control branches to the Tip-s of the write branches
- To rewinding the branches to the Condition

In some cases, only one head has a grip on data, while the other head is free. In fact, the execution takes place on every item in a list of Next items. However, if the condition on subordinate items fails, then the processing continues until the condition is Satisfied. Thereafter, the processing continues recursively, and finds all items in the list where the condition is Satisfied. This is a search mechanism, and it additionally allows for modifications, ie. to execute a write Instruction, everywhere where the condition is Satisfied.

Seen from the outside, it looks as if the control wheel finds only those items where the condition is satisfied. Therefore, it looks as if the control wheel is free between these items, but in reality it is not. Since the full execution only takes place where the condition is satisfied, it may be possible to use direct access to these items without going sequentially through the whole list of items which will not satisfy the Condition. For the direct access approach, we use the Distribution and Physical layers of a database application. However, in this section we discuss primitive execution through lists.

Several axles may be linked, such that data from one axle may be copied to another axle. The read head under the Condition controls what to accept by the control head; the control head takes over and instructs the write head. When finished, it returns the control to the read head.

We have an issue when a Condition has one sub-branch going up, and another sub-branch going down. When having executed the down branch, the execution has to proceed through the up branch.

The difficulty in this case is that the branch is split at its very root. At this root, normally we conclude and discard the read-control axle. Seen from the condition branch, the Condition looks identical to any write Instruction.

At the right hand side of a real write Instruction there most often will be navigation to Superiors during the Backtracking to the Condition. In this case, the write Instruction is executed before Backtracking to Superiors.

In the Condition case, the Backtracking shall switch to Forward execution of the branch via Superiors before concluding the Condition.

When starting the execution of the Condition, we do forward execution to Superiors and then turning the forward execution to Subordinates and Next-s in this branch.

When the Condition is split into one down branch and one up branch, the down branch is executed first. When coming back to the Condition, the execution switches from backward execution to forward execution. When coming back from the up branch, coming down from the Superior to the Condition, we finalize the Condition.

One way to distinguish Conditions and Instructions is to create a new read-write axle for write Instructions, but not to do so for Conditions. Before the main Condition, we have only one read-write head, while the other head on this axle is free. When the read head traverses into the Condition, the other head navigates to and through the control branch.

The effect of the previous paragraph is that for a write Instruction we have two axles; for a Condition, we have one axle only.

We have another issue when the External terminology schema just states a Condition along one path to a referenced entity, while the Contents schema states sub-branches to the identifier attributes of the various entities involved in the navigation path.

In principle, all values in the sub-branches have to be satisfied. The condition branch is executed from its root. Therefore, the sub-branches closest to the root have priority. If these are satisfied, while others are not, the search returns the short branches. Values further out may not be tested. This means that if we search for <u>Person</u>-s and their <u>Loved-person</u>-s, we test the <u>Person</u>-s first, then the <u>Loved-person</u>-s.

The impact of the two above paragraphs is that a search result is returned, while the entire Condition is not satisfied. Hence, the Conditioned data item is deleted. Any write Instruction has to be executed before the deletion takes place.

Note that a write Instruction does not require that the entire Condition is satisfied. It requires that the condition in its sub-branch is satisfied. These sub-branches are satisfied when a complete execution including backtracking to the write Instruction is completed.

Many branches or sub-branches may satisfy a sub-condition, without the total sub-branch satisfying the condition (under the write Instruction).

17 Operation of the read head

This section addresses the operation of the read heads, and do not address operation of control or write heads. See subsequent sections.

We assume that there is one read head per data item, and control signals are passed between them, depending on their state, neighbouring data items and the current phase of execution.

The execution is performed through four phases that are executed in the following sequence:

- Ascending forwards, ie. First-forward
- Descending forwards, ie. Forward
- Ascending backwards, ie. Backward
- Descending backwards, ie. Last-backward

The Read head has the following orientations/states:

1. Condition
2. Instruction

3. Down
4. Next
5. Up
6. Negation
7. Previous

The Negation state is not a pure state of a primitive automaton. It is a compound state that may be defined by the other states. However, it is so important that we have included it here.

The reasoning in the subsequent text is as if we have a three-dimensional representation. Here Negation is represented as a dangling Superior of a non-first data item in a list. In two- or one-dimensional notations, the Negation is marked by a special symbol - ! -.

If we remember all Superior data items, there is no need of backtracking via the Previous ones. Hence, it is a trade off if we define Negation and Previous to be primitive states.

Ascending forwards:

A Condition has been found during Descending forwards; the control of the control branch starts by an action under the Read head at the conditioning side, ie. on the right hand side of the Condition. We are here only addressing the Read head.

- An action tests if the data item has a Superior data item
    - If No, this is a Top, and the state is moved to Descending forwards, Condition state
    - If Yes, the control is moved to the Superior data item

Note that when Ascending forwards, name tags are not checked, even if they may be stated for readability reasons, eg. ‘Person ‘Country rather than just ‘ ‘, as there is only one Superior.

Descending forwards:

Execution of any data item starts in the Condition state.

- An action test if the data item has a Condition.
- If Yes, a (new) Control head is created.

If No, and if the data item is not a conditioning item, the data item is moved to the Instruction state. This state and action may be left out until Ascending-backwards.

- An action test if the data item has an Instruction.

- If Yes, a Write head is created. See Notes below.

If No, the data item is moved to the Down state.

- An action test if the data item has a Subordinate.
- If Yes, the control is moved to the Subordinate data item.

If No, the data item is moved to the Next state.

- An action test if the data item has a Next.
- If Yes, the control is moved to the Next data item.

This terminates Descending forwards, and the execution goes to Ascending backwards.

Note that the creation of the write head may wait until the Instruction is found in Ascending backwards. In a subsequent section on Pseudocode, we will do so.

Note that if the conditioning data item itself is the Top, then the Instruction state is skipped, and a write head is not created. See the second paragraph above.

Note that when Descending forwards, name tags are checked at every level. The name tags for Descending forwards will most often be different from the name tags for Descending backwards, which come first in the lists. In some cases they will be identical, but at some level they will divert, except when referring to a directly Superior, which allows for recursion. See text on the write head.

In order to distinguish the data items of Descending forwards and Descending backwards, we may tag the items already covered, or tag the items for Ascending forwards. In the two- and one-dimensional notations this is done by the single quotation mark - ‘ -. When finding the first item in a list of Previous items, we refer to the first when Descending forwards, ie. we do not include the ascending marks - ‘ -.

Items from Ascending forwards and Descending forwards may appear in the same list only at the first level beneath the Top.

Ascending backwards:

If No (Next data item), the data item is moved to the Superior state.

- An action tests if there is a (direct) Superior data item
  - If No, this is a Top
    - This terminates the Ascending backwards, and the execution goes to Descending backwards

- If Yes, the control is moved to the Superior data item, in its Negation state in Descending forwards.
    - An action test if the data item has a Previous.
        - If Yes,
            - An action tests if the data item has a direct Superior, then it is a Negation.

              If Yes, then perform Negation, and then Continue
        - If No, Continue
    - Continue-ation: An action tests if the data item has a Next.
        - If Yes, the control is moved to its Next data item, in its Condition state Descending forwards.
        - If No, proceed Ascending backwards

This terminates the Ascending backwards at the Top, and the execution goes to Descending backwards.

Note that the test on Negation assumes that the negation is indicated by a direct Superior tag. The test will be different if the negated data item is tagged by the negation symbol - ! -.

The test on Previous is therefore artificial, and come out of order. Since we remember all Superior-s, there is no need to backtrack via the Previous one.

Note that test of Instruction has to be added to the above logic. Also, Ascending backwards may arrive at a conditioning item of a Condition, and without this having a superior.

Descending backwards:

The Descent starts at a Top

- An action test if the data item has an Instruction, ie. it is on the conditioning side of a Condition
    - If Yes, then the Condition is terminated
        - If the Condition is Satisfied, then the control is moved to the conditioned data item, Descending forwards, Instruction state
        - If the Condition Failed, then the conditioned data item is deleted, and the control is moved to its Next data item, Descending forwards, Condition state
    - If No, then the control is moved to its (first) Subordinate data item

Note that the Top and the conditioning data item may be one and the same. In this case, there is no Descending backwards. The Descending forwards will control the Subordinates of the conditioned data item only. See a separate section on Control heads.

In the above text, we have started with the Ascending forwards phase before Descending forwards, which is the normal case. The logic additionally caters for Descending forwards only. However, if the condition branch is split at its root in one ascending and one descending subbranch, then only one of them will be processed. We'll have to extend the logic to cover both in one run.

In the above text, we have discussed execution of the three-dimensional notation. In the two- and one-dimensional notations there is no Top. If we have two sub-branches from the conditioning data item, they are both executed in the sequence they are written in. We may in principle even have more than two sub-branches.

18 Operation of the control head

The control heads are controlled by the read heads, except

a. The control heads have during (implicit) search to parse structures which do not fully Satisfy the structure under the read heads

b. There may be more than one structure under the control heads that Satisfy the structure under the read heads

c. The control heads may find further details that do not appear under the read heads and are copying these to write heads; this may require an extra parsing

The control heads initiate its traverse from the conditioned data item, while the read heads initiate their path from the conditioning data item. Except for the deviations mentioned above, the found paths are isomorphic for the simple Condition, <>. The found paths are homomorphic for the schema references, S<>S.

The control heads may have to parse many data items before they find structures that Satisfy the morphisms. We describe the execution as a sequential search through the structures. Practical implementations may use more efficient approaches, such as hashing.

When finding Condition-s and Instruction-s under a read heads, there may not be any corresponding operations under the control head. If there are corresponding operations, they are parsed as in a normal read of Next and Subordinate data items.

We have so far not discussed processing of recursion symbols - & -. However, the Next operation has an implicit recursion. In the following, we will disregards expressions like '& (&.

In this section, we only present execution of a simple Condition, which searches for one totally identical structure under the control heads.

Example expression:

Country (Name (NORWAY), Person (Name (JOHN), Loved-person <> Loved-person ' ' (Person (Name (MARY

The read heads traverse down to JOHN and his Loved-person. Control heads are created at the Condition. A better solution is that new read heads are created, that executes the right hand side of the Condition. The old reads heads take the role as control heads.

The control heads traverse back through their superior data items on the left hand side of the Condition. This traversal is synchronized by the ascending read heads on the right hand side up to Country. From here, the read heads descend down to MARY while forcing the control heads to do the same.

In the read branch - at the right hand side - there is only one subordinate Person, who has one subordinate Name, who has one subordinate MARY. The control side may contain much more data.

The read heads descend from Country to Person. The corresponding control heads descend from Country, and find its first subordinate item. Then they proceed to the Next data items with all their sub-branches until they find Person (Name (MARY. There will be many Person-s, but only one Name in each, and maybe only one MARY in the whole structure. This gives opportunities for optimization of the search. We observe that the read heads are forcing the control heads to try to find and execute

Person (Name (MARY

for each data item sub-ordinate to Country. When Fail-ing at any attempt at any level of Person, Name, MARY, the search goes to the Next item at that level. At each item, there is a need for a Status with the values Failed, Satisfied or blank (for not executed).

If Fail-ing at any level, eg. at the Name value, then the Descent forwards terminates, and the Superiors are marked as Failed up to and including a level when a Next data item is found, eg. a Next Person. When a Satisfied is found, all Superior data items are marked as Satisfied. First the heads go up to and including an item having a Next item. Thereafter, they go up to the Condition or Instruction.

The Status need to be marked in the control branch, such that when the Satisfied Status has propagated back to the conditioned item of the Condition, then the Condition is Satisfied. If not, it has Failed.

Strictly speaking, we need only to mark a data item if the processing of the Subordinate items have resulted in Satisfied. Hence, we may move the one Satisfied mark up to the conditioned data item. When coming up to an Instruction in the read branch, we have to reprocess the Satisfied items in the control branch.

Note that the Satisfied mark may only be inserted when Processing backwards. From the setting of the mark up to the second processing - of the Instruction -, the Satisfied data items need to be locked against updates, and the Satisfied mark may be removed during this second run. There is need of no other status mark to tell what data have been processed so far.

An alternative to use of the Status mark would be to copy the result of a search interactively to a result branch. However, this result branch would need to be maintained with lots of deletions. Therefore, we will not use this option. Also, we may get challenges with referencing the right items in the control branch.

We have previously mentioned the option to organize the execution per entity. This is similar to introducing a separate Condition per entity, which contains the search requirements. The above example will then read as follows:

> <u>Country (</u>Name (NORWAY), <u>Person</u> (Name (JOHN), <u>Loved-person</u> <> <u>Loved-person</u> ‘ ‘
> <u>/ (Person <> Person</u> (Name (MARY Name, Age

The expression <> <u>Person</u> (Name (MARY finds MARY, and thereafter Name (MARY and Age (18 are fetched. This illustrates bullet (b) in the introduction to this section. The control heads will have to parse detailed structures, eg. (18 of Age, that are not prescribed by the read head. The control heads will have to do this processing as if they were read heads.

Note that when having nested Conditions, ie. Conditions within condition branches, they all operate on the same control branch.

The fetched data, eg. Name (MARY and Age (18, may be used in a write operation, which may include execution of a function. When doing so, the control heads may have to parse conditions and schema references, like any read head.

If the read branch contains an Instruction - >< -, then all the Satisfied data beneath the Instruction in the controlled branches need to be fetched. This is best achieved if all Satisfied data items are marked during a first run. A second run is needed to retrieve these data.

In this paper, we do not deal with clean ups after the processing is done.

19 Operation of the instruction and write heads

When the processing under the read heads have come back to the write Instruction, instruction heads and write heads are created.

The instruction heads are processing its branch like the read heads.

The write heads are following the instruction heads the same way as of the control heads, except if an instruction head come to a data item that does not appear under the write head, the data item is copied into the write branch by the write head.

When coming to the Tip of the of a write Instruction, the write heads copy all Satisfied data from the control branches - beneath the corresponding Tip in the read branch - to the Tip of the write branch. This copying is described as a second run of the control heads. This processing may involve execution of function mappings.

20 State implementations

I have been wondering if the states can be implemented by rotation of the three-dimensional particles that make up the tokens. This may destroy the directional bindings between the tokens, if there is no elasticity that maintains the couplings, and bring the tokens back to their default position.

An alternative implementation of states would be to have and aisle with an extra read-write head attached to every token. This head is itself a token, that may be attached to tokens in other branches, or spin more freely. The state is measured by the differences between the rotation of the two connected read-write heads. The aisle expresses entanglement between the tokens.

A more classical implementation of states - and operations - would be to state them in the previous item of the first item of the list subordinate to the token. So far, we have not used the previous of the first option. Use of this option raises the question on if the data tree should be twisted clock-wise or counter-clock-wise, or if you may have combinations of both.

21 Pseudo code

The entire code becomes surprisingly complex. Here, we will outline execution of a condition branch. This will be extended to instruction branches and schema references in the subsequent section.

Ascending forwards:

A Condition has been found during Descending forwards; the control starts at the conditioning side, ie. on the right hand side of the Condition

- Start Ascending forwards in the Superior state
    - Tag the item under the read head with a Conditioning state

  - Tag the item under the control head with a Satisfied tag
- Test if there is a Superior data item
  - If No, this is a Top
    - Go to Descending forwards, Condition state
  - If Yes
    - Test if the control head finds a corresponding Superior data item
      - If Yes, move both heads to their Superior data item, and go to Superior state to Test if there is a Superior data item
      - If No, tag the controlled item with Failed, and go to Descending backwards

The purpose of the Conditioning state tag is to tell that this is not an Instruction when returning to this item.

The purpose of the alternative value Satisfied, is to tag the path down again during Descending backwards, in case the data items under the read head do not contain full names.

Descending forwards:

- Start Descending forwards in the Condition state
- Condition state: Test if the data item has a Condition.
  - If Yes,
    - A control head is created and positioned at the current conditioned item; This does not apply for sub-conditions
    - The read head is moved to the conditioning item
    - Tag this item with a Conditioning state; note that there may be more Conditions within a condition branch
    - Tag the item with the Satisfied state
    - Go to Condition state
  - If No, and if the data item is not a conditioning item, the data item is moved to the Subordinate state.
- Subordinate state: Test if the data item under the read head has a Subordinate.
  - If Yes, test if the data item under the control head has a Subordinate

        - If Yes,
            - Move the read head to its Subordinate data item
            - Move the control head to its Subordinate item
            - Tag the item under the control head with Satisfied
            - Go to Condition state of the found Subordinate item
        - If No,
            - Tag the item under the control head with Failed
            - Move both heads to their Next state
    - If No,
        - Move the data item under both heads to the Next state; note that we do not test if there are more Subordinates under the control head
- Next state: Test if the data item under the read head has a Next.
    - If Yes, test if the data item under the control head has a Next
        - If Yes,
            - Move the read head to its Next data item
            - Move the control head to its Next data item
            - Tag the item under the control head with Satisfied
            - Go to Condition state of the found Next item
        - If No,
            - Tag the item under the control head with Failed
            - Move both heads to Ascending backwards
    - If No,
        - Terminates Descending forwards, and the execution goes to Ascending backwards.

Note that if the conditioning data item is the Top, then the Instruction state is skipped, and a write head is not created. See the second paragraph above.

Note that when Descending forwards, name tags are checked at every level. The name tags for Descending forwards will most often be different from the name tags for Descending backwards, which come first in the lists. In some cases they will be identical, but at some level they will divert, except when referring to a directly Superior, which allows for recursion. See text on the write head.

In order to distinguish the data items of Descending forwards and Descending backwards, we have tagged the items already covered with Satisfied or Failed.

In the two- and one-dimensional notations we discerns the routes up and down by the single quotation mark - ‘ -. When finding the first item in a list of Previous items, we refer to the first when Descending forwards, ie. we do not include the ascending marks - ‘ -.

Items of Ascending forwards and Descending forwards may appear in the same list only at the first level beneath the Top.

21 Extensions to Pseudo code

Ascending backwards:

You come to Ascending backwards from

- Descending forwards, when not finding any Next data item, or you Fail to find an item under the control head

When having Ascended to an Instruction,

- an instruction head is created at the instructing side
- The instructing item is tagged with an Instructing tag; this tag is not strictly needed, but is another value of the attribute containing the Conditioning tag
- A write head is created at the current position of the control head; the control head remains fixed at the position corresponding to the instruction side of the Instruction
- The instruction head acts as a read head and the write head acts as a control head, as long as they comply; hence, they do Ascending forwards, Descending forwards, Ascending backwards and Descending backwards, just like the read and control heads
- The write head will do search operations; hence, it will use two runs, using Satisfied and Failed tags, like the control head, however, Failed tags are not transferred to Superior entities; see the next list of bullets
- If a Satisfied item is found under the write head, but there is no Subordinate item corresponding to a Subordinate item under the instruction head, then this item is copied from the instruction branch; hence, the write branches are supplemented with data from the instruction branch

- Now the instruction head returns to the Instruction, while the write head remains fixed
- Note that the instruction branch may contain additional Conditions and Instructions
- The read head starts the second execution of the condition branch under the instruction item, controlling the control head through its Satisfied paths
- When coming to the Tip-s of the condition branch, there may be more corresponding items under the control branch; these are copied and inserted into the write branch
- Ascending backwards within an instruction branch proceeds up to the Top of the condition branch, where Descending backwards take over, or directly to the conditioning item, if there is no Top

When ascending backwards in a condition branch and its corresponding control branches, the Failed tags in the control branch are copied to the Superior item according to the following rule:

- If all Subordinate items are Satisfied, then their Superior item also is Satisfied; ie. we do the conjunction of the Satisfied Subordinates, meaning that if one Fail, they are all Failed
- Failed tags are 'summarized' for values, attributes and attribute groups up to the entity level
- If all Subordinate entities Fail, then their Superior entity Fails, as well; this way, the Failed tag is transmitted back to the conditioning item
- If a Superior entity Fails, then all its Subordinate entities Fail

The two first bullets tell that if the search Fail on one attribute value, then the other attributes need not be processed, and we may ascend directly to the Superior entity, which Fails, and then go to its Next entity.

The categorization of entities may be replaced by an extra Condition on attributes and values of each entity. Conditions on attributes may always be Satisfied, while the required values are very constraining, most often to only one value. The Condition on the entity may state that the identifier attribute must contain one value; this is a means to state functional dependency from the entity to the contained value. The value is contained in an attribute, which is contained in the entity; hence, the value is functionally dependent on the entity, as well.

Attributes without values are used to fetch data in searches, or to fetch arguments to Functions.

That each entity may be interpreted as a Condition, implies that a second run may start at each entity. When doing Descending forwards, this means that for each entity, both runs within an entity may be executed before descending to its Subordinate entities.

During the second run through the condition or instruction branches, only items from Satisfied entities are being used.

During the second run through the condition or instruction branches, the Satisfied and Failed tags may be removed. However, execution of the instruction branch may require a third run through its conditioning branch. The tags may be needed during this run. This is for further study.

Practical implementations of Conditions and Instructions may not use tagging, and may not need two runs.

During the second run, the Satisfied search results are found while the Failed sub-branches are skipped.

During the second run of an Instruction, data are copied from the instruction branch to the write branches

After the second run, a third run through the condition branch may be used to copy data from the control branch to the write branch.

21 Comparison with Turing automata

A Turing automaton has

- A one-dimensional tape;
- A read-write head, that reads a letter and replaces it;
- A finite set of states, of which the machine takes one current state
- An instruction register, ie. a state based programme that instructs replacement within the current tape cell, moves the tape one step back or forth, and jumps to the next state

The instructions may involve no operation, ie. no replacement, no move and no jump.

In Existence logic, we have replaced the one-dimensional tape with recursive tree of lists of lists in three dimensions. We have two read-write heads that are synchronized over an axle. We may have a separate pair of heads for every data item in the condition branch. We have signaling on creation and deletion of neighbour pairs.

Each read-write pair holds a small set of states.

The condition and instruction branches correspond to the instruction register. They analyse the control branch, and copy data from the control branch to the write branch.

The Turing tape may contain further instructions. In existence logic, we instantiate Conditions and Instructions from the schemata among the data items in the populations, and execute. The execution may lead to insertions or deletions. Deletion takes place when the Condition on a data item is not satisfied; the data item and everything beneath it, including the condition branch, are deleted. Existence logic has no overwriting of data.

In Existence logic, there is no instruction for copying and deletion; they both take place implicitly.

The state and instruction registers in the Turing machine seem to apply direct access to any of their items, and they need a read head. There seems to be no notion of writing into these registers, but there needs to be inserts when they are created, and deletes when they are erased. This may be avoided if the machine is a Universal Turing machine that emulates any other machine.