Rules

Joop Ringelberg 26-11-19 Version: 1

Introduction

The Perspectives Design Language is a *declarative language*. That is, its expressions describe a part of reality. As such, descriptions are static, while reality is dynamic. Hence there is need for ways to adapt descriptions to that changing reality. The PDL therefore incorporates a non-declarative part, the so-called *assignment statements*. An assignment statement can be *executed* and the meaning of that is that a description changes in a non-additive way[1].

This text is about when assignment statements are executed.

Rules

A rule is a syntactic structure with the following general form:

```
If <condition> then <action>
```

Here, `<condition>` is a PDL expression that evaluates to a Boolean value. A note on expressions: each expression denotes a *query* that is applied to a particular object, the object being either a context or role. Conditions are applied to contexts.

`<action>` is a series of assignment statements or a `let*` expression with a body consisting of assignment statements.

A rule is defined for a bot, that serves a particular user in a context.

Rule evaluation and execution

A rule is said to be **evaluated** when we apply the condition-query to a context instance. Which context instance? An instance of the type that the rule is part of.

If the condition evaluates to `true`, the rule will be **executed**. This means that the assignment statements on its right hand side are executed.

A rule will be evaluated (and possibly executed) each time that its condition could evaluate to a different value[2].

If a rule is evaluated and executed, it is said to be *fired*. A rule that is evaluated is said to be *triggered*.

---

[1] Merely adding facts to a description changes it as well and that partly covers changes in reality. But because we do not add a time parameter to our expressions, adding facts does not cover all necessary changes.

[2] See the text *Overview of the state change mechanism* for an explanation on how this works.

Rule re-execution

The condition of a rule is evaluated in a given state of the context, *for each instance of the role that the bot perspective is on*. So the condition evaluates to true or false for a given state and value of the variable 'object'. Now there may be many states of the context that give, for a particular condition and value of object, the same truth value. Let's lump them together into a single macro-state defined for that condition and object. Obviously, there will be exactly two such macro-states (for each condition and object): a 'true' state and a 'false' state.

Only when the context state moves into the 'true' macro-state, will the right hand side of the rule be executed. That implies that if the state first moves to the 'false' macro-state and then back again in the 'true' macro-state, the right hand side will be executed again.

In other words, if the condition is true and remains true for a given object when the context state changes, the rule does not fire.

Context creation

When a new instance of a context is created, all bot rules for the user creating it will be executed.

Mechanisms

The main question of concern here is: how do we arrange things such that rules are triggered when appropriate? At some point, the PDR implementation used *dependency tracking* to establish rules that should be triggered. But this only works for rules that have been triggered at least once before. Context instances with rules that are not loaded into cache, obviously do not satisfy that condition, yet there may be changes in the underlying network that make their conditions change value. The current implementation deploys *query inversion* to calculate which rules to trigger. This is explained in detail in the text *Query Inversion* and to a lesser extent in *Overview of State Change Mechanisms*.

Notice that this mechanism is rather coarse in the sense that it finds all context instances that have *some relevant change for the owning user*. In particular, if a bot has two rules, one of which should be triggered, nevertheless both rules will be evaluated. We have no way of knowing what rule conditions will yield a new value.

Example: rule with constant condition

Lets suppose a rule like this:

```
bot: for Self
  perspective on: Self
    if true then
      createRole SomeRole
```

What happens when an instance of this context is created? Actually, nothing will happen until role instances are created, too. These will trigger rule evaluation. As its condition always evaluates to `true`, it is immediately executed.

This rule will fire each time it is triggered. As any change in an instance of this context that is in the perspective of Self, will trigger the rule, it effectively fires whenever the context changes visibly for Self.

Note that a rule with a constant condition of false will never be evaluated or executed.

Selecting the rules for the current user

A rule belongs to a bot and a bot represents a single user. Any context may have many users and consequently many bots. On the computer of user A, only the rules belonging to his bots may ever be executed. Otherwise A would execute actions that are not in his perspective!

How do we select the right rules?

Let's call the user that operates a particular installation of the PDR the *current user*. Each context holds, by definition, a role instance that represents the current user. We establish that role instance as follows:

- A role instance that is filled by a role that represents the current user, itself represents the current user;
- The system User role represents the current user.

Thus, we can recursively work out for any role instance whether it represents the current user, or not.

Notice that there may be multiple roles that could be played, in principle, by a particular user. However, he will hold only one of them.

Notice, too, that a user role may be *calculated*. The universal Guest role is an example. It is the modellers duty to calculate a Guest role in such a way that it is empty if the user holds another role! For example, in the Invitation context, a user holds the Guest role as long as he is not either the Inviter or the Invitee.

Indexed information: the IsMe property

We can cache the result of this computation by annotating each role instance with a Boolean property indicating whether it represents the current user, or not. This Boolean – let's call it `IsMe` – is a prime example of *indexed information*. For any context, the role instance that represents myself is a different one for each user! Indexed information is never shared with others.

It may come as a surprise that we cannot achieve this with a Perspective. Consider a role Passenger in the context of a TrainJourney. The `IsMe` property should be in the

perspective of Passenger. But then each user who is a Passenger could see the value of this property for all other Passengers. That itself is not a problem, but from my perspective, each other Passenger should not be me – and this is not how *they* see it!

So indexed properties are a special mechanism[3].

Indexed information: the Me role

For a given context, in order to set up rules, we have to establish which role type the current user plays – that is, of which there is an instance that represents the current user[4]. This is not a difficult computation: just scan all role instances for the one that represents the current user and take its role type.

Again, we can cache this information by establishing a particular role, `Me`, in each Context. Again, this is indexed information.

Setting up IsMe and Me

The very moment that a user gets involved with a context instance, we can set up the `IsMe` property and the `Me` role. There are but two situations in which this happens:

- As he becomes one of the User roles in that Context;
- As he is assigned to a User role in that Context.

Becoming is a particular Perspectives Verb that we use to indicate the following scenario:

- The user creates a new instance of a User role in a context instance;
- The user fills that new instance with a role instance that represents himself.

Obviously, setting `IsMe` and `Me` is easy in that situation.

If another user fills a new user role instance with a role that represents yourself, your PDR will receive a Delta detailing that information. Your PDR creates a new Role instance and fills it with the role instance as specified in the Delta. If this filling role instance has the `IsMe` property, the new role instance should have it, too. Moreover, the context of that new instance should have the Me role filled with that role instance, too.

We cover both situations with extra code in the bind operation:

If the filler has `IsMe`, the filled role should have `IsMe` and its context should have the filled role as `Me`.

[3] As of the time of writing, I do not know whether this mechanism should be available to modelers. It may be purely internal.
[4] We assume there is just one role played by a particular user in each context.

Technical

This paragraph details the actual setting of `isMe` and `me` in code. It turns out that there are just three modules that cover all the `isMe` and `me` computing and accounting.

isMe

The `isMe` property is computed on constructing a role, and on changing its binding.

Roles are *constructed* when a CRL file is parsed. In the parser, the function `Perspectives.ContextRoleParser.userData` computes that value. Roles are constructed, too, with the function `Perspectives.BasicConstructors.constructRol`. This function computes the value of `isMe` for the new Role.

Roles are *bound* and *unbound* in the functions `Perspectives.Assignment.Update.setBinding` and `Perspectives.Assignment.Update.removeBinding`. Both functions recompute the value of `isMe` for the role whose binding changes.

Me

All functions that modify a context by adding or removing role instances, are in the module `Perspectives.Assignment.Update`. These are the functions that manipulate the context:

- addRol
- removeRolFromContext
- deleteRol
- setRol
- moveRoles

All these functions set me of the changed context, if the occasion requires it.

Transactions

Up till now we've evaded the question of how Assumptions are actually changed, so rules are triggered and executed. As it is designed, the functions that change Assumptions save deltas in a Transaction. A `Delta` is a small data structure detailing the change to a context, role binding or property value. Also, context- and role creation and deletion is administrated in a Transaction, too.

At times the code runs a Transaction. This means, among other things, that Assumptions are distilled from the Transaction, rules are traced that depend on them and then run.

The consequence is that functions that actually change state, must save a record of this change in a Transaction.