Insertions and Deletions

04.04.2020, 25.03.2020 Arve Meisingset

Introduction on Insertion

Conditions and Instructions are attached to a data item. Functions are contained under the data item. The Conditions and functions have to be executed before the Insertion of the data item takes place.

The above means that there is a need to look one level down to find the functions that may affect the creation of the data item. The functions contain Conditions, but they are one level down.

When looking further down, we may find constructs that affect the above items. Suppose there is a schema of Car-s with their identifier and Colour. A Car with its identifier is created. However, the schema does not allow Red Car-s. If Colour Red is inserted, then maybe the Car has to be deleted. His is though an extraordinary requirement. Most application would reject the value Red and let the entity Car remain with an empty Colour.

If we execute bottom first, we ensure that all constraints and derivations are enforced for the creation of the data item. However, we cannot do the insertion before we have executed its subordinates. Also, it would be unfortunate to execute all subordinates, such as subordinate entities, as this execution would be unnecessary.

We might Insert en entity without its subordinate identifier attribute and value, but this does not make much sense. Hence, we must execute all Conditions, subordinate functions and identifier attributes (with sub-attributes) and values before we Insert an entity.

Note: So far, we have tagged the identifier attributes with an Id<> function. The Condition may point to the root of the name space or be default. This means that we have to look two levels down to find the Id function. An alternative would be to place the Id function directly under the entity, with a branched reference both to the root and to the attribute. This is for further study.

Introduction on Deletion

When Deleting a data item, there may be no internal Condition to Satisfy.

However, there may be internal decremental functions to activate with updating of their function values. Separate Conditions on activation of these functions may identify them.

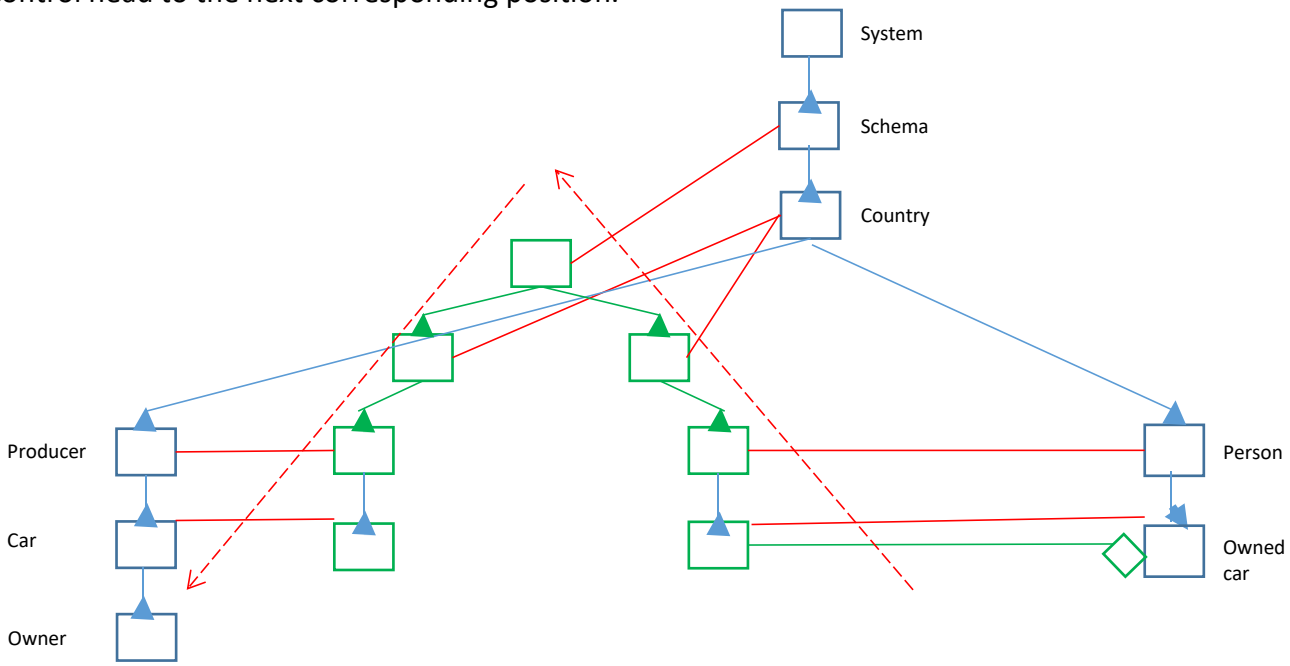
There may also be Constraints on other/external data items that depend on the existence of the current item. In order to find these Conditions or functions, we may need two-way references. This is the complex issue.

One-way references

In the Example Application document we have a schema containing two-way references.

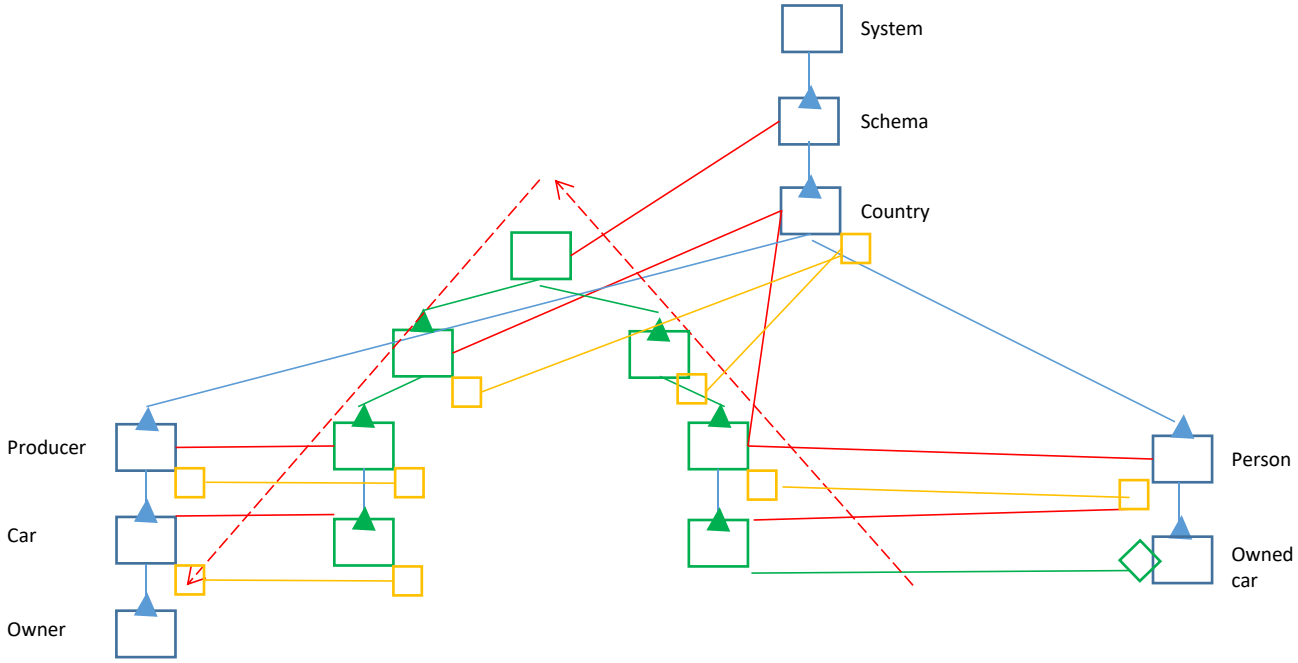
```
:                       System
    :                   Schema
        :               Country
            :           Person
                :       Owned car
                    <>  Owned car ‘Person ‘Country ‘: (Country (Producer (Car
            :           Producer
                :       Car
                    :   Owner
                        <> Owner (Car ‘Producer ‘Country ‘: (Country (Person
```

The following Figure illustrates the first reference in green, from a Condition on Owned car, to its Tip referring to a Car. The red solid lines indicate an axle moving between corresponding data items. Either the axle moves between these positions, or there is a new axle for each item in the reference, forcing the corresponding control head to the next corresponding position.



The correspondences along the axles may be understood as entanglements between the read items and the control items. We note the similarity with the two strands in the DNA. The dashed arrows show progress of the execution. The progress is the same when creating the reference as when later enforcing the reference.

The reference is during pre-compilation supplied with identifier attributes, and the Condition is moved from the Owned car entity to its identifier attribute. When ascending in the navigation, these identifier attributes are executed before execution of the subordinate entities. Note that the execution is ascending when having reached the Top both in forward and backward execution.



Note that the entity classes Owner and Owned car may have no attribute. However, sometimes it may be convenient to assign identifiers as well as other attributes. The identifier values may be 1,2,3.. and A,B,C... In classical logic, the combination of the identifiers of the two superior entities are used as the identifier of the relationship. We are not constrained to this convention. The user points out one of the Owned car entities of a Person and refers to the identifier of the Car - superior to the Owner entity.

The end user may put values into the navigation path, ie one value per identifier attribute. This extends the execution as being illustrated with a double two-colloured line between corresponding attributes and values. The Condition is now put on the identifier value, and each value along the navigation path is controlled.

Strictly speaking, the identifier values are not needed for ascending, but are needed for descending. When having two-way references, they are needed both ways. See next section.

Two-way references
We use the same example as above. However, now we study the second reference, ie from Car via Owner to Person.

We want to define the second reference to be the reverse of the first reference. We may certainly write the second reference by hand, like in the example. However, it would be much more efficient if we can derive it automatically.

We may at the Tip of the first reference, ie on the Car entry, insert an Instruction to copy the path of the first reference in reverse order up to the first Condition.

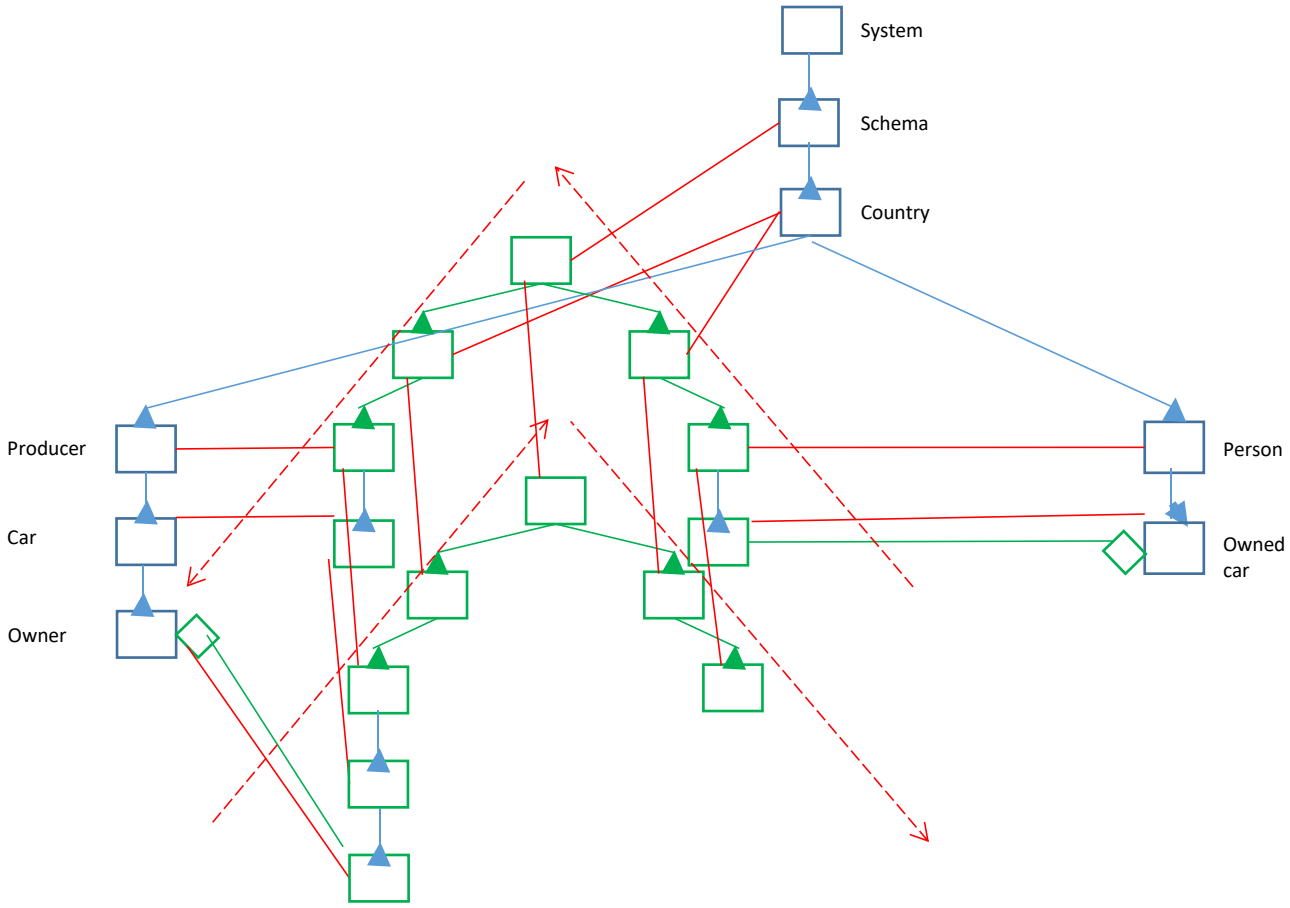
```
:                                   System
    :                               Schema
        :                           Country
            :                       Person
                :                   Owned car
                    <>              Owned car ‘Person ‘Country ‘: (Country (Producer (Car
                        /<>             (Owner <> Owner ‘R<> Car, Person) !
                        /><>< (Owner <> Owner ‘R<> Car, Person
            :                       Producer
                :                   Car
                    :               Owner
                        <>          Owner ‘Car ‘Producer ‘Country ‘: (Country (Person
                                    <> (Owned car <> Owned car ‘Person R<> Person,
                                    Car) !
                                    ><>< (Owned car <> Owned car ’Person R<> Person,
                                    Car
```

The next line under the reference shows a sub-Condition under Car that the reverse reference does NOT (!) exist. The sub-Condition is on the Owner contained in Car. The reverse reference (R) starts with Car - above Owner - and ends with Person.

If the sub-Condition is Satisfied, the reversed list is inserted into a reference starting with Owner. The hyphen in front of the R is indicating navigating up. Note that up and down are switched in the reversed list.

The R is working on the path under the read head, and not on the paths under the control and write heads. See later on instantiation.

The reversal of lists or sub-lists is a very important operation. The operation is a means to introduce consistency into the theory. If the R operator is applied twice - RR - this produces a copy of the original list. Each element of the lists may contain a sub-structure, which is not affected through these operations.



Note that Owner is the head of the reversed reference, and Owner was not contained in the first reference. Note also that the creation of the reversed list operates on the condition branch under the original read head, and it does not operate on the controlled structure - in blue. During the creation, the axle puts an instruction head on the original condition branch and the write head onto the reversed reference, starting with the Owner.

We see that the two-way reference looks like double stranded DNA strings. However, the two strands have opposite directions, and they have separate heads.

With pre-compilation of this Schema and Insertion of values into the Population, we get the following:

: System
: Schema
: Country
: Name (:)
: Person
: Name (:)
: Owned car
<> Owned car ‘Person (Name (:)) ‘Country (Name (:)) ‘: (Country (Name (: I), Producer (Name (: I), Car (No (: I))
/<> (Owner <> Owner ‘R<> Car, Person) !
/><>< (Owner <> Owner ‘R<> Car, Person)
: Producer
: Name (:)
: Car
: No (:)
: Owner
<> Owner ‘Car (Name (:)) ‘Producer (Name (:)) ‘Country (Name (:)) ‘: (Country (Name (: I),Person (Name (: I))
/<> (Owned car <> Owned car ‘R<> Person, Car) !
/><>< (Owned car <> Owned car ‘R<> Person, Car)

The Insertion commands (I) in the first and second reference are tags on the values (:) to be inserted by the end user. From these commands, an insertion of the entire reference may be derived.

The sub-Conditions and sub-Instructions are here left as of the entity level in the Schema. However, in the Population they will be supplied with attributes and values.

If each Car has only one Owner, it is sufficient to test if it has an Owner. Hence a test of the entire reversed reference is not needed.

If a Car may have multiple Owner-s, then the existence of the entire reference back to the right Person is needed. The reversal of the lists ensures that the same attributes and values are used in both references and in the logic on them. Therefore, the second reference is the right reverse of the first reference.

Deletion
In the above example, only the role labels Owned car or Owner need to be deleted. The rest of the references are contained in these role entities and will automatically be discarded together with the role entities. Hence, the Deletion statements are simpler than for Insertion.

<> Owned car ‘Person (Name (:)) ‘Country (Name (:)) ‘:
(Country (Name (:), Producer (Name (:), Car (No (: D))
/<> (Owner <> Owner ‘R<> Car, Person)
/><><><>< (Owner)

Note that the deletion of the No value implies deletion of the entire first reference.

Instantiation

Instances in a population are executed the same way as of their classes in the schema. The population contains a schema reference to the schema. The schema reference is a Condition.

The population will certainly satisfy the reference to its schema, as this is just navigation. However, the population needs only satisfy insertions from the root of the schema, and needs not to satisfy every branch of the schema to their Tip-s. Every sub-tree from the root of the schema will do.

The population data are homomorphic to the classes in the schema or of any recursively higher meta schema. Homomorphic means a many-to-one mapping. However, the mapping does not need to be onto every detail of the schema.

When executing the population, the read and instruction heads will be in the schema, while the control and write heads will be in the population.

The Contents population will contain Conditions that correspond to references in the External terminology population. These Conditions are used to traverse the External terminology population, or to insert or delete the references therein.

The Contents layer needs not contain any constraint or derivation, as all these are covered by the External terminology layer, or are deferred to the Internal layers. The Contents layer may contain constraints on the search, eg that the searched Car-s are either Red or Blue.