Perspectives across context boundaries

Joop Ringelberg 06-11-19 Version 1

The importance of context

Context introduces privacy in the sense of providing a comprehensive view of participants. As a first approximation, we can provide this with the requirement that a user role can just have perspectives on other roles in his context. However, this requirement is too strict. Context roles and external roles of contexts broaden the horizon of user roles in a limited way, but it is not enough.

By incorporating calculated roles in the language, we provide access to a wider environment around the users' context. All contexts that can be reached through some role path can be consulted. This raises the issue of privacy, however. At the very least, a user participating in some context should be informed about who else can consult (parts of) that context.

Perspectives across borders

Let's examine `Party`, as modelled below. `Guest` has a perspective on a calculated role:

```
case: Party
  user: Guest
    perspective on: Giver
  user: Giver = WishInParty >> binding >> context >> Giver
  context: WishInParty filledBy: Wish
  case: Wish
    user: Giver
```

We could write this more compact by losing the calculated `Giver` role in Party:

```
case: Party
  user: Guest
    perspective on: WishInParty >> binding >> context >> Giver
  context: WishInParty filledBy: Wish
  case: Wish
    user: Giver
```

Notice how we moved the calculation of `Giver` to the perspective declaration. In a diagram, we just draw a line across context boundaries. There is, however, a condition on such lines: there must be a valid path from the source context (`Party`) to the destination

role (`Wish`). In our example the path exists because we have bound `Wish` in the context role `WishInParty`.

Such a line would be a command to the system to compute the path. We assume the system would notify the modeller if no path exists. In future versions of the system we could allow the textual modeller to command the system to find a path, too, for example with this notation:

```
case: Party
  user: Guest
    perspective on: ... Giver
  context: WishInParty filledBy: Wish
  case: Wish
    user: Giver
```

Computed user roles

There is another way to interpret this example. We might say that there is, in the case type Wish, a *calculated user role*. Let's call it GuestInWish:

```
case: Party
  user: Guest
  context: WishInParty filledBy: Wish
  case: Wish
    user: Giver filledBy: Guest
    user: GuestInWish = External >> binder WishInParty >> context >>
Guest
      perspective on: Giver
```

The computation of `GuestInWish` proceeds in four steps: from `Wish` we move to its external role, then to its binding role instance of type `WishInParty`, then to its context and finally to the instances of the `Guest` role.

Notice that we've given this calculated user role a perspective on `Giver`.

Transparency is restored, with this model. At a glance we see who can consult the roles of Wish. At a glance, because we only have to examine the definition of Wish itself. We don't have to scan other context definitions for calculations that provide access to Wish's roles.

Equivalence

The models given above are completely equivalent. There is no difference, in effect, between a perspective on a calculated role, or a perspective on an enumerated role by a calculated user role. The system should be able to transform one into the other.

A difference might arise when we interpret the perspectives to screens. On first sight, one might think that in the original model, the Guest would consult Wish in the screen that is

created for him for Party. In contrast, with the second model, he would be able to open a screen for Wish itself. Whether such decisions on screens should be decided by a choosing one of the two otherwise equivalent models, is an open question. Modelling process logic might dictate a course of action that runs contrary to user experience design.

Transparency restored

One way or another, the equivalence between both ways of modelling allows us to create a system that can, in principle, show a participant in a context who else can consult that context, however it is modelled.

We can imagine a query function that computes all user roles in a context – including the calculated users[1].

Assignment

Up till now we've written in terms of *consulting* perspectives on roles outside the context. Do the conclusions extend to *changing* those roles (and their property values)? We think so.

A primary example is the common case where a user role *becomes* another role. In the `Party` example, `Guest` can fill `Giver`. That is, he can bind the role instance that represents himself in `Party`, to a new instance of Giver in `Wish`. `Guest` becomes `Giver`.

There is – in this case – a restriction on this perspective, however, and that is that a `Guest` can only bind *himself* in `Giver`. Otherwise, we would have the situation where any `Guest` can make another `Guest` give a present! But, again, this is particular to the *become* Verb.

Becoming is almost the reason for perspectives on calculated roles (or for calculated users): without them, a model like Party-Wish would become very clumsy, as Guest would need a role in Wish before being able to fill the Wish role. An infinite regression threatens here, only to be broken by *another* user role. Becoming is an elegant solution for this problem.

We can extend these powers of change across context borders to other perspectives. For example in the bot rule below:

```
External >> binder Role1 >> context >> Role2 >> Prop1 = true
```

Here, the bot reaches outside its context to set a property on a role on its enclosing context. The left hand side of the assignment is a path; the property value on the end of that path is set to `true`.

[1] This function is necessary for synchronization, too.

In general, any perspective with `Change` powers on a calculated role allows the user having that perspective to modify that calculated role. These are great powers, indeed! With them come responsibilities for the modeller.