Sync subnetworks as a consequence of a new binding

Joop Ringelberg 28-05-20 Version: 1

Introduction

In the text *Perspectives on Bindings* we've concerned ourselves mainly with changing values of Properties of a Role. Another relevant change is that some user adds a binding to a Role. While we have (in version 0.4.0) a mechanism in place that informs peers of such an isolated change, we yet lack a mechanism to provide them with the necessary deltas to ensure they have access to the role graph of which that binding is the root. This text describes that mechanism.

Before we do so, it is important to notice the large differences between binding a role and the other modifications, to build some intuition for the magnitude of this operation.

Binding: the way to build large structures

Adding roles and properties will build up isolated contexts. However large these may be, their structure is simple. Only by creating bindings between roles can we build more complicated structures. In general, the data structures that we can create are *graphs* of contexts and roles.

User roles have perspectives. For the users, they make roles in and around their context *visible*. The modeller may define quite long paths through the graph, that give a user playing a specific role the right to view roles in far-off contexts. These paths are defined over *types*.

When in runtime a graph consisting of *instances* is built, more and more roles become visible to the end users. Now imagine two as yet unconnected subgraphs of instances and a user role U in one of them[1]. The other subgraph has been created by other uses and is not yet available to U. Let's assume that for users of that type, a path is defined that runs through both subgraphs. The moment that the right binding connects both subgraphs, user U should suddenly be able to 'see' into the other subgraph.

This must translate into a whole set of deltas to be shipped to U, so that his PDR can construct that formerly invisible subgraph.

[1] One may wonder: what is the graph that they are subgraphs of? In our example we assume that part of it is available to one user, while another part is available to others. It helps to image a very large graph – we call it the Perspectives Universe – that contains all data kept by all users of Perspectives. There is no single place (no computer) where this graph exists in its entirety. Each user sees part of it: what he sees, falls within his *horizon*.

This algorithm is suboptimal

We send deltas for the entire subgraph that should be visible to peer P. However, P might already have access to parts of that graph – indeed, maybe to the entire graph. So the set of deltas we prepare may contain too much information.

On receiving the Transaction, P will have to handle this: detecting a `UniverseRoleDelta` for a Role instance he already knows, he will simply not create a new instance. The other deltas are treated similarly.

While this is not optimal, there is no quick check to find out whether P already has access to parts of the graph. We might run the inverted queries stored with the type of a node in the graph to find all peers having access to it. If P is one of them, we don't have to send a new `UniverseRoleDelta`. However, this might be a very expensive computation. Indeed, when we bind a user role to a graph that bottoms out at the 'own' user, that query would return all peers with access to the own users' properties – potentially many hundreds or even thousands of them!

General approach

Consider, again, the simplest case of a query that computes a role set: a series of steps (a path). It will be applied to a context instance. Its first step will take it to a role instance. At that point, we can invert that first step to bring us back to the context of origin; we can also run the rest of the query and it will bring us the results *as computed from that role instance*. Notice, that the first step may have landed us on *multiple* instances of a Role!

We can repeat that for each successive step in the query: invert the steps we've taken so far, to take us back to the context of origin; run the rest of the query to get (part of) the result set.

Notice, that the steps back always form a path. That is true, even, if the original query is a tree! No matter how high up in its branches we are, the way back is always a path.

To return to the problem at hand, suppose that some user U has two subnetworks within his horizon. Another user, P, has only one of these within his horizon; the other is hidden. P has a perspective on a calculated role that will fetch him roles from the hidden part, but its query execution halts at a particular node that lacks a binding. That binding would, as it were, bridge the two networks for P.

Now U constructs that binding. Surely, U should now send him that formerly hidden network.

We can envision the two parts of the query at the node R that received the new binding. One part is the inversion that leads us back to the context of origin of the query. Here is P, who has a perspective on that particular calculated role. The other part is the rest of the original query. This is how we find out what to send to P:

1. Apply the inverse query to R, to find the context of origin and with that the user P2.
2. Apply the rest of the query to R, to find all contexts and roles that it visits. These we must send to P.

Intuitively, we have 'kinked' the query at R. The left part we invert: it takes us backwards to the context of origin. The right part we keep as it is: it takes us forwards to the query results.

So here is the outline of our approach:

1. In compile time, we do not merely invert queries at each step; we *kink them*, keeping both the inverted backwards part and the remaining forwards part. These we store with the role types (just as we have described for inverted queries in *Query Inversion over Model Boundaries* and in *Perspectives on Bindings*).
2. In run time, we do some extra work for RoleDeltas, to find the extra nodes we should send to some users. This involves running the remaining forwards part of the kinked query.

Finding nodes to be sent from query assumptions

When we run a query, we obtain a role set as result (or a set of values in case of a Property query). However, the query may have visited many intermediary roles that should be sent to P as well. To find them, we can re-use the existing *dependency tracking mechanism* we deploy to keep up to date the results of queries sent in by the client through the API (a client program requests a query to be executed and expects to be notified when the results change due to changes to the underlying network of context- and role instances).

The dependency tracking works by accumulating *assumptions*. Each query step adds an assumption.

Step	**Assumption constructor**	**Assumption parts**	**Query triggered by:**
role R	`RoleAssumption`	the context instance + R	Any change to instances of R
external	`External`	the context instance	Never (external role is fixed)
binding	`Binding`	role instance	Changes to the binding of the role instance
binder R	`Binder`	the role instance + R	Changes to the binders R of the role instance
property P	`Property`	the role instance + P	Changes to the values for P of the role instance
context	`Context`	the role instance	When role instances are moved to another context.

2 We store, with an inverted query, the user role types that have a perspective on the query whose inversion it is.

me	`Me`	the context instance + maybe the role instance	When a user role is added to the context that is ultimately bound by the user of the system.

From these assumptions, we can derive the role and context instances that should be sent to P.

For a `RoleAssumption`, we create a `UniverseContextDelta`, a `UniverseRoleDelta` and a `ContextDelta`. These instruct P to create an empty context, an empty role and to connect the two.

We do the same for the `Me`, `Binding`, `Binder`, `Property` and `Context` instance.

For the `External` assumption we do nothing, because a query with a kink by construction never starts with the `external` step. So whenever `external` is applied, a `context` step will have been applied before (as it is the only way to get to a context). That step already adds all deltas that we could wish to add for an `external` step.

This algorithm is suboptimal

We currently (version v0.5.0) handle each assumption in isolation from the rest. That causes a lot of deltas to be generated twice. For example, a query where a `context` step is followed by a `role` step will create the `ContextDelta` twice. Only one will be added to the transaction, but we could avoid the double work if we take the *order* of assumptions into account and keep an eye on the history of assumptions we've handled. This is for future optimisation.

Property set

User P will have a perspective on the Calculated Role that allows him to see the value of some Properties. We create Deltas for these Properties by obtaining, for each such Property, its value from each of the role instances in the set that results from applying the forwards part of the query. Remember that we run that forwards part from the node that U just added a binding to. Getting a Property's value from a role instance generates assumptions, too, so this ties in nicely with the general approach outlined above.

Now, what if the original query actually *ends* at the role that U added a binding to? We then obtain the property values from that node itself. As a justification: the remaining forwards part of the query is empty: we could construe that as the identity function, that, applied to the role instance, yields itself, so it forms by itself the result set of the remainder of the query.