Synchronisation Edge Cases

Joop Ringelberg 22-11-21 Version: 2

Introduction

Consider a context with several thing roles and a couple of user roles. Now assume those user roles have exactly the same perspectives. Anytime one of the peers makes a change, the same transactions[1] are sent by his PDR to all his peers. This is the base case for synchronisation.

But user roles are defined by their perspectives, so, really, our context has but a single user role[2]. This is a degenerate case indeed: a context with a single user type could be served from a single store, as all participants have access to exactly the same information.

Obviously, the general case is more complicated. It turns out that there are two important ways in which contexts differ when it comes to synchronisation:

1. By having user roles have different perspectives on the non-user roles;
2. By having user roles have different perspectives on other user roles.

Different perspectives on non-user roles

When a given non-user role R falls into the perspective of some user roles but not that of others, a change to that role should be distributed to just the first group of users. This is the base case of differential synchronisation. We have the PDR send changes just to those users with a perspective on the changed object.

An aside: trusting transactions

A PDR receives transactions from peers. These transactions reach it through a messaging service[3]. Anticipating on malicious agents, we ask ourselves the (rethorical) question: should we trust each incoming transaction? The answer is no, as it is quite possible that the address of an inbox will be stolen. Consequently, a *denial of service* attack could be mounted on that PDR. We can mitigate the gravity of such an attack by abiding by the simple rule that we accept transactions *from peers only*.

This may seem to implicate that perspectives on peers should always symmetrical (run both ways). This is not so. A *peer* is not the same as another user role in the same

[1] PDR's exchange *transactions* that consists of *deltas*, the elementary changes to structural elements such as contexts and roles.
[2] This may not be obvious at first sight, as roles might have different fillers or properties. Perspectives may be valid in some states only, where state can be defined in terms of fillers, but we explicitly stated that all perspectives were equal. Roles might have different properties, but because all user roles have the same perspectives, these properties evidently play no role in them.
[3] In essence, each user has a *message box* on some server that others may send transactions to.

context. Peers are instances of the User role in the PerspectivesSystem context. Consequently, perspectives need not be symmetric for PDR's to accept transactions coming from the peers behind them.

Perspectives on users: the *user graph*

In the most symmetrical case, every user type has a perspective on every other user type. This makes synchronisation easy. All we have to do for a given change is to calculate which peer types have a perspective on the structural element that has been changed[4], and then send the change to their instances.

Hidden peers

But we can think of many realistic cases where peers do not all 'see' each other. For example:

- In a scientific peer review, the reviewers are hidden to the authors;
- In an examination, a second examinator may be hidden to the student;
- A controller may have access to financial facets of sales contexts, without clients needing to see that controller;
- A shop has many clients but these generally are not aware of each other.

In each of these situations, disclosing the identity of the hidden roles to the other peers may be considered a breach of privacy. Of course each participant can be aware that *others may be involved in the context*, by reflecting on the model. But they can never recover the *identity* of those others.

The User Graph

Consider a graph where the user roles form the nodes and the perspectives on user roles the edges. Incomplete graphs will reveal challenges to synchronisation, as we will see below. In fact, a first and immediate problem will be recognised when the graph consists of two unconnected subgraphs: in such a case, there are really two contexts, with peers that will never know of each other's existence.

Why is that a problem? Because the PDR of a user is only able to communicate deltas to peers it knows about. So if a user in one subgraph makes a change, none of the users in the other subgraph will ever know about it – even if they have perspectives that would allow them to perceive the changed element. See Figure 1 for a simple example. We call this situation *incomplete synchronisation*.

[4] This may be a context (to which roles may have been added), a role (that may have gotten a filler) or a role one of whose property values has been changed. Finally, we can have the situation that a new context instance has been created.

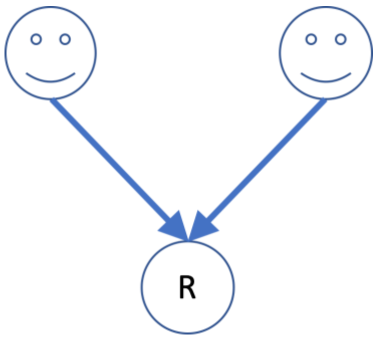

Figure 1. The users are unconnected by perspectives, hence they form two separated subgraphs of the user graph. They both have a perspective on non-use role R. But a change to R made by either user will never be communicated to the other user.

The user graph shows us how users connect, gives us paths between user roles. We will use those paths to spread changes beyond the immediate circle of peers a given user can see.

A synchronisation principle: *passing on*

Consider the user graph (with a non-user role R) in Figure 2. When the left user[5] modifies R, he[6] can send the change to the middle user, but not to the right user. But the right user has a perspective on R, too, so should receive the deltas. As we do not wish to add a perspective from left to right, we need another principle in synchronisation. This we call *passing on*. The middle user should pass the deltas concerning R, on to the right user.

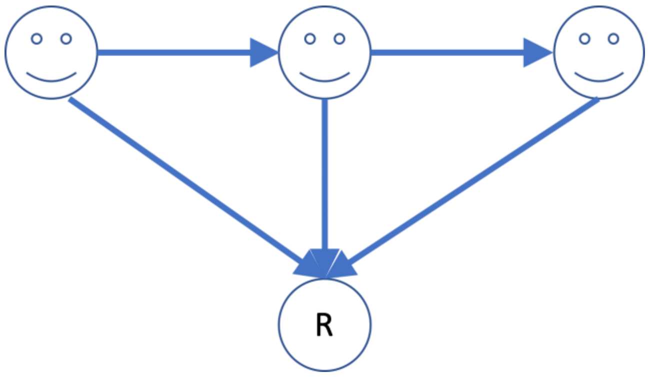

Figure 2. The user graph (top three nodes) is not fully connected. The left user is not directly connected to the right user (also, all arrows are unidirectional). Changes made to R by the left user should be passed on by the middle user to the right user.

Computing users to pass on to

The situation in Figure 2 is easy to comprehend. This is less so, however, in Figure 3. We can explore the graph by building a table with user roles on the rows and columns, and the paths between them in the cells (paths are expressed as user role nodes to pass on the way).

[5] Remark: a user graph is defined on the type level. However, in the examples we will occasionally treat the graph as if the nodes were instances rather than types.
[6] To avoid the tedious distinction between a user and his PDR, we will take the liberty of writing that a user sends a change, while meaning his PDR does so.

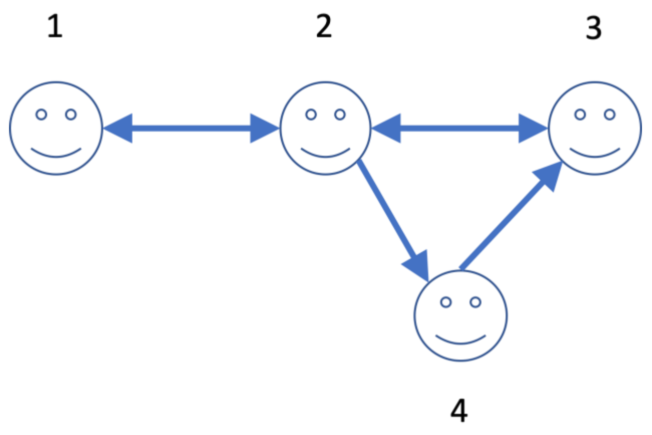

Figure 3 A more complicated user graph. We've omitted the non-user role R. However, all user roles have a perspective on it.

	1	2	3	4
1			2	2
2				
3	2			2
4	3, 2	3		

Table 1 The paths in Figure 3 (expressed in terms of user role type nodes to pass through) from user role to user role.

Table 1 shows us that we have to pass through node 2 in order to reach node 3 from 1. This can easily be verified in the graph. Less obvious is the path from 4 to 1: it is through 3 and 2.

Here is how we use the paths table to create deltas that are passed on:

- We start with the user role authoring the change. This we use to index a row in the table.
- We then determine the destination of the delta (according to the perspectives of the peers). This we use to index a column in the table.
- We then add to the delta the user types we find in the cell we've located.

In many cases above, there are no intermediate user types. But if an instance of user type 1 adds an instance of a role R, we create a delta, add user types 3 and 4 to it and send it off to all instances of user type 2 that we know about.

An instance of user type 2, upon receiving that delta, notes that it contains user types (3 and 4). It looks up all instances of those types and sends the delta to them, after removing user types 3 and 4. Obviously, the instance of type 2 also modifies its local store by adding the new instance of R!

An instance of user type 3 or type 4 receives the delta and just modifies its local store.

This is a general principle: it will work in every conceivable situation. We will explore more examples below. It depends, obviously, on the user role graph and the paths table.

How to handle multiple paths

It may happen that there are more than one way to connect two user nodes. In fact, Figure 3 contains a few examples. We can reach node 3 directly from node 2, but also via node 4, for example.

This is an easy case, as the first path is shorter than the second. We prefer shorter paths.

But we may have situations where two paths of equal length exist. In such cases it seems tempting to choose one at random. However, we deal with the type level here. It may well be that on the instance level one path may exist while the other is absent. So we should use both paths.

Optimization

Let's reconsider the example we worked out above where an instance of user type 1 adds an instance of a role R that can be seen by roles 2 and 3 as well. What if there are *multiple* instances of user role type 2? If we send the delta to all instances, all will send it on to instances of user role type 3. This doubles the load on those instances while it achieves nothing extra. In fact, only one of them needs to do the task.

In general it is not easy to coordinate work among the peers in a distributed system, but here we are in luck. The instance of role 1 that created the change can act as coordinator by

- Creating an instance of the delta *with* user role type 3
- And creating an instance of the delta *without it*.

It then sends the first delta to *just one instance of user role 2,* and the second delta to all others. Thus, it burdens just a single peer with the task to pass the information on to instances of user role type 3.

It may send the *pass on* instruction to multiple instances of role 2 to increase the chance that it is sent on promptly (not all peers will be online, the more receive it the sooner it will arrive at its final destination).

Adding new peers

A special case of synchronization arises when a user adds a new peer to a context. In such cases, the entire context *as perceived according to the perspectives of the new peer* should be sent to that new peer (and, obviously, the fact that a peer has been added, should be sent to other peers).

An example

In Figure 4 we have a context instance C, a role instance 1 and a role instance R. Now 1, having a perspective on a user role type 2, adds an instance of 2 to the context[7]. Obviously, the instance of 2, being new in the context, has no prior information about it. So the burden falls upon 1 to send all deltas necessary to recreate C and R locally, to 2. In the figure we've shaded the structural parts that are to be sent to 2.

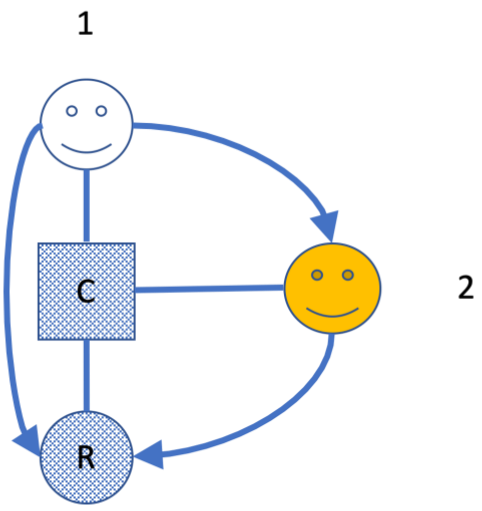

Figure 4. The yellow circle represents a peer added to the context C by user 1. Both peers have a perspective on role R.

This works by handling one by one all perspectives the new peer has on the context. Handling here means: run the queries that provide the objects of the perspectives while jotting down all context and role instances that are passed by the query evaluator. For example: for 2 to arrive at role instance R, the evaluator passes C. In a similar vein all property values are collected. For each of these structural elements, *the original deltas that created them* are sent to 2.

When adding a peer brings in even more peers

Consider a webshop in Perspectives. The visitor, deciding to browse the virtual shelves, creates a shopping basket (a context instance). It will contain himself, as (prospective) client, so he can see inside it, but otherwise it will be empty. He will then proceed to add articles to it in some way (presumably by dragging roles representing those articles over it).

Finally, he decides to really buy these articles. Only now will he reveal his identity to the shop, *by dragging a representation of a sales person onto the basket*.

This situation is covered by the example in the previous paragraph, where user role type 1 is the client, R represents the articles and 2 the sales person.

[7] Note how we jump opportunistically between the type- and instance level!

Let’s increase the complexity of the case by involving the webshop’s financial controller. This role must have access to all shopping baskets to retrieve the financial information from it. We assume the sales person and the controller are roles in the webshop context.

Now, the controller does not need to be an enumerated role in the basket context. We can easily compute this role once the sales person has been added to the basket: we just follow the path from the sales person role in the basket to its filler as employee in the shop, to the shop context itself and then to the controller role.

Obviously, the PDR of the client should disclose some information to the financial controller. But how? It has no access to the web shop context. See Figure 5.

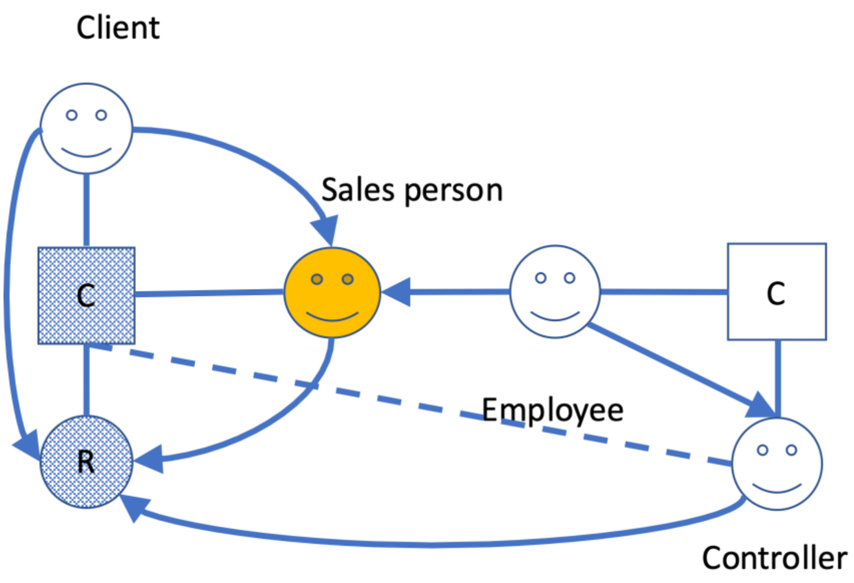

Figure 5. The webshop. The sales person is added to the basket (C) containing the articles (R). The Controller is a calculated role in the basket (represented by the dotted line). NOTE: this model is not final: see Figure 7.

As a matter of fact, this situation can be handled by the same mechanism of *passing on* we’ve defined above. In short: the Client is aware of the Controller role and includes it in the deltas it sends to the Sales Person. Thus, the Sales person (or rather the PDR for the peer that ultimately fills Employee) passes these deltas on to the Controller. The Employee has all information to compute the Controller.

But there is a catch we’ve glossed over. How does Client know he cannot address Controller directly, and how does he know he can reach Controller through Sales person?

This would happen if the user graph were like Figure 6. But Sales person does not have a perspective on Controller; Employee does.

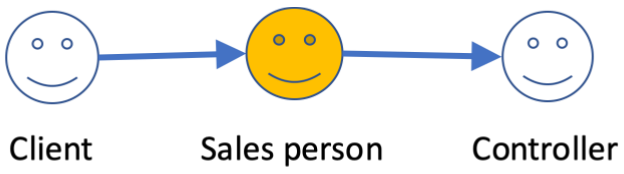

Figure 6. The user graph we would like to derive from Figure 5.

It takes some reasoning to derive Figure 6 from Figure 5. We start with the observation that Controller is a calculated role in the basket (C). We then should note that the user graph represents *connections between PDR installations*. This is because there is a one-to-one correspondence between User roles of PerspectivesSystem and a PDR – intentionally, PerspectivesSystem *represents the PDR*. So because Sales person is filled by Employee (and Employee is filled, in the end, by User) we may include the perspective of Employee on Controller in our user graph.

We will not implement this reasoning. Instead, we restrict ourselves to perspectives on roles in the context. Hence, we require a perspective of Sales person on Controller (as a calculated role of Basket). See Figure 7.

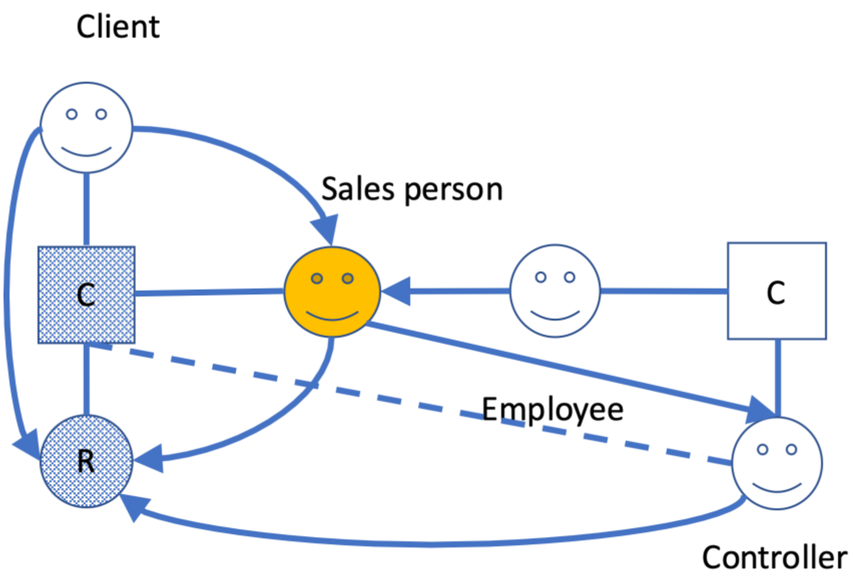

Figure 7. An improved version of Figure 5. The perspective of Sales person to Controller has been added (and we've omitted the perspective of Employee on Controller for cosmetic reasons; there may be reasons to have that perspective that fall outside this example).

It is straightforward to derive the user graph in Figure 6 from this model. Should the model lack this perspective, the system should warn that synchronization will be incomplete.

When multiple peers must contribute

What if the new peer has a perspective on something the user who adds him cannot see? Think about a soccer club with multiple youth teams. A parent enlists her daughter and the clubs' administrator assigns her to a team. The teams' trainer then provides her with the training schedule etc. We can model this as shown in Figure 8.

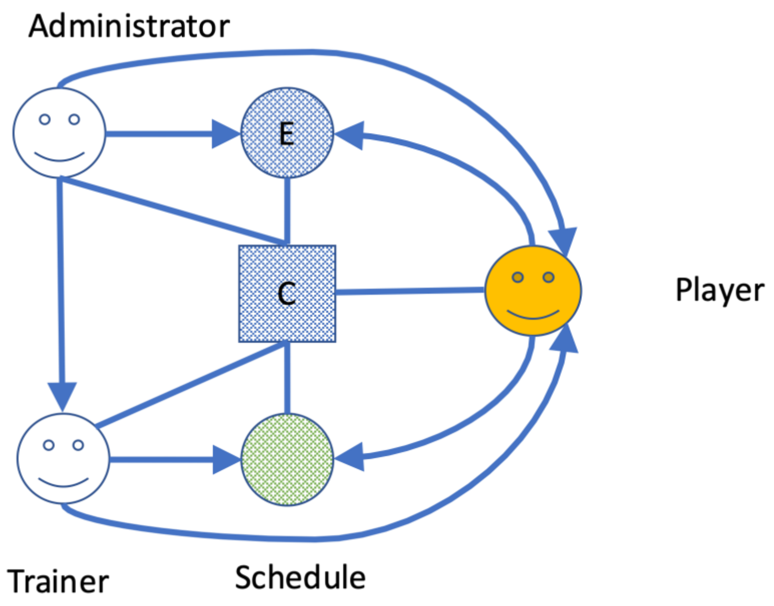

Figure 8. A soccer team (C) with external role (E) and a Schedule role. The Administrator adds the Player (yellow) to the team and sends the blue items to her. The trainer receives the new player and sends the schedule (green) to her.

The situation is like in Figure 3, but with an extra user role (Trainer) and an extra thing role (Schedule). As the Trainer has a perspective on Player, Administrator sends deltas to Trainer so he can update his local database with the new Player.

However, we can see in the picture that the Administrator cannot send the Schedule to the new Player: he has no perspective on it.

Instead, his PDR makes use of the connection of Trainer to Player. But, unlike in Figure 2, he cannot send a delta along that path – because Administrator has no information on the Schedule instance. However, his PDR can *reason* about Schedule instances (using the model). So instead of sending a concrete delta with details, he sends a *type level delta* to the Trainer. This type level delta contains both the Schedule role type and the path to Player from Trainer. We call this type of delta an *InformNewPeerDelta*.

Trainer, upon receiving a InformNewPeerDelta, verifies if he can find instances of Schedule in his database. He can and therefore creates a delta on the instance level and sends it to Player.

Several caveats here:

1. How does Administrator find out he has to instruct Trainer to send Schedule to Player?
2. Trainer should send the Schedule *only to the new player!*

Determining who should inform the new peer

Administrator should reason like this:

1. What roles does Player have a perspective on?

2. Which of those roles do I myself not have a perspective on? Let's call them M_i.
3. What roles (other than Player) do have a perspective on M_i?
4. Do those roles have a perspective on Player (how do they connect to Player in the user graph)?

Obviously, as this reasoning is based purely on the model, the other roles could do it too. However, if we have the role that added the new peer, perform the reasoning, it can coordinate the work in the same way we've seen before. Trainer is likely a functional role, but suppose there were two Trainers, only one of them needs to inform the new Player.

Making sure only the *new* peer is informed

It requires the special InformNewPeerDelta to make other user role instances inform (just) a new peer. So in our example, only because Trainer receives a delta of this special kind, he informs just the new Player. The user path in such a delta will end in an *instance,* rather than a user role type.

Refinement: properties in a perspective

In the above, we've dealt with perspectives as if they only concern roles. However, usually a perspective pertains to properties as well. So instead of merely asking whether the new peer sees the same role, we should ask about properties, too.

Here it is useful to introduce the concept of the *difference between two perspectives*. Let's conceive of a perspective as the combination of a role type and a set of properties: <R, props>.

We define P1 minus P2 as follows:

- If the role object of P1 is not equal to that of P2, the difference is P1: P1 – P2 = P1.
- If the role objects are equal, the difference is the a perspective with the same role combined with the difference of the sets of properties in the perspectives: <R, props1> – <R, props2> = <R, props1-props2>.

Let's start by subtracting from a perspective of the new peer (let's call that P1). For each perspective P2 of the user that adds the new peer (the *initiator*), we subtract it from P1. If the difference is P1, it is of no use to us. If the (property set of the) difference is empty, we can inform the new peer completely from the PDR of the initiator. If the difference is a non-empty set of properties, we'll have to enlist the help of other peers.

Remember we can only make use of peers who are on a path between the initiator and the new peer!

For such a peer, we evaluate one by one his perspectives. From such a perspective we subtract the difference we obtained above. If the result is empty, we're done. Otherwise we'll have to take the remaining difference and try another peer, until we're done.

The above algorithm ends with either a set of peers, each combined with a partial perspective to send to the new peer, or with the conclusion that synchronization cannot be complete, it does not give the best result in the sense that we've enlisted the help of more peers than needed. This can be fixed easily but is computationally more intensive.

But there is another issue. Consider this situation: the initiator can send the object of a perspective and some properties, but not all. Another peer need only send the rest of the properties. How do we do that?

We'd need to make clear *exactly what the peer doesn't need to send*. That is not trivial. Properties may reside anywhere on the role telescope. The initiator may have sent part of that telescope, while the peer needs to send a further part, that bears the required properties (and those fillers require their contexts too!).

I see no easy way to communicate, to an enlisted peer, what he doesn't need to send. So as it stands, we accept that enlisting a peer to complete a perspective for a new peer involves overhead for both that peer and the new peer (who'll receive some role- and context information more than once[8]).

Refinement of the notion of InformNewPeerDelta

In the light of this discussion, we can exactly define the – up till now informal – notion of a InformNewPeerDelta. It is the combination of three things:

- A user path, in terms of user role types;
- A final destination, in terms of a user role instance;
- A simplified perspective, as the combination of an object[9] and a set of property types.

Reaching out to peers along a user path to make them inform a new peer

Finally, a really complicated case that combines aspects of what we've seen above. With a stretch we extend our soccer example by including a Trainer Assistant, who is in the know of the Schedule, while Trainer himself has no access to it (unlikely, but just for the sake of the example assume it is so). Moreover, Administrator has no direct perspective on Trainer Assistant.

This requires Administrator to send an InformNewPeerDelta along a user path to Trainer Assistant. The path is via Trainer. See Figure 9.

[8] This is not a problem in the sense that handling a delta is an idempotent operation; there is no difference to the state if it is added more than once.

[9] To get really technical: the object is given in the form of a QueryFunctionDescription (the internal representation of a query path to a role).

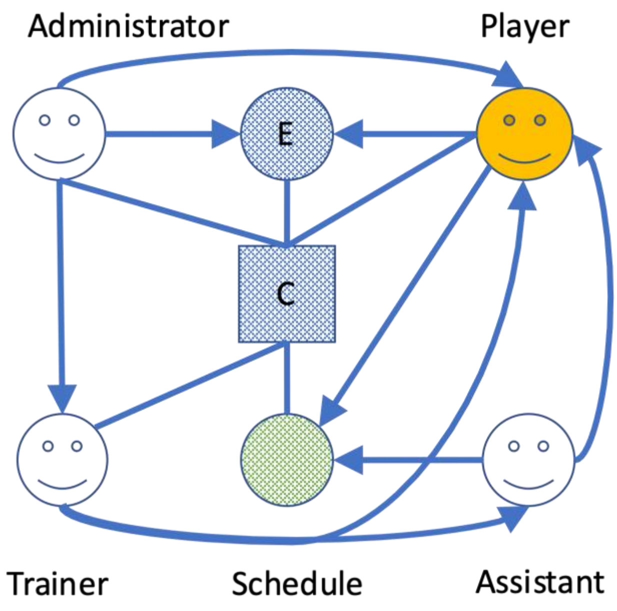

Figure 9. An extension of the situation given in Figure 8. The Trainer now has an Assistant. Trainer cannot see the Schedule, but Assistant can.

Administrator must instruct Assistant with an InformNewPeerDelta, so includes the path from himself to Assistant in it: (Trainer, Assistant) (see Figure 10). The new Player instance is included as the final destination. And the Assistant perspective on the Schedule is included, so Assistant knows what to send to Player. We assume the Assistant perspective on Schedule covers that of Player.

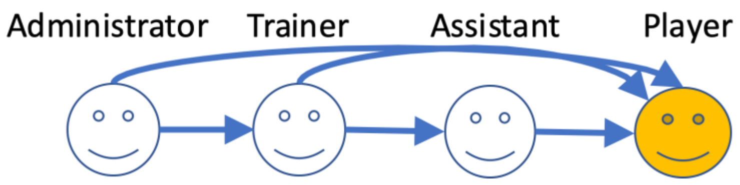

Figure 10. The user role graph for the model in Figure 9.

Implicit perspectives

In Figure 1 we presented two user roles with a perspective on the same role R, that did not see each other. We speculated that if one of those roles modified R, the other would never know. Thus, to synchronize properly, we must add a perspective from the modifying user to the other user. In Figure 11 we've reiterated that situation. We've drawn the context as well and we've indicated which of the users modifies the role R (by starting the perspective arrow with a closed circle). We've also drawn the *inverted query* that originates in R and leads to user role 2. This is the inversion of 2's perspective on R.

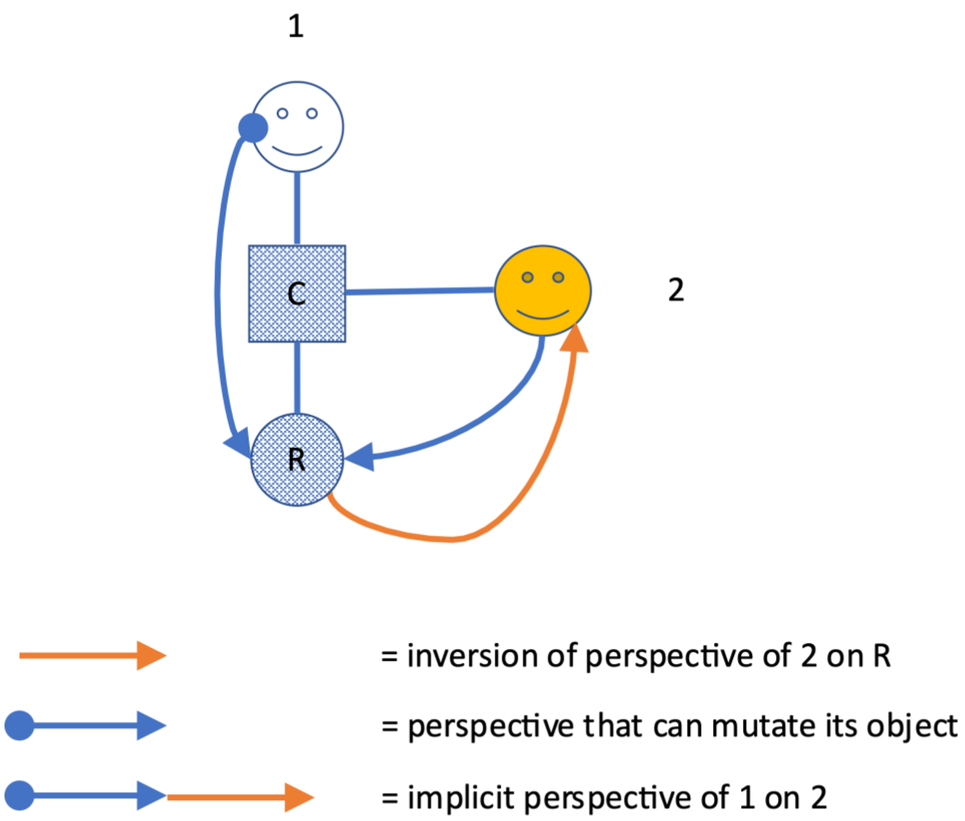
1
C
2
R
= inversion of perspective of 2 on R
= perspective that can mutate its object
= implicit perspective of 1 on 2

Figure 11. User role 1 can modify role R. User role 2 can 'see' role R. This implies that 1 should send his modifications of R to 2, hence has an implicit perspective on 2.

As shown in the figure, we might consider the combination of the modifying perspective on R and the inverted perspective from R to be an *implicit perspective of user role 1 on user role* 2. In other words, we interpret this situation as if there really **was** a perspective of 1 on 2.

Consider the impact of that interpretation. It would mean that as soon as an instance of 2 was added to context C by some role 3 (not drawn), 3 would send that instance to 1, because 1 has a perspective on role type 2.

Similar reasoning applies to the inverse situation that arises as an instance of 1 is added to C. This new peer would receive the user role instance of 2.

In other words, under this interpretation of inverse perspectives, there is no need for explicit perspectives on user roles for the good of synchronization.

However. It would constitute a security breach as we've discussed in the paragraph *Hidden peers*. In the example of the peer-reviewed article, the identity of the Authors would be disclosed to the Reviewers, and the other way round (as their comments on the Article would be directly sent by their PDR's to the Authors'). We may be able to limit the disclosed information to just that needed to send transactions (omitting the identities and other personal information). However, the address information that must be exchanged could be a real giveaway.

For this reason we will not interpret inverted perspectives as implicit user perspectives.

Perspectives over model boundaries

What happens if we give a user role a perspective on a role in a context that is contained in an *imported model?* We can easily do so by either adding that role as a calculated role to our local context, or by specifying the object of the perspective with a query.

As a consequence, modifications made to the object of the perspective should be sent to our user role. But, and this is a **big** but, there may very well be modifying roles in the *original model*. They should have a perspective on our user role in order to be able to send their deltas to it – but this would reverse the dependency relation between the two models!

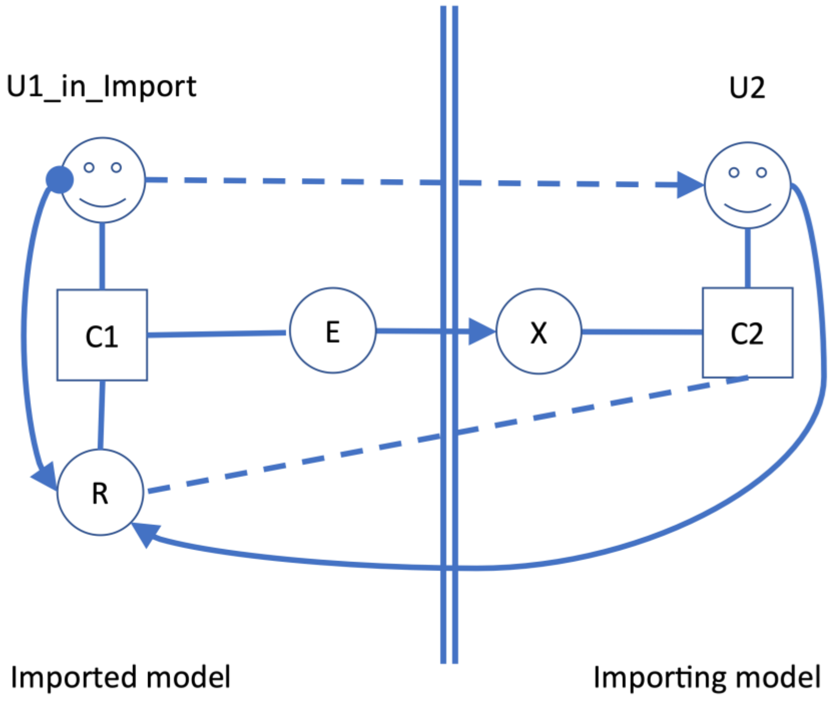

Figure 12. Crossing the border between an importing model (right) and its import. The dotted line between context C2 and role R indicates that R functions as a calculated role in C2. The dashed perspective line from role U1_in_Import to role U2 represents the perspective that is necessary for U2 to receive updates to R made by U1_in_Import.

Figure 12 shows us a model of this situation. On the left we find the model that is imported; on the right the model that imports. Now it is important to realize what happens when the *importing model* is created. It may refer back to contexts and roles in the import, as is shown with role R, that is added to context C2 as a *calculated* role. The query path that calculates it runs from C2 to X, to E, to C1 and then to R. This is a perfectly normal query.

The synchronization problem arises from the fact that user role U1_in_Import has a modifying perspective on R. When it modifies R, it should send deltas to U2 – but it can have no knowledge of U, as the imported model was written before the importing model!

We cannot even adapt the imported model, because it would then need to refer to a role in a model that imports it. This cannot be: the import relation between two models runs one way only.

Solution

We can solve this by adding perspectives *in runtime* to the imported model. This is not as crazy as it may seem; as a matter of fact, we routinely add inverted queries to imports as importing models are added to a PDR installation.

It is important, however, to understand that this only solves the problem when *both the PDR installation whose user fills U1_in_Import, and the PDR installation whose user fills U2, have the importing model*.

To make things easier, when you look at Figure 12, identify the part to the left of the double line with one PDR, the part on the right with another.

In other words, both peers must have the importing model. Otherwise either deltas will not even be sent (when the importing model is absent on the left), or no destination for them will be found (when the importing model is absent on the right).

An alternative way to *pass on*

In the chapter "A synchronisation principle: *passing on*" I describe in conceptual terms an algorithm to make sure a particular delta reaches all concerned peers (peers that have the modified structural element in one of their perspectives). Here I present an alternative algorithm that may be easier to implement.

Overview

Let's start with the user who makes a change (the *Initiator*). He will send a transaction consisting of deltas. To implement the algorithm, these deltas must be augmented with two elements:

- The *user types* with a perspective on the structural element that has been changed (`AllUsersWithAPerspective`)
- The difference between that set and those types that the Initiator has a perspective on (`ToBeInformed`). These are types whose instances the Initiator cannot reach.

Now consider a peer receiving such a transaction. He will carry out the change described by each delta, so his database is in sync with that of the sender. Next, he'll compute all user types he has a perspective on and

- Compute its union with `ToBeInformed` and
- Subtract it from `ToBeInformed`, to form `RemainingToBeInformed`.

For the first set, he computes the user instances known to him. For them he creates a personal transaction and adds this particular delta to it, *after replacing the element* `ToBeInformed` *with* `RemainingToBeInformed`.

When all incoming deltas have been handled in this way, he will send the personal transactions that have been built in the process to their respective recipients.

This algorithm will make sure that no user receives the same delta twice and it guarantees that all users receive the deltas they need.

Some implementation details

Computing the set of user types connected to a particular user type

This algorithm requires all users to compute the users they have a perspective on, quite often. It seems worthwhile to compute this graph once during the processing of the model source file and save it with the DomeinFile. We compute this graph for *the entire model*, rather than per context, because when we deal with a particular change, users outside the context may have an (implicit) perspective on it.

We start with the perspectives stated explicitly in the model source text. These form the base of the UserGraph.

Calculated User Rule. For the next step, realize that some user roles having a perspective may be calculated. We compute the extension (in terms of Enumerated Roles) for such a user role and add them to the graph. We then connect all these new roles to the same targets as the original Calculated User Role.

Inverted Calculated User Rule. The user role that a perspective is on, may be calculated, too. Like above, we add to the graph all Enumerated Roles that form the extension of the Calculated target and we connect the user having the perspective with all of them.

Filler Rule. An Enumerated User Role U1 that is filled with another User Role F1, gives rise to even more connections. If U1 has a perspective on U2, its filler F1 should be connected in the UserGraph, too. It's not that the filler has the same perspective; but we know that F1 'can see' U2.

Inverted Filler Rule. An Enumerated User Role F1 that has a perspective on another user role U2, will be connected in the UserGraph with an arrow. But then a user role U1 that is filled by F1 may be connected to U2 as well. After all, U1 and F1 represent the same user (or installation) and the UserGraph shows connections between users.

Computing user types for a delta

For any given delta, we have to compute all user types with a perspective on that type. Notice that because a user role may have a perspective on a calculated role, not only the end results of such calculations are in scope for him, but all intermediate steps are so, too. How to compute them?

This is where inverted queries come to the rescue. We have established, while processing a model source text, an inverted query on each element a user can 'see' by force of a perspective. So, starting with the type of a structural element, we can read off all user types that have it in scope.

These user types are contained in the newtype `InvertedQuery`. We use this type to compute users instances that should receive a particular delta. To implement this algorithm, we should add some code to collect the user type from all inverted queries at a particular role- or property type and then add them to the delta's constructed for the modification.

Optimization by a refinement

What if there are more than one user instances for a particular type? All of them would carry out the above algorithm, duplicating work and moreover burdening peers with double work too, if they have something to pass on.

We can avoid that by having the Initiator select a single instance of each type at random and only send him a delta with a non-empty `ToBeInformed`. The others receive a delta with an empty `ToBeInformed`. Subsequently, each peer that receives the delta does the same. This guarantees that for each user type, just a single instance will pass the delta on to types that could not have been reached before.

Checking for incomplete synchronization (obsolete!)

The text in this paragraph is out of date, being overtaken by a better insight. See chapter "Connecting users" *instead. I've left this chapter in the document because finding unconnected subgraphs may turn out to be relevant, after all, for a different reason.*

In the paragraph "The User Graph" we have argumented that if the user graph of a context[10] consists of unconnected subgraphs, synchronization cannot be complete. This is an extreme case. A problem more likely to occur is that the graph is fragmented *when projected on a particular element*. After all, exactly what have we in mind when we speak of synchronization? For any element that may be changed (or created or destroyed), we can remove from the graph those user nodes that have no perspective on that element. This is because, with respect to the particular element, those user nodes play no role in synchronization. The remaining graph is its *projection* on that element.

[10] Notice that this graph may well contain user types *from outside the context*. This is because of calculated roles; queries executed for a user that may trace a path through a context of which this user is not a member. We may consider these users to have an *implicit perspective*.

Obviously, the reduced graph will have less connections and may very well consist of unconnected subgraphs. In that case, synchronization is said to be incomplete for that element.

A complete synchronization model check requires us to carry out the above for each structural element. In the next paragraph we describe how to find the user types to project the user graph on for each such element.

Should we take into account whether a perspective can modify?

Some user roles have a perspective that allows them just to consult a structural element. There must be roles that have a perspective that allows them to modify the element, otherwise nothing would happen.

When a graph consists of unconnected subgraphs, each user is in exactly one subgraph. So each 'modifying user' is in a subgraph. This means his modifications cannot be sent to users in the other subgraphs. So synchronization is incomplete.

Conclusion: the very existence of unconnected subgraphs is a strong enough test of incomplete synchronization.

Finding user types to project the graph on

As I've explained above in the paragraph "Computing user types for a delta", we have to inspect `InvertedQuery` elements to find users with a particular perspective.

Adding and removing role instances

An `EnumeratedRole` type has two members that contain `InvertedQuery` elements that relate to adding and removing role instances:

- `onContextDelta_context`
- `onContextDelta_role`

The difference between the two has to do with the direction of the query that passes through a given instance of this role type. It is of no concern to us here: we just compute the union of all user role types we can find in the `InvertedQuery` elements in both members.

Adding and removing a filler

Similarly, an `EnumeratedRole` type has two members that contain `InvertedQuery` elements that relate to adding and removing a filler (binding) to a role instance:

- `onRoleDelta_binding`
- `onRoleDelta_binder`.

Modifying property values

Finally, `EnumeratedProperty` types have a member `onPropertyDelta` that contain `InvertedQuery` elements.

User types may be outside the context

One should realize that the user types collected using the procedures outlined above may result in a set that contains user roles *from outside the context*. This is because of calculated roles.

Informing the modeler

The object of the check for incomplete synchronization is to inform the modeler. What should we tell him?

Suppose we find that for a particular type and modification, synchronization is incomplete because the graph contains two disconnected subgraphs. We would like to say to the modeler something to the effect of: *An instance of role X added (or removed) by a user of types A, B and C cannot be shared with user instances of type D and E*. This would be for role instance creation and deletion.

Similarly, we would say: *When a user of types A, B and C adds a filler to a role instance of X, this cannot be shared with user instances of type D and E.*

And, finally: *When a user of types A, B and C modifies property P in any way, user instances of type D and E cannot be informed about this*.

This works fine for two unconnected subgraphs. But what if there are more? Then we have to resort to a more general message: *With regard to adding or removing instances of role X, the following groups of users cannot communicate: A; B and C; D, E and F.*

In the implementation we use the latter formulation. It also discharges us from finding out in which group modifications can arise (and we'd need that to formulate the first sentences correctly).

Either way, we have to find the unconnected subgraphs in the projection of the graph on a particular type.

Finding subgraphs in the projection of the graph on a type

We can use the following representations and algorithm to find all subgraphs.

Representation of the graph

The graph is directed, may contain cycles and some connections can run both ways. Represent it as a collection of nodes:

```
newtype Graph = Graph (Array Node)
newtype Node = Node {userType :: RoleType, edges :: Array RoleType}
```

From a Graph we can easily create its set of nodes. Projecting the Graph we do by filtering it with a criterium function on Node.

An algorithm in State

To compute the subgraphs, we run an algorithm in the State monad with the following state definition:

```
type SubGraphState =
     { currentSubGraph :: Array UserType
     , subGraphs :: Array (Array UserType)
     , visitedNodes :: Array UserType
     , unvisitedNodes :: Array UserType
     }
```

The algorithm

The algorithm consists of two functions. Both are run in State SubGraphState.

The first function (FindAllSubGraphs) is applied to a graph and is run with the state initialised to have all nodes in the (projected) graph in unvisitedNodes. This function

- terminates when there are no nodes to visit, by adding the currentSubGraph to subGraphs, unless it is empty;
- otherwise
  - adds the currentSubGraph to subGraphs unless it is empty;
  - creates an empty Graph and makes it the value of currentSubGraph;
  - calls the second function (BuildCurrentSubGraph) with the first element of unvisitedNodes.

The second function BuildCurrentSubGraph is defined in scope of the first so it has access to the graph parameter and is called with an argument node.

- adds its argument node to the visitedNodes
- removes node from unvisitedNodes
- adds node to currentSubGraph
- reads the edges for node from the (projected) graph
- filters out those edges that are not in unvisitedNodes
- calls BuildCurrentSubGraph with all edges remaining.

Connecting users

It turns out that a user graph that consists of unconnected subgraphs certainly means synchronization cannot be complete, but it is not the other way round. In other words, this test is not strong enough. Have a look at Figure 13. There are no unconnected subgraphs, but only changes made by Client will be communicated to Sales Person and Controller.

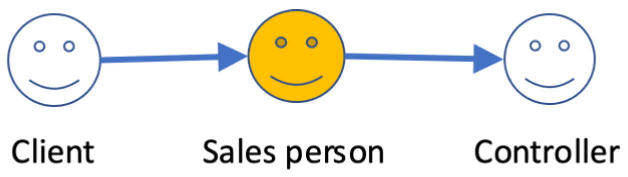

Figure 13. A UserGraph that does not allow synchronization of changes made by Controller.

A better test is to check whether, from a node representing a user with a modifying perspective, all other nodes representing users can be reached. This can be done with a simple depth first search (while preventing loops).

We still want to project the graph for a given mutation, but on top of that we need a list of nodes that represent a user with a modifying perspective.

A special case: modifying a property on the role graph of a user role

Consider Figure 14. This user graph allows for complete synchronization, as the left user is the only one to modify R *and* has a perspective on the only other user who, in turn, has a (consulting) perspective on R.

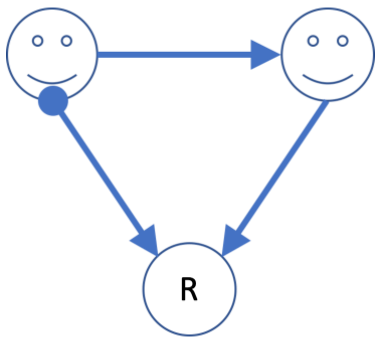

Figure 14. The left user has a modifying perspective on R, while the right user has a consulting perspective.

What about when it is the other user itself that is being modified? Figure 15 depicts this situation, with an added twist.

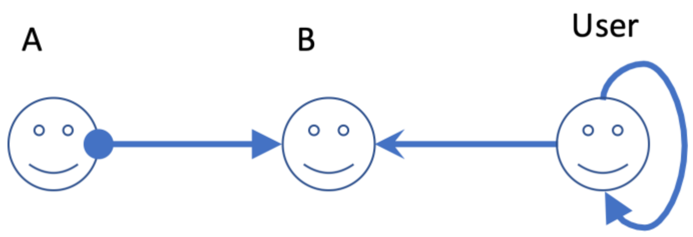

Figure 15. User A modifies a property on user B. B is filled by the system role User, having a self-perspective.

Now suppose that a modifies a property P on User – by virtue of having a perspective on B (this is allowed, as a perspective includes all fillers of the role, recursively). Clearly, because User has a self-perspectives (and we make it include P), he should be informed. The user graph projection for P is given in Figure 16. The synchronization checker will report a problem: changes made by A to P will not arrive at User! This is because a has no expliciet perspective on User.

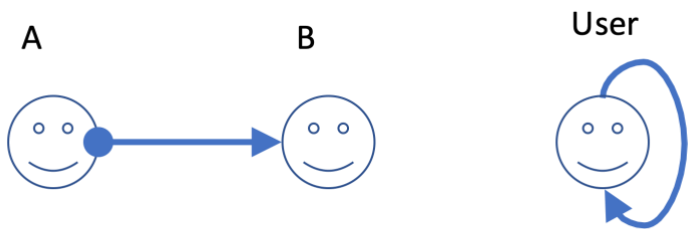

Figure 16. User graph for Figure 15. There is no path from A to Role "User"!

However, a user role with a perspective on a calculated object needs access to all entities along the query path. We say this user has an *implicit perspective* on those entities. This is worked out by establishing an inverted query on each entity along a query path, for the user having the calculated perspective.

We consider the property P on User to be a *calculated property on B*. Consequently, there will be an inverted query on P saying that when it changes, A should be informed.

In other words, A has an implicit perspective on User. This is not just theory but works in practice: User makes sure that A has all information about him that his (A's) perspective allows – due to the inverted query on P.

So the simple connection check between A and User gives us a false negative. Should we include implicit perspectives when we build the user graph? It turns out that this is not necessary (and that is a good thing, because recursively adding perspectives to all fillers of a user role would make the graph grow very big).

Instead, we can simply skip the connection check for projections on a user role U and its properties, *for a modifying user M and U*. The reasoning goes like this:

1. We should check whether M can reach U;
2. We do check because M modifies U;
3. Hence M can reach U.

We should still check *other* user roles with a perspective on U. M may or may not have a perspective on them.