Assignment

Joop Ringelberg 12-11-19, 17-01-22 Version: 3

Introduction

A user role can have a perspective on roles in- or outside his own context. This brings with it the possibility to change those roles, i.e. to add or remove instances and to add or remove values to or from their properties.

A user agent exclusively changes state through (graphical) user interfaces. Automatic actions on his behalf when state transitions occur, have to be programmed however and to do so we need a vocabulary. This text contains the design for that part of the Perspectives language[1].

Statements that change the state are called *assignments*. There is a small vocabulary for assignments on roles, including a number of special *operators*: `move`, `remove`, `delete`, `bind`, `bind_`, `unbind` and `unbind_` and the keyword combinations `create role` and `create context`. There is a smaller set for assignment on property values: `=`, `=+`, `=-`. We also use `delete` on property values.

The default case is assigning to roles and properties in the context that contain where the state transitions are modelled. We design our language to be intuitive for that default case, extending it for the situation where state of other contexts is changed (see the design text *Perspectives across context boundaries*).

Design

Current context

An automatic action can happen only in a state transition. States are defined with conditions, which are just Boolean queries. The assignments that make up the automatic effect are executed when the condition becomes true. Conceptually, the system observes each context- and role instance. As soon as the state of one of these instances changes in such a way that the condition of one of its defined states evaluates to true, the automatic effect is carried out.

It is therefore important to realise, when reading below about the language for effects, that the effect is computed *relative to some context instance*. All queries that are part of effect statements are executed with that context instance as their starting point. Below, we call it the *current context instance*. The current context can actually be used inside an expression. Reflect on the nature of expressions: they trace a path through the network of

[1] Also note that the API of the Perspectives Distributed Runtime exposes these functions.

contexts and roles. Using the keyword `currentcontext`, the modeller can re-base a subexpression to the context instance that the rule is executed in.

At the same time, the rule is defined in a perspective. The perspective has an object: a role in the context. When the automatic effect is defined on a role state transition, that role is the object. When it is defined on a context state transition, all instances of the role in that context are the object. These instances are bound to the query variable `object`. It is available in the state condition and in the statements of the effect. It is useful, for example in the situation where one wants to bind that object to a Role instance.

Assignment statements for roles

Below, we'll find that assignment operators on roles take one or more arguments, that specify what they operate on. For example, in the case of the remove operator its first argument says what we want to have removed automatically. These arguments can be arbitrary expressions – as long as they select roles of the same type as the Object of the Perspective. The system, after all, must operate within the limits set for the user that it works on behalf of!

Implicitly, we give the user that the system carries out an automatic effect for, the role- and property verbs required to do so.

Remove

Let's consider the situation where we have some instances of a role that we want to remove from their context. We have selected them with a query, `<roleExpression>`:

```
remove <roleExpression>
```

Remember, `<roleExpression>` is a query applied to the current context instance. For example, this could be such an expression:

```
filter SomeRoleType with Completed = true
```

This query will select some instances of `SomeRoleType` in the current context instance. If we prepend the keyword `remove` to that expression, all those role instances will be removed from the current context instance.

This works on any context, because role instances are bound to a context. We need not say *from which* context we want to remove them. So in order to remove role instances from an embedded context, we'd write for example:

```
remove filter AContextRole >> binding >> context >> AnotherRole with Completed = true
```

We select a context role, move to the context that is bound in it, move to one of its roles and filter it like before. As before we end up with a number of role instances: these will be removed *from the embedded context*.

What if we want to remove a role that has been added to a Database Query Role? In order to enable the system to decide whether removing the (external) role instance is allowed, it needs to know the Calculated Role type that it should be removed from. So we get:

```
remove <roleExpression> from RoleType
```

CreateRole

Create a new role instance in this way:

```
createRole RoleType
```

Here `RoleType` is not a query, but the identifier of the role we want to create. We can use an unqualified name but it will have to resolve in the type of the current context. The new role instance is attached automatically to that context. In that sense, createRole is an assignment statement, too.

We can also create a role instance in another context:

```
createRole RoleType in <contextExpression>
```

`<contextExpression>` selects one or more context instances. `RoleType` now has to resolve in the type of those instances.

Move

Usually, when we add a role instance, we create it at the same moment. However, it is possible to move role instances from one context instance to another. Then we use the `move` operator:

```
move <roleExpression>
```

This removes the selected instances from their origin context and adds them *to the current context*[2]. To move to another context, we use an extra clause:

```
move <roleExpression> to <contextExpression>
```

Again, in order to make this useful, `<roleExpression>` should select from another context than that identified with `<contextExpression>`. Notice that `<contextExpression>` can only select a single context.

Delete

Sometimes we just want to remove all instances of a role. Then we use `delete`:

```
delete role <roleType> [from <contextExpression>]
```

Select the instances to be removed. Optionally, to select from another context, add the `from <contextExpression>` clause. Be careful with deleting the instances of a Database

[2] Note that if <roleExpression> selects instances in the current context, this statement does not change state!

Query Role: all instances that are not bound to some other role will be removed from the users Bubble!

To remove all values of a property, use a slightly different syntax:

```
delete property PropertyType [from <roleExpression>]
```

Notice that we do not provide a query to select instances. We just want to remove all values of the property. By default, we remove them from the *current object set* (see *Object of the Perspective* below). To remove from another role, add a clause:

```
delete property PropertyType from <roleExpression>
```

Obviously, `PropertyType` has to resolve in the type of roles selected by `<roleExpression>`.

Bind

To fill a role with another role, use `bind` (when A fills B, we say that B *is bound to* A. A is the *binding*; B is the *binder*). Bind like this:

```
bind <binding> to RoleType
```

```
createRole RoleType filledBy <binding>
```

Here, `<binding>` selects instances of a role (the bindings) whose type must be equal to, or more specialized then, the possible bindings of `RoleType`. *A new instance of* `RoleType` *will be constructed automatically* (the binder) and attached to the current context.

To bind in another context, add a clause:

```
bind <binding> to RoleType in <contextExpression>
```

Again, `RoleType` should resolve in the type of the instances selected by `<contextExpression>`.

If `RoleType` happens to be functional, `<binding>` must evaluate to a single role instance as well. Notice that `<contextExpression>` may evaluate to multiple contexts; we just bind in all those contexts.

Bind_

The variant `bind_` can be used to bind an instance of a role in a previously existing instance of `RoleType`:

```
bind_ <binding> to <binder>
```

The first expression (`<binding>`) selects a role instance that is going to be bound to the role instance selected by the second expression (`<binder>`). Notice the singular: this operation only works on singletons[3].

[3] If we allowed more bindings and binders, it would be unclear what should be bound to what.

Obviously, the binder must be legally able to attach to the binding. That is, the possible bindings of the binder must be equal to or more general than the type of the binding.

To bind in another context, just select binders in another context.

Unbind

The inverse of `bind` is `unbind`. Notice that `unbind` does not remove anything. Both the binder and the binding remain attached to their contexts.

```
unbind <roleExpression>
```

Here, `<roleExpression>` selects role instances as before. But do we consider them as binders, or bindings? Both are possible. By convention, we choose them to be bindings (fillers) and thus we release them from the roles that bind them (filled by them).

Notice the plural. A role can be bound many times, in many different other roles. By just using an unqualified unbind, we break all bonds that this instance has. Usually, we want to be more selective and this we achieve with another clause:

```
unbind <binding> from RoleType
```

Now, we just release the instance from a particular type of binder. Still, this is across all instances of the context with that type of binder. We can't be more selective with `unbind`, but we can with `unbind_`.

On removing the last binder of an external role, the context it belongs to may be removed, too! This process can cascade recursively to nested contexts.

Unbind_

Remember that `bind_` allowed us to select roles that become a binding and a binder respectively. Similarly, with `unbind_` we select a role that *is* a binder and a role that *is* a binding and break them apart:

```
unbind_ <binding> from <binder>
```

As with `bind_`, this only works with singletons.

There is another use case for `unbind_`. It is possible to bind a role instance to *more than one* other role instance, of different types. This enables us to create role instances as combinations of property packages, as it were: think of a role at the pharmacy that you'll fill both as patient and as bank account holder. `Unbind_` allows us to pick those multiple bindings apart. We can just remove, say, the bank account role from the pharmacy client role.

As with `unbind`, on removing the last binder of an external role, the context it belongs to will be removed, too! This process will cascade recursively to nested contexts.

‘missing’ statement types: add and set

One might expect an operator `add`, to add role instances to a context. However, just where would these instances come from? We don’t need `add` for creating instances, because `createRole` ‘adds’ the created instance to the context anyway. The only possible source for the right kind of instances would be from another context than the current. However, for this we have the `move` operator. Notice also that a role instance can only be attached to *one* context instance. So to move, we have to detach and re-attach somewhere else, preferably in a single transaction. This is precisely what `move` accomplishes. By omitting the `add` operator, we protect the modeller from mistakes without compromising what he can express.

A `set` operator would replace the current instances of a role with a new set. There are use cases for this operator, but these can always be programmed by a combination of `delete` and `move` or `delete` and `createRole`.

Creating contexts

One of the design goals for Perspectives is that all context- and role instances must be *reachable*. This can be attained by direct indexing (e.g. a role is directly linked to its context), by deploying an indexed name[4], or by a database query that retrieves instances of a particular type[5]. Such a query has to be the expression by which we define a Calculated Role. To differentiate such database-query-based Calculated Roles from those that are defined by a path query, we call them Database Query Roles.

The assignment statements for roles preserve this quality. In order to do the same for freshly constructed contexts, we have to bind them directly to a context role in another context, or we must ensure that they are available through a Calculated Role somewhere that performs a database query.

Notice there is no operator to remove a context. Contexts are deleted if they are no longer bound, or, in the case of a context that was never bound but added directly to some Database Query Role, as soon as they are removed from such a role (a role based on the same type).

Create context

With `create context`, we create a context of the given type and bind it to a new instance of the given Enumerated Role type in the current context:

```
create context ContextType bound to RoleType
```

In order to bind it in another context, we add a clause:

[4] An indexed name has a different extension (reference value) for each end user, e.g. My System.
[5] They must be external roles, possibly filtered. Database Query Roles must be context roles.

```
create context ContextType bound to RoleType in <contextExpression>
```

It goes without saying that actually the external role of the fresh context is bound to the new role instance.

If `RoleType` is a Calculated Role that qualifies as a Database Query Role, no role instance is created to bind the new context. However, we require `ContextType` to be equal to or a specialisation of the result type of the Database Query Role.

Create context_

Like with `bind_`, we may be in the situation that we already have a role instance that acts as binder. For that case, `create context_` creates a context instance of the given type and binds it to the role instances selected with `<roleExpression>`.

```
create context_ ContextType bound to <roleExpression>
```

Assignment statements for properties

The object of the perspective

For role assignment, we discussed the importance of the current context. For property assignment, a similar importance is attached to the *object of the perspective*. Remember that automatic actions are run for a user role having a perspective. A perspective has an object. The object is selected as a is query applied to the current context (it follows that there may not be an object and that there may be multiple objects).

Again, when the automatic effect is executed, there is a *current object set* (possibly empty). A the assignments are executed on each element of that object set in turn, binding an instance to the query variable `object`. It is available in the condition of the role state and in the effect making up the automatic action.

When we change the values of a property, we really change a role's properties. If not stated otherwise, we change the properties of the current object set.

Operators

For assigning values to properties, we use a number of infix operators: `=`, `=+`, `=-`. We also re-use the `delete` operator we've seen for roles, but with an extra keyword `property`. However, property values are not moved or created, neither bound nor unbound.

=, =+, =-

The syntax for these three operators is the same. For example:

```
PropertyType =+ 10
```

would add the value 10 to the existing set of values for `PropertyType` for each element in the current object set.

In order to change the property values of another role, we provide an extra clause:

```
PropertyType =+ 10 for <roleExpression>
```

Here, `<roleExpression>` is a query executed on the current context. Of course it can select roles outside the current context, too.

An expression can be used on the right of the operator:

```
PropertyType =+ SomeRole >> AnotherProperty
```

The meaning of this expression is: add the value(s) of `AnotherProperty`, for each of the role instances of `SomeRole`, to those of `PropertyType` (of the same instance). The query expression is evaluated relative to the current context.

Delete

This is how to delete all values for a property on the instances in the current object set:

```
delete property PropertyType
```

And here is to delete the values on another role instance:

```
delete property PropertyType from <roleExpression>
```

Why there is different syntax for properties and roles

Superficially, assignment does not look that different for roles and properties. So why not adhere to the same syntax for both? There are three reasons:

1. There are assignment operations that work on roles but not on properties: `bind`, `bind_`, `unbind`, `unbind_` and `move`.
2. There are assignment operations that work on properties but not on roles: `=` (`set`) and `=+` (`add`).
3. The `remove` operator is quite different for roles than for properties.

To remove a role, we have sufficient information with:

```
remove <roleExpression>
```

The expression identifies the role instances that we want to remove. They are represented internally by identifiable data structures that we can find and destroy. Moreover, we can look up any references to them, so we can clean those up, too.

In contrast, to remove a property value, we not only have to find the role instance that bears the values to be removed, but we also need the name of that property (and, of course, the values to be removed). So we must write down both a `<roleExpression>` **and** the name of the property (an `EnumeratedPropertyType`):

```
EnumeratedPropertyType =- <valueExpression> from <roleExpression>
```

(we can omit the `<roleExpression>` from our expression if we want to operate on the current object set, but to effectually remove the values we do need role instances!).

These differences are great enough to justify different syntax for assignment to roles and assignment to properties.