Execution through State Changes

10.03.2020, 10.06.2019 Arve Meisingset

Preface

This paper is a working document, where I explore use of a multi-automaton architecture to execute Existence logic. I realize that enforcement of consistency implies that multiple automata will have to work as if one centralized automaton is handling State Changes. Execution without handling consistency may be done with a much simpler automaton.

Table of Contents

1 States and Status

We assume that each data item is an automaton. Each item is communicating with its nearest items only. The candidate nearest items are:

- Superior and Subordinate
- Next and Previous
- Condition and Instruction

Execution of a Condition results in one item acting as a read head of the condition branch, and another item is acting as a control head that is synchronized with the read head, whenever the Condition is Satisfied.

Execution of an Instruction results in one item acting as an instruction head of the instruction branch, and another item is acting as a write head that is synchronized with the instruction head, copying the instruction items into the write branch at proper places. The instruction branch may contain sub-Conditions for navigation, and may refer via the condition branch to arguments for the write operation.

When arriving at a Tip of an instruction branch, the read head may give the control to the control head, copying its Tip-s into the Tip-s of the write branch. See details below.

Each item holds a State, telling which item it is currently trying to communicate with, and on if the transfer of control with these items is to be moved

- Forwards towards the Tip-s of the current branch or
- Backwards towards its root

For handling information on what has happened remote from an item, each item may hold three statuses:

- Status1, having the value set Conditioned, Conditioning, Instructed, Instructing, empty
- Status2, used for reading operations, having the value set Satisfied, Failed, empty
- Status3, used for writing operations, having the value set Satisfied, Failed, empty

The various statuses typically appear in different branches. Hence, they may be merged into one attribute, if adequate care is taken of the overlaps. However, use of three attributes makes the reasoning clear.

If the data items were not an automaton each, then the statuses may be taken care of in the temporary memory of the computer, and the tagging of each item is not needed. We do not discuss these issues of practical implementations in this paper.

When executing a condition branch, all control branches have to be checked to find any branch that fully satisfies the Condition to its Tip-s. Status2 and Status3 are updated when backtracking through the branches. Therefore, the fully Satisfied paths can only be found in a second run.

Sub-Conditions are just additional Condition-s that each needs to be Satisfied. Instruction branches are different. Their first segment is a kind of condition, marked as an Instruction, that needs to be Satisfied, and is used to navigate to the right insertion point. Note that additional Failed Condition-s may be used for deletion. The second segment of an instruction branch is copied into the insertion points, ie. Tip-s of the write branch getting Status3 (Satisfied. The third segment copies from the Tip-s of the control branch to the Tip-s of the write branch.

The three segments are separated by Insertion symbols (><). An Insertion at the Tip of an insertion branch indicates copying from the control branch. The Insertions may be combined with functions, eg F<>, ><. A functions will contain a schema reference to its definition. It will contain an Insertion and may have a Condition. These may or may not contain further navigation.

Note that the three segments do not correspond to the three statuses.

2 Overview

We will first outline execution of a Condition branch. This branch may contain several sub-Conditions and several Instructions. Only initiation of the Instruction execution will be covered, while the full execution is left for a subsequent section.

Any Condition branch is validated against a Control branch somewhere else in the data tree. The Condition branch may be Satisfied by multiple Control sub-branches. Likewise for sub-Conditions and other sub-branches. Hence, the execution mapping from items in the Condition branch to items in its Control branch is one-to-many. The Condition branch contains navigation instructions for where to find its Control branch. In fact, it is the Control branches which are Satisfied or not.

Any Condition branch may branch off into several sub-branches, not necessarily initiated by sub-Conditions. For the main Condition to be Satisfied for a Control branch, all sub-branches of the Condition have to be satisfied simultaneously. Hence, if a sub-branch of the Control branch satisfies the Condition having more sub-branches, then the corresponding Control sub-branches have to be satisfied, as well. For the Condition to be satisfied, the sub-branches cannot be satisfied independently of each other.

Any Condition branch may or may not contain navigation to Superior-s or Subordinate-s. Navigation to the most Superior is executed first. The other branches are found during Backtracking. I have been wondering if the opposite sequence could be more natural, but the subsequent logic depends on executing the longest paths first.

At each level of Forward execution to Subordinate-s, any Condition is executed first, then the Subordinate is executed. Instructions are found during Backtracking, and thereafter the Next.

If a Condition is Satisfied, execution continues to the Subordinate at the Conditioned item. If the Condition Fails, the Conditioned item may be deleted.
An Instruction is executed if the controlled item corresponding to its Instructed item is Satisfied during Backtracking. If not, the execution continues to its Next.

In the following algorithm we will use Go-to, as we imagine a clock work rotating though the same sequence of states for each new item. This may be inefficient, and formulations using function calls may be more convenient. The algorithm became much more complex than I envisaged, due to the need for maintaining the overall ‘consistency’.

I believe there is a need to develop more primitive automata for simpler condition and instruction branches. Preferably, any entity should be able to serve as a primitive automaton for any other entity.

3 Execution

Each item is executed according to the program below. The execution may ‘Move’ to a neighbouring item, as indicated.

A Move of a read head will impose a Move of the corresponding control head. If the Move of the control head Fail-s, a Status2 (Failed is set on the controlled item or its superior, and the execution is wrapped up in the Backwards sequence.

When a Condition is found, the conditioning item is tagged with a Status1 (Conditioning under the read head. When backtracking, this status will separate this item from an Instruction. The Status2 (Satisfied will tag the controlled items for finding back to the Condition without always having (full) name tags.

We have assumed that each data item acts as one and its own read-write head. Therefore, Status2 (Succeeded or Failed, is copied between the controlled items. The Status tags show sub-paths through the controlled data tree. The paths tagged with Status2 (Succeeded will in a second run be used to fetch data satisfying a Condition.

Note that when having nested Conditions, ie. additional Conditions within a Condition branch, the same read head proceeds into the sub-condition, and the same control head continues from its current position. It is only at the original Condition that the read head and control head depart in different directions from the same position. The control head is controlled by the read head.

The instruction head is a role of the original read head. The write head is a role of the original control head. See a separate section on Execution of the Instruction branch.

In the steps in that subsequent section, the execution may arrive at an item when having carried out a previous step, or when the tested subsequent entity did not satisfy the conditions.

4 Execution of the Condition branch

For each main step, test in the following sequence if it has a reference to one or more of the nine alternatives. If Yes, carry out the sub-steps, which may result in moves to subsequent steps.

Note that sub-Conditions are found and parsed already during Ascending and Descending Forwards. Instructions are found and parsed during Ascending and Descending backwards. The execution of Instructions is not covered in this section.

This is a first draft that will be modified later.

Ascending forwards

1 Forward Condition
   a) Test if the item under the read head has a Condition
   b) If Yes,
      i. Move the read head to the Conditioning item
      ii. Tag the Conditioning item with the Status1 (Conditioning
      iii. Create a control head at the Conditioned item, or in case of a sub-Condition, the control head is already at a controlled item
      iv. Tag the Conditioned item with Status2 (Satisfied, or in case of a sub-Condition, tag the controlled item with Status2 (Satisfied
      v. Execute the Conditioning item, starting at step 1
   c) If No,
      i. Go to step 2

2 Forward Superior
   d) Test if the item under the read head has a first Forward Superior
   e) If Yes,
      i. Test if the control head has a Superior, ie. a Superior item of the first of the recursively Forward Previous
      ii. If Yes,
         1. Move the control head to the Superior item
         2. Copy Status2, eg. (Satisfied of the sub-sequent controlled item, ie. from the passed Subordinate item
         3. Move the read head to the Superior item
         4. Execute the Superior item, starting at step 1
      iii. If No,
         1. Set Status2 (Failed on the controlled item, ie. on the current Subordinate item of what is looked for
         2. The read head remains at the corresponding item
         3. Go to step 3
   f) If No, then this is the Top
      i. Go to step 1

ii. When descending to Forwards Subordinate, do so via Forward Subordinate Next, ie from the second Subordinate, as the first Subordinate is Backwards Subordinate

Descending forwards

3 Forward Subordinate
- a Test if the item under the read head has a Forward Subordinate item
- b If Yes,
  - i Test if the control head has corresponding Forward Subordinate items
  - ii If Yes,
    1. Move the control head to the corresponding Forward Subordinate items; note that there may be more than one in a list
    2. Copy Status2 (Satisfied on each controlled item, from the passed Superior item
    3. Move the read head to the Subordinate item
    4. Execute the Subordinate controlled items, starting at step 1 for each
  - iii. If No
    1. Set Status2 (Failed on the controlled item, ie. on the Superior item of what is looked for
    2. The read head remains at the corresponding item
    3. Go to step 5
- c If No,
  - i Go to 5

4 Forward Next
- g) Test if the item under the read head has a Forward Next, ie. another Subordinate of the Superior in addition to the one found in step 3
- h) If Yes,
  - i. Test if the control head has corresponding Forward Nexts
  - ii. If Yes,
    1. Move the control head to the corresponding Forward Next items, note that there may be more than one in a list
    2. Copy Status2 (Satisfied from the passed Superior item
    3. Move read head to the Next item
    4. Execute the Next items, starting at step 1 of each
  - iii. If No
    1. Set Status2 (Failed on the controlled item, ie. on the Superior of what is looked for
    2. The read head remains at the corresponding Superior item
    3. Go to step 5
- i) If No,
  - i. Go to 5

5 Backward Condition
- j) Test if the item under the read head has a Backward Condition
- k) If Yes,

i. Test if the item under the read head has Status1 (Conditioning
ii. If Yes,
    1. Test if the item under the control head has Status2 (Satisfied
    2. If Yes,
        a) Move the read head to the Conditioned item
        b) The control head remains on its current item
        c) Go to step 3 of the read head
    3. If No, ie. Status2 (Failed
        a) Test if the controlled item has a corresponding Next
        b) If Yes, and this is an insert attempt of the controlled item
            i. Move the control head to its Next item
            ii. Delete the passed controlled item, ie. the Previous item, ie. discard the attempted insertion
            iii. Go to step 3
            iv. If the operation is search, and not insertion, then skip deletion
        c) If No,
            i. Move the control head to its Superior via step 7
            ii. Delete the passed Subordinate controlled item
iii. If No, this is an Instruction
    1. Go to step 6 of the Instruction item

l) If No,
    i. Go to step 7 of the current item

Ascending backwards

6 Backward Instruction, coming from step 5
    1. Coming from Backward Condition without Status1 (Conditioning
    2. Test if the item under the control head has Status2 (Satisfied
    3. If Yes,
        a) Move the read head to the Instructing item
        b) Tag the Instructing item with the Status1 (Instructing
        c) Keep the control head at its current controlled item
        d) The controlled item is already tagged Status2 (Satisfied
        e) Go to step 2 of the Instructing item
    4. If No, the Instruction shall not be executed
        a) Go to step 7 of the Instruction item

2. Backward Previous, coming from step 5 or 6
    a) Test if the control head has <u>any</u> Backward Previous corresponding to the current item under the read head
    b) If Yes,
        i. For each: Tag the Superior item with the Status2 (Succeeded if it has minimum one Succeeded of the Backward Previous items corresponding to the current item under the read head
            1. If No, ie. No Satisfied
                a) Tag the Superior item of the control head with Status (Failed

ii. When having finished all controlled items corresponding to the current read item, test if the read head has a Backward Previous
iii. If Yes, ie. the read head has another Backward Previous
    1. If Yes,
        a) Move the read head to Backward Previous
        b) Go to step 7
    2. If No
        a) Go to step 8
iv. If No,
    1. Go to step 8

c) If No,
    i. Tag the Superior item of the control head with Status (Failed, as it is missing a Subordinate item corresponding to the item under the read head
    ii. Go to step 7.b.ii

7 Backward Superior, coming from step 7 only, ie. from the first Previous

d) Test if the read head has a Backward Superior
    i. If Yes,
        1. Move the read head to the Superior item
        2. Move the control head to its Superior item
        3. Go to step 5
    ii. If No, then this is the Top, and we know from step 5.c that if this were a conditioning item, the execution could not come here; therefore, turn towards the Subordinate
        1. Move the read head to the Backward Subordinate item
        2. Move the control head to its Backward Subordinate item, which is remembered/tagged from Ascending forwards
        3. Copy the Status2, (Failed or Succeeded, from the Superior to the current Subordinate item
        4. Go to step 9

Descending backwards

8 Backward Subordinate, coming from step 8 only

e) Test if the item under the read head has a Backward Condition, ie. has the Status 1 (Conditioning
f) If Yes,
    i. Go to step 3
g) If No,
    i. Test if the item under the read contains a Next item
    ii. If Yes, this is another branching downwards
        a) Test if the Next item is already executed
            i. If Yes, Go to step 9.iii
            ii. If No, Go to step 1 of this Next Subordinate item
    iii. If No,
        1. Move the read head to the Backward Subordinate item

2. Move the control head to its Backward Subordinate item
3. Copy the Status2, (Failed or Succeeded, from the controlled Superior to the current Subordinate item
4. Go to step 9

Note that when Descending backwards, the first item at any level above the Conditioning item is a backtracking item. If the condition branch has subordinate branches to the Conditioning item, then the first item of these subordinate levels are not backtracking items.

Note that 7.b.i reads: “Tag the Superior item with the Status2 (Succeeded if it is not already Failed”. This applies for backtracking through a normal condition branch. If there is no item in the control branch corresponding to the item in the read branch, then the Superior item in the control branch fails. If the search fails in a control branch for one of the items in the read branch, then the Superior item in the control branch fails. For one item in the read branch, there may be many items in the control branch that fail, but one has to succeed if the Condition shall succeed. This way, the ordinary Condition is a strong logical conjunction.

If the condition is a schema reference, see the subsequent section.

5 Execution of Conditions

A control branch of an ordinary Condition - <> - shall be Satisfied at all levels from its root to its Tip-s. If there are several constraints at the same level, eg an Identifier attribute group consists of several attributes, or several different attribute values are given, then all constraints shall be satisfied simultaneously. If not, then this control branch does not Satisfy the Condition.

A reference between entities is stated as a Condition on the Role that refers to the other entity. Before execution, this role must be supplied with attributes and values that refer to the Identifier value of the other entity. Except for this, an ordinary Condition may be executed in one pas.

Schema references - S<> - are different. The Population has a reference to a Schema. The references may be recursive, such that a class within a Schema may contain a schema reference to still another class in a meta-Schema.

When a Contents schema reference is being executed, the Tip of the reference will contain the entire referenced Schema. Identifiers and other constraining values are filled into the population by the end user. The filled in values will be selected and activate the proper subset of the meta-classes, which may include constraints and derivation to still other classes.

The filled in values by the end user constrain search activated by this user. These values typically comprise the Identifier of the root entity, and maybe some attribute values of other entities. The set of given values typically do not identify entities at every level to the Tip-s of the Schema. Some sub-branches may be constrained, others may not.

First, the execution of the search identifies every path that are constrained by the end user input. In a second run, all values of the specified attributes of Satisfied entities are retrieved.

All constraining attribute values of entities must be Satisfied simultaneously for the entity instance to be Satisfied. All other instances of this entity class Fails, and are excluded from the search result.

If there is no constraint on a subordinate entity or referenced entity class, their instances are all included in the search result - in the second run. However, the end user may exclude these by special commands in the user interface.

A satisfied entity instance may contain no subordinate or referenced entity instance. If so, the entity instance is included in the search result even if the schema contains more classes. This makes schema Conditions different from ordinary Conditions, which have to be satisfied to the Tip-s. Search through Contents schema references produce a kind of disjunction.

The search result is put into a Contents population.

The constraints and derivations, eg by references to meta-Schemata in the Application layer are only executed during insertion, modification and deletion. The constraints for search may be validated by the same means before the search.

6 Branching of the Condition branch

The execution is

- first trying to go up and executes the longest sub-branches
- next descends to shorter sub-branches, and
- thereafter, looks for and executes the sub-branches that are going down only, and
- finally, executes the Condition

It may be preferable to turn the execution sequence, going down first, and then the longest branches last. This is for further study.
A condition may contain sub-conditions. They are all operating on the same control branch, but may address different sub-branches. Sub-conditions are not strictly needed, as sub-branches of the main condition may control the same sub-branches.

However, use of sub-condition allows these to be terminated before the whole condition is executed. Also, the main branch may formulate the search request, while the sub-Conditions restrict the result. If a sub-condition Fails, the controlled item will be deleted before the main condition is executed. Hence, the effect of sub-conditions may be different from using sub-branches only. However, deletion as an effect of Failed Conditions is for further study.

6 Execution of the Instruction branch

The Instruction branch is executed in the same way as the Condition branch with some additions. The read head continues from the condition branch into the instruction branch. The control head is controlled by the instruction head from its current position where the read head, when backtracking, found the Instruction.

The controlled branch may now be called a write branch, even if the first part is a control processing to the item where the writing starts.

In the case where the Instruction is placed inside the condition branch, the control branch and write branch will have a common item, which is the root of the write branch. They may or may not overlap in knots further out.

Often the instruction and write branches are very short. They may for example update an attribute within an entity, but may take arguments from other entities by means of their control and condition branches. However, we want also to facilitate updating of large and complex structures.

In a database application, the Instruction is written within an entity that is updated by the end user. Here the condition part is short, but the update may result in editing of an attribute of some other related entity. Hence, the instruction may be long.

The instruction branch instructs a fixed updating of the write branch. The control branch provides variable updating. The updating may be initiated by the instruction branch and be terminated by the control branch, and they may interactively replace each other during execution. They never work simultaneously.

The placement of the Instruction may identify the common Superior item of all its updates, ie. where the control head branches off to fetch arguments and to do the writing.

The fixed copying from the instruction branch to the write branch takes place one item at a time. The insertion takes place Subordinate to the current position of the write head.

The instruction branch may contain sub-Conditions, which may result in deletion of the item under the controlled write head when the sub-Condition fails. This is for further study. Note that Failed sub-Conditions do not result in a deletion of some part of the instruction branch, as the controlled write head is somewhere else.

The instruction branch may contain sub-Instructions, all resulting in insertions by the same write head. The instruction branch may be split into three segments, delimited by separate Instructions:

1. The first segment acts much as a condition branch, commanding the write head to the position where insertions shall start; this is not a real condition branch, as it cannot result in a deletion
2. The second segment commands copying of the contained instruction segment into the controlled write branches branching off from what satisfies the first segment
3. The third segment starts with an empty Instruction, and it copies segments from the Tip-s of the control branches having Satisfied the condition branches

The reasons for having the first segment within an instruction branch, and not within a condition branch, are:

- This control shall only take place when the Condition is Satisfied up to this item in the control branch
- This item in the control branch is the root of the control sub-branches to be used as arguments in the write operations
- This item acts as the common item between the arguments in the control branch and the function values in the write branch
- This item cannot result in deletions, as may become the effect of a Condition

The first segment may be substituted by a sub-branch of the main condition branch. At the Tip of this segment, the second segment, starting with and ending with an Instruction contains the fixed instructions to be copied into the write branch. The Instruction at the Tip of this second segment refers indirectly to the variable items to be found at the Tip-s of the controlled branch. However, now it will not be clear which Tip-s are used for which purpose. Therefore, we will use an Instruction for the first segment, as well.

We use the Instruction symbol in three different roles. In a practical implementation, we may use reserved words to distinguish these roles. These roles are illustrated in the following alpha-graphic figure:

---:----------	Control branch up to variable arguments
--	Contents of variable arguments
<> '..'----------	Main condition branch
><'...'-------	Navigation up and down in the instruction branch
><----	Instructing fixed insertion
><	Instructing variable insertion from contents of the arguments (second line)
-------	Write control branch to fixed insertion
----	Fixed insertion into write control branch
--	Contents of inserted variable arguments

The first segment of the instruction branch must be processed in a first parse, and its controlled items be tagged with Satisfied before the second segment can be processed. Here items from the second segment are copied into the Satisfied Tip-s of the controlled write branches.

In order to distinguish Satisfied items of the evaluation of the Condition from the Satisfied items of execution of the Instructions, we use a separate Status3 attribute for the write branch, with the values Satisfied, Failed and blank/empty. This implies that all items inserted under the write head are tagged with Status3 (Satisfied. Condition branches within the write branch will have to be tagged by both Status2 and Status3.

How do we distinguish the three segments of the instruction branch? The first segment follows after the first Instruction symbol. The third segment has no contents. The second segment follows when coming to an Instruction where the previous items in the write branch are tagged by Status3.

The copying and writing of fixed or variable items takes place one item at a time. For example, if values from an attribute group are copied, one value is fetched, then being updated, thereafter the next value is fetched etc. The values have to be distinguished from the Tip-s in which they are contained. This may be done by not tagging the value contents with Status2 (Satisfied, as they are not referenced from the condition branch, but are implied by the condition branch.

The control branch is tagged with Satisfied and Failed during the backtracking through the first parse. The Satisfied branches are used for updates in a second parse.

During the second parse, the second segment is copied from the instruction branch to the Tip-s of the Satisfied write branches.

If the Tip of the instruction branch ends with still another Instruction, then arguments are copied from the Tip-s of the control branch beneath the Satisfied

items. It will be natural to find the Tip-s of the control branch in a second run through that branch.

When operating on the write branch, instruction branch, condition branch and control branch simultaneously, it will be convenient to use more than two synchronized read-write heads.

As the write branch is completed with whatever is stated in the instruction branch, there is no notion of Failed sub-branches of the write branch, except for sub-Conditions in the first segment of the write branch. Sub-Conditions in the second or third segment are inserted, not tested. However, if Condition are instructed or copied, they may not fit where they are filled in, and may Fail. Conditions that may Fail are permitted.

Whole sub-branches from the control branch are copied to the write branch one-to-one if Functions are not involved. Functions may be associated with the Instructions, eg. F ><.

Functions typically appear as items in the main condition branch, and they both have a Condition (<>) - pointing out the arguments - and an Insertion (><) - pointing out where and what to insert. Additionally, they contain a schema reference (S<>) - pointing out the definition of the Function with mappings from arguments to function values. It will be convenient to write the schema reference first under the Function symbol, then the Condition, and the Insertion last. This gives a convenient execution sequence. The schema tells what arguments are permissible. The actual arguments are pointed out by the Condition. Each value combination is checked against the schema. Their function value is pointed out and used in the Insertion.

Several arguments may map to one function value. The arguments may be pointed out explicitly by F<> or implicitly if the Condition of the function is not stated. If the Tip of the instruction branch contains an Instruction only, ie. ><, then the ‘function’ value is put at the corresponding Tip of the write branch. Note that there may be several Tip-s in the control branch corresponding to one Tip in the instruction branch, and the instruction branch may have several Tip-s. The explicit navigation beneath the Condition and Instruction of a Function must satisfy the navigation in the definition of the Function. Some Function applications may only have implicit references to the arguments, as we may use an Instruction without a Condition.

If the Function uses more than one arguments, they have to be referenced explicitly. The write Instruction, without a Function, copies the arguments to the their destination one-by-one.

7 Overview of Insertion logic

The control branch is already executed down to the items corresponding to the Tip-s of the read branch and the execution has returned up to an Instruction. One and the same item may hold both the Condition and the Instruction. This item may be a function (F), ie it contain a schema reference (F (S<>)). The Condition is executed before the Insertion. The items satisfying the Condition are tagged with Status 2 (Satisfied.

Step 6 is executed for the Instructing item, and the control is given to Step 2 of the same item. The instructing item is tagged with Status 1 (Instructing and the corresponding controlled item is tagged with Status 3 (Satisfied.

The instruction branch is executed in the same way as the condition branch. The instruction branch may contain its own sub-Conditions. In order to distinguish Satisfied items of the main Condition from the Satisfied items of the Instruction, the controlled items according to the Instruction are tagged with both Status2 and Status 3. Of the same reason, the controlled items of sub-Conditions and sub-Instructions may have their own Status n, as well. I am though wondering if for the branches of the sub-Conditions it is satisfactory to do the backtracking without use of extra Status n in the control branches. This is for further study.

The instruction branch may have three segments. The first segment acts similar to a condition branch, and is used to navigate the control to the right item for starting insertions.

The second segments of an instruction branch contains fixed Insertions. The corresponding items of the write branch are tagged with a Status3 both for the first and second segment. The second segment is copied into the write branches.

When all the Tip-s of the second segment are reached, ie or eg when the third Instruction is reached, and all fixed insertions are finished, the execution is backtracking - like in the main condition branch - up to the main Instruction.
For each tagged Tip of the write branch with Status3 (Satisfied, the control branch is executed along the Status2 (Satisfied tags corresponding to the condition branch. The Tip-s of the condition branch are reached. We may call their corresponding items in the control branch for c-Tips. They are not actual Tip-s, as they may contain further details, that the Tip-s of the condition branch do not contain.

When the control head is at the first c-Tip, the write head shall be positioned at the first w-Tip by following the Status3 (Satisfied tags. The positioning is controlled by the read head in the condition branch and the instruction head in the instruction branch.

When the execution of the entire main Condition is completed, all Status1, Staus2 and Status3 tags are cleaned up.

8 Copying

The second last paragraph of the previous section describes copying from the control branch to the write branch. The source is referenced by the condition branch, and the sink is referenced by the instruction branch.

For each c-Tip, the items under the c-Tip are copied to the corresponding w-Tip, ie. from the first c-Tip to the first w-Tip, from the second c-Tip is attached to the first w-Tip etc. Then this process is repeated for each w-Tip. The copying stops when there are no more c-Tip-s to copy from or are no more w-Tip-s to copy to.

Note that when copying, the first level under the c-Tip-s are concatenated under each w-Tip. The sub-ordinate levels are copied untouched.

The insertion starts from the start of the list under the w-Tip and may appear in reversed order compared to the list under the c-Tip. At first, I was unhappy with this, as it would be more convenient to have the insertions at the end, and in correct order. This will require navigation to the end before insertion starts. This is for further study. See at the end of this section. Later, I realized that this reversal of a list is what is needed for managing two-way references. Hence, this functionality is very important. In another paper, we will call this reversal function R<>.

If copying from several c-Tips to the same w-Tip, the w-Tip will appear as a concatenation of strings (maybe in reverse order of the c-Tip-s). We may call this addition of lists.

For implementation of insertions at the end of the lists, it would be convenient to have two-way pointers between the elements - in a circle, like in Codasyl network databases. This may be implemented in the internal layer of a multilevel architecture. This we do not discuss here, as we want to address the primitives. We want to know that the primitives are sufficient, even if they are not practical. If we defer the insertions to the internal layer, then the detailed discussions in this and the next section are not relevant.

9 Deletion

This section gives a new introduction to and extension of the deletion notion.

We have so far provided Deletion as a consequence of a Failed Condition. The expression : <> :! will always Fail, as it states that there is a Condition on the

conditioned item that it shall not exist. However, this interpretation only applies for main Conditions. For sub-Conditions in a condition branch, the conditioned item refers to a controlled item in a control branch. It is the controlled item that will be Deleted.

In the instruction branch we have a Tip A, that we call Tip Ar, and this item refers to an entry A in the write branch. We assume that we have already navigated to A, such that A is a different data item from Ar. We want to delete the A. This we do by stating a sub-condition on Ar that states that Ar cannot exist, Ar<>Ar !. This expression is copied for execution into the write branch, which will contain A<>A !. When this expression is executed, A will be deleted. When the sub-Condition Ar<>Ar ! is executed in the instruction branch, it will only result in a Failed Ar, not Deletion of Ar. This way, we can state impossible conditions in the instruction branch without them destroying themselves.

Note that it is not impossible to write A <> A !, but its execution may result in a deletion of the whole expression. This presupposes that a Failed Condition results in a Deletion, which we at a later stage may abandon.
Note that A <> A >< : is impossible to write in three dimensions, since <> and >< denote the same connection in three dimensions. However <><> and ><>< are possible expressions.

We make the following example and discussion to show the feasibility of the language. The following is not what the user will see.

In the instruction branch we have a Tip Ar that refers to an entry A in the write branch. A contains items A (x1, x2 etc. A contains n items, ie in general, A (:,:,..
In the condition branch we have a Tip Br that refers to an entry B in the control branch. B contains items B (y1, y2 etc. B contains m items.
We want to delete m items in A, corresponding to the m items in B.
Under the first Br (: item, we state a fixed Instruction reference to the first A (: item with a sub-condition that there shall not exist Ar (: <> : !. Hence, (:<> : ! will be added to A (:. When this expression is executed, the item A (: will be deleted. However, we want to have one deletion for each B (: item. Therefore, only the first A (: shall be deleted. Hence, we have to add the sub-condition <> :, ; !. Put together, this will read

<>: ... Br (: (<> : (... Ar (: (<>:, ; !), ><), ; !), ><)).

Note that an Instruction without an instructing item, ie ><, is interpreted as a delete of the current item.

The main Condition is stated at the left hand side of the above expression. The two other Conditions are sub-Conditions. Note that the last exclamation mark is on A (:. Exclamation marks are executed last, ie when backtracking. Therefore, we here have placed it last.

The inner parenthesis states that the item A (: shall have no (!) previous item (;), ie A (: will be tagged with Satisfied for the first item only.

When an A (: item is deleted, we have to delete one B (: item, as well. Hence, we write

<>: '&(& Br (: (<> : ('&(& Ar (: (<>:, ; !), ><), ; !), ><)).

The end, :, ; !) ><)), is using the same Condition as of B (:.
Here we have replaced the dot notation ... with '& (& for any arbitrary navigation. This arbitrary navigation is replaced by concrete navigation in a each application. Note that we do not have right hand parentheses for the left hand in this navigation expression.

We have deleted an item B (: which has no previous item. Then we want to repeat this process until A is empty. This is done by attaching a schema reference S<> at the Tip of the expression that refers to the top of the expression to be repeated.

We will now write the same expression in the two-dimensional notation:

<> : '& (&	Main Condition
B	The subtraction list
:	Its first item
<> : '& (&	Sub-Condition
A	Referring to
:	this item
<> : ; !	has not a a previous item
><	Delete the first/current item of A, ie insert nothing=Delete
; !	The first item of B has not a previous
><	Delete the first/current item of B
S <> S ' ' B	If B has one (or more) items left, Go-to B and repeat the process

All the lines above are stated in the condition and instruction branches. They refer to control and write branches. The last line above does not appear/correspond to items in the control or write branches. It refers to B in the condition branch, that refers to B in the control branch.

In the write Instruction, we write blank in the two- and one-dimensional notations. Blank means : !, ie write the item Not (in the write branch) , which is interpreted as a Delete Instruction.

When all B (: items are deleted, a similar number of A (: items are deleted. Hence,

A ::= A - B.

The reason why we study these examples is that a Number is defined to be the length of a list. We have just shown how we can do addition and subtraction of Numbers. We have shown elsewhere how the length of a list may be turned into Arabic numbers.

There is a problem with the branching conditions. I have assumed that in a Condition branch we first navigate to the Top, thereafter navigate downwards. This was convenient in the above examples, but may not always be so.

10 Instruction

We have used a single Instruction to indicate Delete. A single Instruction is an Instruction without any argument.

We additionally have used
- a single Instruction to indicate navigation
- an additional Instruction to indicate Fixed insertion
- a third single Instruction to indicate copying from the Tip-s of the control branch

We'll start with discussing the single Instruction. In a three-dimensional notation, an Instruction and a Condition are along the same axis, but in different directions. Backtracking over a Condition is similar to Forward execution over an Instruction. The only difference is that you may only backtrack over a Condition if you already have executed forward over the Condition. The controlled item is tagged to indicate that the forward execution has already taken place. Sub-Conditions may be reached during forward execution of the Condition branch. The Condition-s are validated during the backtracking of the condition branch. During forward execution, the controlled items - in the control branch - are tagged by Status 2, and the conditioning item in the condition branch is tagged by Status 1. The values are Conditioned, Conditioning or blank.

An Instruction is reached via a backward execution of a condition branch.

However, the item by the Instruction in the condition branch may tagged by Status 1, as well. The additional vales are Instructed and Instructing. The corresponding write branch is tagged with Status 2.

The instruction branch consists of three segments as indicated above. Since a single Instruction without an argument is used to indicate deletion, we may use a double Instruction (><><) to indicate copying from the control branch. In a practical notation, we may use separate symbols or words to indicate the three segments. Here we discuss how we can do with a minimal set of symbols.

We will need a status to indicate which segment an item belongs to, as we have no central logic. We will need to distinguish Navigation, Fixed insertion and maybe Copying. These values may be stored in a Status 4 in the insertion branch, or may be added as values to Status 2.

Note that since we have introduced an explicit deletion symbol, ie the empty ><, we need not have deletion as an outcome of Failed Conditions. With the explicit deletion, deletion can be the outcome of a Satisfied Condition, as well. In search operations, an item will be skipped - not deleted - if the sub-condition Fails.

We use the same Status 2 in the write branch as of the control branch. A Failed Instruction has no consequence for the Conditioned item of the main Condition or for the Instructed item elsewhere. Hence the effects of Failed Instructions and Failed Conditions are different.

11 Execution overview

When executing a condition branch, we fine all control branches which each satisfies the entire condition branch. Hence, when having found one control branch that satisfies all, the execution of the condition branch has to be repeated to find more control branches until none are left.

An Instruction may be found when finding a Satisfying control sub-branch. This means that if we want the Instruction to apply only for fully Satisfied control branches, we need to place the Instruction on the conditioned item. The instructing branch may address a write branch that overlaps with the control branch.

The instruction branches are executed in a similar way as of the condition branches. The three segments of the instruction branch are initiated by >< ..., >< ... ><><. The paths, indicated by ..., may be empty. Hence, each of the segments may be empty, ie ><, >< ... ><, and >< ... >< ... ><, which means deletion of the last entry. Deletion of an entry may cause deletion of all subordinate items.

12 States and Naming

A name in front of a Condition is a local name of that condition branch. Hence, if we have a Condition on a data item, eg Driver<> ..., Driver is a local name of this Condition.

There is no distinction between a Function and other data having a Condition. However, a function has two additional features:

- a Function may have a write Instruction, ie >< ..., telling where to put the function result
- a Function may contain an explicit schema reference, ie S<> ..., specifying the mapping from any combination of arguments to the function value; the schema references may be nested

Deletion of a controlled item corresponding to A is handled as an empty Insertion, ie A ><.

An implicit deletion during insertion of a controlled item corresponding to A is carried out by a Failed Condition, ie from A <> !. During selection, a failed Condition results in exclusion from the search result, not deletion.

An explicit deletion is stated by a Satisfied Condition, ie. A (<> ...), >< or A (<> ...), >< ... ><.

A Condition may contain several sub-Conditions recursively. Each sub-Condition acts separately like the main Condition. However, the main Condition acts on its conditioned item only. A sub-Condition acts on the control items corresponding to its conditioned item. For the main Condition, the controlled item and the conditioned item are one and the same.

Sometimes we may write the conditioning item explicitly, eg A <> A ‘&(&. Other times we may omit the conditioning item, eg A <> ‘&(&. The meaning is the same. Note that the name of an item is contained in that item. The item is indicated by a colon :, its name is indicated by : (:, each of its letters are indicated by : (: (:, and each pixel by : (: (: (:.

From the above, it follows that a valid reference may be written A <>: ‘&(&. This means that the name of the conditioned item will not be tested in the Condition.

The functioning of Insertions depends on the nesting of sub-Insertions, as described above. Hence, Status 4 will prescribe what functioning to be executed.

I have been wondering if Insertion symbols should be copied into the write branch to indicate what editing has been carried out. This may follow from copying Conditions and Functions into the control branch to indicate what execution is carried out. This may be needed if capabilities of a full roll back is wanted. This is for further study on persistent databases.