# Overview of state change mechanisms

Joop Ringelberg 15-02-20 Version: 2

## Introduction

Perspectives has straightforward declarative semantics, explained elsewhere[1]. Program use proceeds by changing that declarative state. In other words: the set of facts kept by Perspective users changes. In this text we give a high-level overview of the techniques involved in dealing with that state change, in order to preserve the declarative semantics in each new state. Where appropriate, we give pointers to other texts that have been written on the subject.

Where do changes come from? Ultimately, they come from the users of Perspectives. Keep in mind that there is a *Universe* of facts collectively tended to by many users. Each user 'sees' facts within his (or her) own horizon. Horizons overlap but are never equal.

When thinking about the origin of changes that arrive in the PDR, we must distinguish between those made by the owning user and those made by his peers. This is because it (the PDR) must send the owning users' changes to his peers – but just absorb and integrate send changes by those peers.

Finally, we must remember users can have bots that act on their behalf.

So when we think about the implementation of the PDR, we have to reckon with three sources of changes:

1. The API that enables a client program to translate the owning users' actions into updates;
2. The rules of the owning users' bots, that make changes on his behalf;
3. Incoming transactions with changes made by peers of the owning user.

The first two translate into the application of a limited set of *update operations*. The third translates into incoming *Deltas*, where each Delta describes an atomic change.

Where do Deltas come from? They are created by the same update operations.

## Five responsibilities

To preserve the declarative semantics, the PDR must recompute requests by client programs sent in through the API. Clients can request neighbouring nodes in the network of context- and role instances, as detailed in *Implementing the Functional Reactive Pattern*. A client should be informed of a change to those parts of the network it has requested (clients are said to have *subscribed* to parts of the network). Now it is important to realise that some of the relations between nodes may be *calculated*. We call

[1] See *Semantics of the Perspective Language*.

a role calculated if it is retrieved by a *query expression* (see *Perspectives Across Context Boundaries*). So it may happen that part of the network traversed by the query engine to compute a role may have changed, even if the nodes that ended up in the query result have not. In such cases, the query must be re-run in order to be able to inform the client about the new declarative state: the *selection* of end nodes is likely to have changed. This is the first responsibility of the PDR.

Secondly, a change might imply that a bot rule must be triggered. Rule conditions are Boolean queries. This is similar to the situation with the client requests. However, rules do not make an explicit request. Models may contain many rules; in theory we need to evaluate them all after each change. The challenge is to create an efficient mechanism that prevents this, yet fires all rules whose conditions have changed.

Third, the PDR must find out where to ship the Deltas that arise from the update functions. Whom should we send a particular Delta? A first approximation is: to anyone who has a perspective on the changed node, as described by the Delta. But, again, as with the client request subscriptions and the rule triggering, queries complicate the matter. A change may happen in some context that user A has no role in, but that is passed through by a query for a calculated role that A has a perspective on.

Fourth, obviously, the PDR should hold on to changes. They should persist from session to session. Persistence relies on Couchdb and on a cache in memory.

Fifth and finally, the PDR keeps a record for each context, telling it what role instance represents the owning user. Many computations depend on that information, so it is prudent to keep a permanent record. This, however, is merely a matter of efficiency; we could do without. The bookkeeping is simple: for a context we record which role instance represents the owning user (dubbed `me`), and on that role instance we have a Boolean property `isMe`.

To sum up and give compact names to these five responsibilities, we have:

- request updates
- rule triggering
- synchronisation
- persistence
- current user computation.

# Mechanisms

We use various mechanisms to shoulder the five responsibilities.

## Request updates: dependency tracking

A query is in essence a series of steps through the network of context- an role instances[2]. A step is one of the five fundamental moves through the underlying network:

- from role instance to its context;
- from a context to the instances of a particular role type;
- from a role to its binding
- from a role to roles that bind it
- from a role to values for a particular property.

Each step is carried out by a simple function. It records its own application. A client request through the API also leads to application of one of these functions. In other words, we can equate each client request, whether it is just a request for the binding of a node or the value of a complex calculated role, with a query[3]. The recorded steps are associated with a particular client request.

A Delta[4] also corresponds to a step. So when we have a Delta, we can look up what client requests are affected by that Delta and recompute them.

## Rule triggering: inverting queries

Even though a rule condition is just a query (with a Boolean result), we cannot reuse the dependency tracking mechanism. This would require us to evaluate all rule conditions at least once. The computational costs may be considerable, especially when we realise that most context instances are not loaded into memory, for any given session. In order to evaluate each rule, we should, indeed, load everything in memory and that is something we have taken great pains to avoid in the Perspectives implementation.

So another mechanism is necessary and we have found it in running the rule conditions in reverse. This is explained in detail in the text *Query Inversion*. Briefly, it consists of inverting conditions so they run from each node that would be visited, to the context that holds the rule. These inverted queries are stored with role- and property types. When a Delta comes into effect, we look up the inverted queries for the resource in question and run them to find *affected contexts*. Then we run the rules for the owning user in those context instances[5].

[2] See *Implementing the Functional Reactive Pattern* and *State and Dependency Tracking* for more detail.
[3] To prevent misunderstandings: the client can also order changes to the network through the API; however, in this context we will not call them ‘requests’.
[4] We use Delta and ‘change’ as synonyms in this text.
[5] Inverted queries are stored with the user type(s) that they are relevant for. In the case of a rule, this means the user role that the bot is for. Finding the owning user is just a matter of lookup, anyhow.

This mechanism will run rules even for contexts that do not reside in memory as the Delta takes effect.

## Synchronisation: inverting queries, too

Inverted queries give us a solution for computing users that should receive a Delta, too. To implement it we do not just invert rule conditions, but, indeed, each query defined in the model. Remember that they define either (Calculated) roles or (Calculated) properties[6].

Requesting a Calculated role is trying to 'move' from a context to the instances of that role. Now, because the role is calculated, retrieving the instances will in general involve a number of moves, possibly outside the context of origin. When we invert those steps, where do we end up? In the context of origin, of course.

We invert the calculation for a CalculatedRole in design time (on processing a model). Doing so, we store the user types that have that role in their perspective, with the inversions. So when we later (in run time) run the inverted query and end up with some context instances, we can immediately look up the instances of those user types. They should receive the Delta.

## Persistence

For persistence we have a number of functions that cache in memory and store in Couchdb. This task is straightforward.

## Current user computation

The User role of model:System represents the owning (current) user[7]. Consequently, any role instances filled by this role represent the current user, too. This definition can be construed recursively.

For now, we hold that a context instance can have only one role instance that is its current user. In other words, a user should take only one role in any context. This may change in the future, as we extend the language.

[6] Or the condition of a rule, or the value expression in a let body or its bindings.
[7] Notice that these are indexed concepts! All users are the owning user with respect to some PDR installation. While doing its work, that user is the 'current' user for that PDR.