Execution model

Joop Ringelberg 14-01-22 Version: 1

Introduction

The Perspectives Language is partly declarative. It lets the modeller describe contexts, roles, properties, perspectives and more, on the type level. But it also contains *assignment statements* to update *instances* of those types, to track a changing world. Assignments are always carried out by an end user, either by direct manipulation through the user interface (e.g. by dropping a role onto another, signalling the PDR that the former should fill the latter), or by executing *actions*, which are pre-packaged sequences of statements. But there is yet another mechanism responsible for executing assignment statements and that is actions carried out automatically on behalf of some user when the state of a context or role changes.

More specifically, when a resource enters or leaves a state, actions may be carried out automatically. This raises some questions, a.o.:

1. Does state change statement by statement, during action execution?
2. If a resource moves in a new state S because of the execution of a statement in a sequence, when exactly will statements to be executed on entering S be carried out?
3. How does resource creation and deletion relate to state change?
4. When a context and its roles are deleted, in what order are `on exit` statements executed?

In this text we give answers by describing our design decisions. Together, they give the *execution model* of the Perspectives Language.

Statements execute in order of appearance

Consider the following action:

```
action A
      create role SomeRole
      SomeProperty = 10 for SomeRole
```

where `SomeRole` is functional and `SomeProperty` is Numeric. Assuming the context in which this takes place has no instance of `SomeRole` yet, this will work perfectly, because the statements are executed *in order of appearance*. So an instance of `SomeRole` is created; then, on the next line, the (minimal) query `SomeRole` returns this instance and it will acquire value 10 for its property `SomeProperty`.

Notice that each statement is executed in an environment that has been changed by the statements above it, in the program text.

States associated with resource creation and deletion

Each context or role has at least one type. The modeller may add Aspects to a type, thereby adding more types. When a resource is created, it immediately afterwards enters the *root state* associated with each type.

A type's root state may be filled in by embedding an `on entry` or `on exit` clause directly in its body:

```
thing SomeRole
    on entry
      notify SomeUser
        "An instance of SomeRole has been created!"
```

In contrast, when a resource is deleted, it *first* leaves all states it is in (its *active states*). Only then is it actually removed. More on this topic below.

Staged evaluation of state changes

Let's suppose that `SomeRole` had been defined like this:

```
thing SomeRole
    on entry
      do
        AnotherProperty = 20
```

And let's change action `A` as follows:

```
action A
    create role SomeRole
    SomeProperty = SomeRole >> AnotherProperty + 10 for SomeRole
```

This will compile, but after executing A, `SomeProperty` will not have a value. This is because execution of statements proceeds in *stages*. Let's work this out for our example:

- Stage 1 sees execution of
    - the `create role` statement (but *not* its `on entry` clause!)
    - the property assignment.
- Stage 2 sees execution of the `on entry` automatic action of the newly created `SomeRole` instance[1].

[1] The argument query `SomeRole >> AnotherProperty` of the property assignment in the action consequently finds no value, hence the entire addition expression has no value.

In other words: when a resource changes state during an execution stage, *the effects of this state change will only occur in the next stage*. That is, automatic actions due to state changes are postponed till all statements scheduled for the current stage have been executed.

Why this design? Our example is so small that it is easy to form a picture in our mind of how the `on entry` action would happen, before executing the property assignment. But things get very complicated quickly. Actions carried out automatically may trigger more automatic actions, recursively. The code describing this may be scattered all over a model. In no time at all it is very difficult to piece together what is happening where, when and why.

We've chosen this model because it allows the modeller to form a clear picture of what will *actually happen locally*, that is, the direct consequences for the state of the resources affected by his statements.

This is not to say that understanding automatic actions is always easy.

Another way of looking at this execution model is that state changes of various resources happen in parallel. If a statement sequence triggers state change with automatic actions in two other resources, these will execute (as if) in parallel. This means one should never count on a particular order of their execution!

Removing a role instance

Roles form the connections between contexts. A role can be filled by another role (usually a role from another context) and a role can fill many other roles. These connections are bi-directional in the sense that a query can traverse them

- both from a role to its filler, and
- from a filler to the role(s) it fills.

Think of these connections as pointers in and out of the role instance. It is convenient to name these two types of pointers:

- a role can have many *fills* pointers;
- but it has only one *filledBy* pointer.

When a role is removed, *we only detach the pointers towards the role instance*. We need not remove the outward pointers; we're going to throw the role away, anyway. So, for any role instance to be removed,

- we remove the filledBy pointer from all roles that are filled by it, and
- we remove the fills pointer from its filler.

A user is endowed with a set of perspectives that are qualified by role- and property-verbs, including verbs that affect roles:

- `Remove`

- `Delete`
- `RemoveFiller`
- `RemovePropertyValue`

A user can only remove roles she has a perspective on with verbs `Remove` or `Delete`. Notice that `Remove` and `Delete` effectively imply `RemovePropertyValue` and `RemoveFiller`.

State evaluation

Before a role is removed, it exits all its active states. The deepest nested active state(s) are exited first, meaning their `on exit` actions are executed first. The actual algorithm is formulated the other way round:

1. For each state:
    a. exit active substates;
    b. execute the `on exit` action.

And this starts with all root states of the role. *All these actions are executed in the same stage!* Subsequent automatic actions due to state changes caused by these `on exit` actions are *all postponed to the next stage*.

Furthermore, the automatic actions are execute *before the pointers into the role are removed*. This means that the statements are executed on the structure as it exists before removal. This is important, because it allows the modeller to modify remote parts of the web of roles and contexts from such an on exit action.

Synchronization

There may be peers in roles that have a perspective on the role instance that is removed. They should be informed. We compute these peers *using the network prior to the removal of the resource!* This should be obvious: we find peers by following outgoing links. When the resource is destroyed, there are no outgoing links left.

Removing a context instance

A context is embedded in the network of contexts and roles through the connections of its roles. In order to remove a context, we can simply

- remove the incoming links (fills and filledBy) of all of its roles
- and then throw away the context and all its roles.

In other words: the internal structure of the context does not need to be torn down to remove the context (but see below for role state evaluation).

A user is endowed with a set of perspectives that are qualified by role- and property-verbs, including verbs that allow her to remove information:

- `Remove`
- `RemoveWithContext`[2]
- `RemoveWithContextLocally`
- `Delete`
- `DeleteWithContext`
- `DeleteWithContextLocally`
- `RemoveFiller`
- `RemovePropertyValue`

In order to initiate removing a context, a user must have a perspective on a contextrole filled with that context, with the verb `RemoveWithContext`, `RemoveWithContextLocally`, `DeleteWithContext` or `DeleteWithContextLocally`. These are the only verbs that allow removing a context.

Notice, that, in effect, `RemoveWithContext` implies `Remove`, `RemoveFiller` and `RemovePropertyValue`. `DeleteWithContext` implies `Delete`, `RemoveFiller` and `RemovePropertyValue`. However: this does not mean that a user with a `RemoveWithContext` perspective therefore has the right to remove individual roles, for example.

The verbs that end in `Locally` permit the user to remove or delete their own copy of a context, but peers are not required to do the same. However, they must annotate the user roles with the removing peer such that they are no longer involved in the synchronisation process.

State evaluation

Before a context is removed, it exits all its active states. This happens in exactly the same way as for roles: deepest nested active states are exited first and all actions `on exit` are executed in the same stage.

Consequently, *all statements are executed while the context is still fully intact*. For each statement, the modeller can ‘reach out’ of the context and change things there.

Can he change the context that is about to be removed? He could, actually; and this may affect the states that are subsequently exited. Another reason to modify a resource that is about to disappear would be to enable conditions for *statements that follow*. The modeller is strongly advised against this[3], because it makes removing a context less transparent. The acting user removes the context based on what she can perceive of it; if

[2] `RemoveWithContext`, `RemoveWithContextLocally`, `DeleteWithContext`, `DeleteWithContextLocally` are not yet implemented in InPlace v.0.12.0. We will introduce them to indicate whether the perspective allows removing (or deleting) the bound context, too.
[3] We may add, in the future, a compiler check that warns against this.

it is modified on the fly, she actually removes it in a different state from what she thought.

None of the modifications to the context or its roles itself, make a material difference for peers: they throw away the entire context. Of course, modifications *outside* of the context that is removed, will be communicated to the peers.

Embedded role state evaluation

What about the roles embedded in the context? They are removed, too, so they should exit their active states as well. The question is: do we exit role states *before*, or *after* context states? We choose to exit them **before** exiting the context states.

Synchronization

Removing a context is a very powerful operation. All peers are required to remove their local version of the context, too, completely. Even if it holds more information than it did for the acting user, they should still remove it entirely. Synchronization is therefore simple, because it consists of a single delta that instructs the receiver to completely remove the context.

Consequently, we do not need to collect deltas when we actually remove the context, detaching it from its surroundings. All these deltas are superseded by the powerful instruction to remove the context[4], making them redundant.

Which peers that should be informed about removing the context? This is the union, over all roles, of the peers that should be informed when the **incoming pointers** to the role are removed.

On collecting those users, we *should* ***not*** *already remove the pointers*. Otherwise, for each consecutive role instance, the computation is carried out on a diminished context representation. A simple example shows why that is a problem.

Consider a context with two user roles, filled by different peers of the acting user (who removes the context). Clearly both peers have to be informed that they no longer fill a role in the context after its removal. However, were we to remove the roles one by one, it is obvious that

- the removal of the first peer can be communicated to the second peer (who is still in the context)

[4] We may ask ourselves: can there be a peer with a role that fills a role of the removed context, without having a representation of that context? The answer is no, because *each reference must be locally resolvable*. That is, the ‘fills’ link of that peer must point to a role that is present in his installation – and hence the context is present, too. The same holds for a link in the other direction.

- but the removal of the second peer would never be known to the first peer (who is, after all, no longer present in the context by this time).

So, in effect, we first run a kind of *simulation* of removal of the context:

- first we collect peers that should be informed when incoming pointers to the contexts' roles would be removed;
- then we exit, for each role instance, its active role states;
- finally we exit all active context states.

Only then

- do we send the context removal delta to the collected peers;
- and we finally actually severe incoming pointers and remove the resources.

Synchronization may need *passing on*

The user that removes a context, may not have a perspective on all users in that context. As a consequence, he cannot inform all those concerned about its demise. This means that we require the synchronization mechanism of *passing on*. This is that some users receive the delta not from its originator, but via other users in the context.

Refining understanding of resource removal

Reconsider these three important rules of the execution model:

1. Statements are executed in order of appearance in the model source text;
2. Statements in an on exit clause are executed *before* the resource is actually removed from the structure of contexts and roles.
3. When a resource changes state during an execution stage, *the effects of this state change will only occur in the next stage*.

These three rules are not compatible, as we will illustrate with this example:

```
thing SomeRole
    on entry
      do
        remove role currentcontext >> AnotherRole
        create role YetAnotherRole in currentcontext
thing AnotherRole
    on exit
      do
        create role TheThirdRole in currentcontext
```

Clearly, rule 1 dictates that the `AnotherRole` instance must be gone by the time that the instance of `YetAnotherRole` is created. However, rule 3 says that the action on exit of the instance of `AnotherRole` can only be executed in the next phase (that is, after both statements have been executed). And rule 2 states that these actions must be completed while the `AnotherRole` instance is still there.

Solution: monotonic inference first

We solve this problem by, in effect postponing the actual removal of resources until the very last moment. This means that, seen per `on entry` or `on exit` clause, ***removal statements always come last***. So the first part of our example is equivalent to (and should be written as):

```
thing SomeRole
    on entry
      do
        create role YetAnotherRole in currentcontext
        remove role currentcontext >> AnotherRole
```

Removal comes last. This holds recursively, for 'nested' automatic actions. As a consequence, execution of automatic actions is divided in two steps:

I. First, all additions to the structure are made, recursively, all the while postponing any removal encountered, while yet executing all on exit clauses of resources that are to be removed[5];
II. Then, in one fell sweep, all resources marked for removal are detached from the network.

This may, of course, trigger fresh state changes in some resources, so then the entire process begins again.

Complicated though this may seem, it actually has a desirable characteristic: to understand the execution of automatic actions in a model, you can try to understand the *additions* independently from the *removals*. Both can be understood as *monotonic inferences* from the state of all resources. This is good, because it means we can analyse what happens in terms of ordinary mathematical logic.

In other words: you can use logical inference to determine from a given overall state:

- what will be added to the structure, and, independently,
- what will be taken away from the structure;

Then consider the new state that arises when new things are first added and then some others are removed[6].

[5] And, obviously, any actions `on entry` as well.
[6] Actually, it does not matter whether we first add and then remove the results, or the other way round. This is because the system is robust enough not to fail if we try to add a role instance to a context that does not exist. But it's certainly more elegant and efficient to first add and then remove.

On the implementation

We can implement this in Inplace v0.12.0 as follows:

1. compile a removal statement not directly to the `removeRoleInstance` function (module `SaveUserData`), but instead schedule both the on exit and the removal in the current `Transaction`.
2. In `runStates` (module `RunMonadPerspectivesTransaction`), run the `exitStates` function on all instances scheduled for exit (this is what we actually do right now).
3. Finally, in `runMonadPerspectivesTransaction'`, do the actual removal. Currently, we just remove the resources from cache and from Couchb; now we must detach them first.

This last step will generate new information in the current `Transaction`, which then again needs executing.