Implementing the Functional Reactive Pattern

Joop Ringelberg 06-02-20 Versie: 1

Introduction

The perspectives Distributed Runtime implements the *functional reactive pattern*. To be more specific, it keeps requests issued by clients up to date with changes of the underlying structure of context- and role instances. This text explains how this works.

The underlying network and queries

We can think of context- and role instances, and sets of property values, as nodes in a graph. Property value set nodes (short: property nodes) are terminal nodes. The others are connected by role binding and by the relation between a context and its roles. In Figure 1 below, we've drawn contexts as rectangles and nodes as circles. Nodes are drawn within the contexts they belong to, or just outside if their context is of no concern. The external node of a context is drawn outside its rectangle and is connected to it with a line.

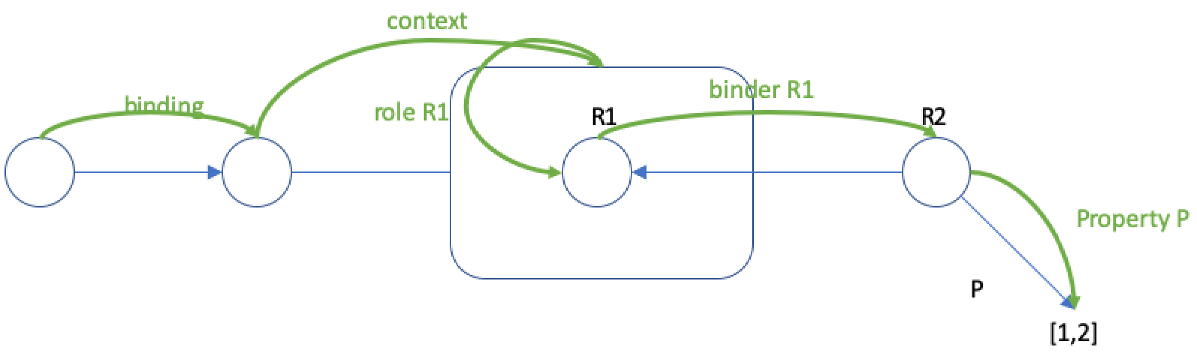

Figure 1. The thin blue lines are part of the structure; the green lines represent query steps.

Figure 1 also shows the jumps a query makes through the graph. In fact, it shows all possible types of jumps available:

- From a role to its binding (`binding`) and back (`binder R1`);
- From a role to its context (`context`) and back (`role R1`);
- From a role to a set of property values (`property P`).

Client requests

A client sends requests to the PDR. The core keeps the client up to date with changes to the underlying network by a mechanism of callbacks. In short: the client sends a callback along with its request and the PDR sends (new) results by calling it with those results.

In essence, the API of the PDR allows the client to request any of the five types of query step. Each step must be complete with arguments: for example, the client should say which role instance it wants to see the binding of; similarly, it should additionally say what type of role binders it is interested in (i.e. the type of the role instances that bind a given instance).

It follows that the client always requests nodes that are direct neighbours of another node. This means that the order in which a client requests nodes, depends on the underlying structure, as the client has no means of knowing the identity of a node[1] before it has received that node from the PDR.

Consequence of request order

Consider the situation in which a client (typically, a React screen) has requested a context, some roles in it and some of their properties. Now that context is deleted, let's say by another user. What requests does the PDR recalculate?

Let's take the React screen as an example. It is built, tree-like, from nested data containers that hold, as terminal leaves, simple display components. A role data container will be nested inside a context data container. Now if the context container disappears (is *unmounted)*, the client is no longer interested in updates for requests that originate from the data containers inside it. This is because it will scrap the entire context data container with all that is inside. As a matter of fact, it is even an error to try to update the data in unmounted components.

How does the PDR prevent updates to updated components? How does it ensure that other requests that are affected by this change, are updated?

How to visit a context

There are but three ways to enter a context, or, in other words, queries can alight in just three ways on the nodes that make up a context in the graph from outside that context.

[1] With the exception of *Named Nodes*. These are nodes that have a universal readable name besides their GUID-based identifiers. A modeler may use a Node Name (mostly context instance names) in a query; a screen programmer can start a data container with such a node.

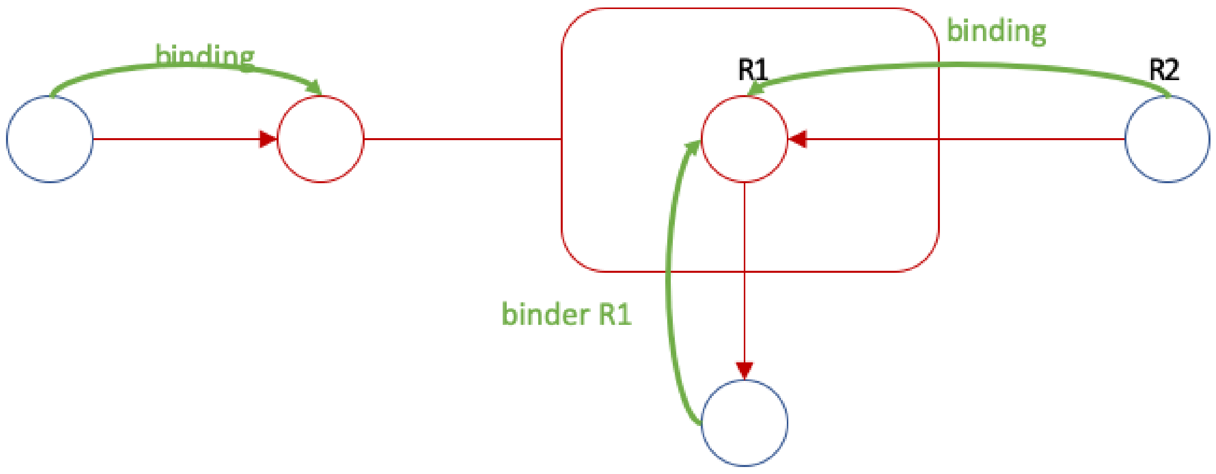

Figure 2. All ways to visit a context. Red parts will disappear when the context is removed.

As the figure shows, first contact with a context is always through a role instance; and this is either through the `binding` query step or the `binder <RoleType>` querystep.

Deleting a context

Figure 2 illustrates clearly what client requests should be recomputed if a context disappears. These are requests (green lines) from roles that will remain (in blue) to roles that will disappear (in red).

All other requests involving parts of the context that will disappear, must be ignored. Without exception they are requests that originate in parts that will have gone after the context has been removed; moreover, they will be `context`, `role <RoleType>` or `property <PropertyType>` requests.

By updating the former types of requests and ignoring the latter types, the PDR addresses both questions. All requests that need be updated are, in fact, updated; and no requests are updated that the client is no longer interested in.

Queries

A screen programmer can only make a client proceed step by step through the network, but a modeler may create a query that performs any number of steps with a single request. Such queries are stored as Calculated Roles or Calculated Properties. Incidentally, these calculated roles can be requested by a client, just like other roles and properties.

If we remove a context, how do we detect (in order to recompute) queries that have passed through that context? In general, how do we detect the queries that should be recomputed after any of the atomic changes that can be made to the network?

Queries trace their steps

The query execution engine records each step of a query. It saves the identification of the client request with the (function) type of each step, permitting the PDR to re-compute the request if the node visited by one of these steps is affected by a change to the underlying network.

Let's look at the details of saving a query step. A step consists of the identification of a node, and the name of the step. This works out differently for the various steps.

Step type	Node identification	Step name
`binding`	role identifier	"model:System$Role$Binding"
`binder`	role identifier	Role Type
`context`	role identifier	"Model:System$Role$Context"[2]
`role`	context identifier	Role Type
`property`	role identifier	Property Type

The `binding` and `context` steps use a generic step name; for the other steps, we use role or property type names.

Now let's look at what happens when changes are made to the network.

Adding a role instance

What queries can proceed once we add a role instance to a context? Figure 3 clearly illustrates there is only one kind of step that is enabled by adding a role instance: `role R1`. So if query that would have passed through instances of R1, should be recomputed. Because we just add a role (and no binding!) there are no other queries to consider.

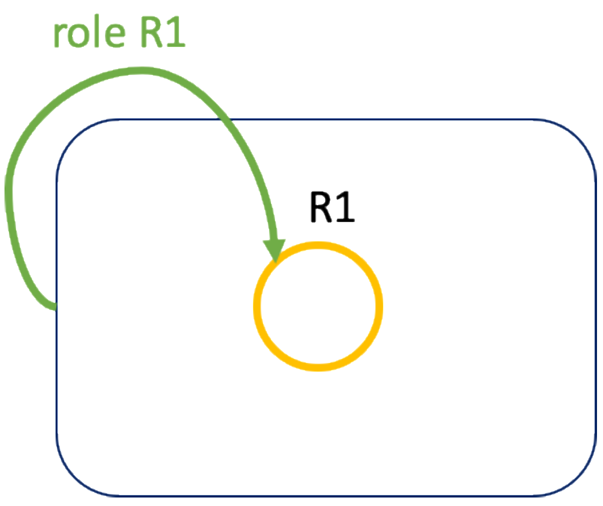

Figure 3. The yellow circle represents a new role instance added to the context instance.

[2] Actually, we do not record this step. See *Deleting a role instance* for an explanation.

Deleting a role instance

Much like when we remove a context instance, we have to consider the case of removing a role instance. How does the PDR ensure that requests that are affected by this change, are updated?

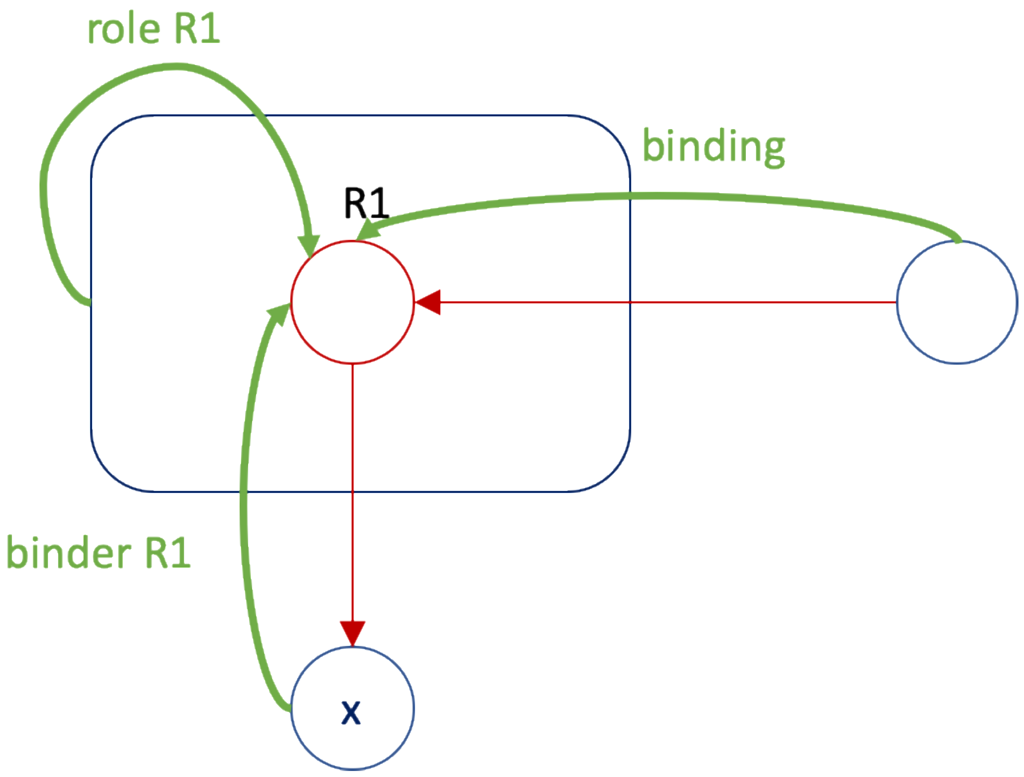

Figure 4. All ways to visit a role. Red parts will disappear when the role instance is removed.

From Figure 4 we learn that there are three ways for a query to alight on a role instance node. When that node is removed, all queries with such steps (the green lines) should be recomputed.

Shouldn't we recompute queries that have one of the inverses of these steps? Just think how such a query would arrive at R1 in the first place: it can only be by one of steps illustrated by green lines. Hence, if we find all queries with the green steps, we've got them all.

Changing the role binding

What if we just remove the binding of role instance R1? A glance at Figure 5 tells us that all queries with the step `binder x <R1>` should be recomputed, and the queries that contain the step `binding R1`. This is because we've severed a connection between nodes and those are the two kinds of step that traverse this type of link.

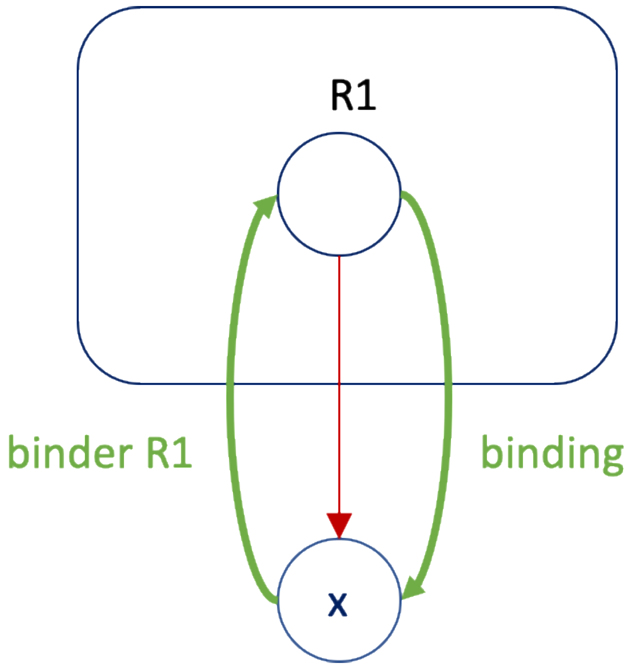

Figure 5. Remove the binding of node R1.

If we add a link, we do the same. Now one could wonder: can there be queries that have recorded this step, as they could not traverse the link before? Yes, there could be, because we record such steps, whether they succeed or not. So a query that tries to move from x to an instance of R1 by attempting the step `binder x R1`, records that step - even if it fails. Similarly, a step from R1 by following its binding records the step `binding R1`, even when there is no binding at all.

Changing property values

Any change to the values of a property P should lead to recomputing all queries with step `property P`.

Adding a context

We will consider the case of adding an empty context. To add a context with a role is like first adding an empty context, then add a role.

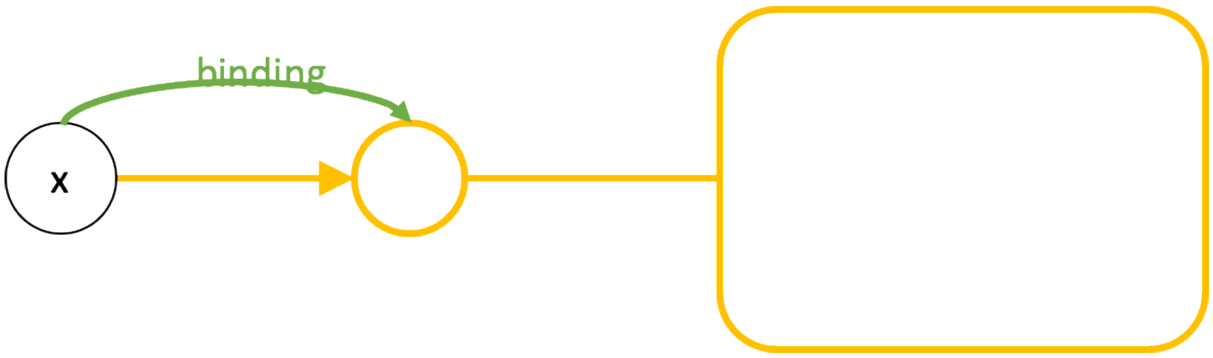

Figure 6. Adding an empty context.

Actually, adding a context is a simple case. It is very much like adding a binding because the only thing we can do – indeed, should do – with a new, detached empty context is to bind some role to its external role. Because we bind to a new role (the external role is created with the context) and because the external role can have no binding itself, we just have to consider queries with the step type `binding <ExternalRoleType>`.

The external function

We've left the `external` function out of our considerations. This function moves query evaluation from a context to its external role. Surely, this is a step very similar to the `role <RoleType>` step? But a moment's reflection tells us that there is no mutation of the underlying network that would add or remove an external role. Such roles come with contexts and leave with them, too. So: yes, we could trace the step, but it would never figure in computing which queries to recompute.

This is especially relevant for contexts that are not bound. Such contexts are allowed, as long as there is a Calculated Role that retrieves them directly from the database. When we remove such a context (by deleting it 'from' the Calculated Role), we make no effort to trace queries that depend on its external role (there can be none that come from outside of the context).