Understanding a Perspective model text

Joop Ringelberg 19-05-21 Version: 2

Types and scopes

With this text we explain how to understand a Perspective model source text. We do not give grammar definitions here, but focus on the meaning of the various expressions instead.

First of all, indentation matters. A model text consists of *blocks* of lines with equal distance from the left margin. Such blocks can be nested arbitrarily deep. We will call such a block a *lexical scope*, or just *scope*.

Second, a model is a number of *type definitions* (at least one: the *domain*). A type definition consists of a *type declaration line*, followed by an indented block that gives the type's details (we might call this block the *body* of the type definition text). A type declaration can be easily recognised because it must be preceded by a keyword:

- a *context* keyword: domain, case, party, activity;
- a *role* keyword: user, thing, context, external;
- or one of the keywords property, view or state.

Any point in a model text (a *lexical* position) will, of necessity, be enclosed in some scope (except for the domain type declaration line). As scopes can be nested, any given point might be inside an arbitrary number of scopes, each enclosing the other until we arrive at the domain's body. Some, but not all, of these scopes will be type definition bodies. Now we will say that all these definitions *are visible* from that lexical position. The first visible context definition that we encounter on 'moving upward', is the *current context*.

Perspectives

A perspective is not a type. A perspective governs what a particular end user, in some user role in a context, can perceive of a role in that context and how he can affect that role. We call the role that a perspective is about, the *object role* while the user role that the perspective is for is the *subject role*. A perspective must be built of three type of parts:

- the *role verbs* that the subject role may apply to the object role, such as Create and Delete;
- the *property verbs* that the subject role may apply to given properties of the object role, such as DeleteProperty and SetPropertyValue;

- the *actions* that the subject role may execute. An action is a sequence of *assignment statements*[1] executed in order, bundled under a name.

Here is an example:

```
domain Parties
  case Party
    thing Wishes
    user Guest
      perspective on Wishes
        all roleverbs
```

User Guest can apply all role verbs to the Wishes role, in the Party context. In order to prepare ourselves for more involved texts whose meaning may not be immediately clear, lets dissect what we have.

We need two concepts: the *current subject* and the *current object*. We've already met the current context, and these two new concepts are just like it. Let's pick the line `all roleverbs` as our lexical position. We must be able to establish for what user role this line is meant (as part of its perspective). Moving upward through the blocks, we encounter the type declaration line `user Guest`. Here is a rule: each user role type definition sets the *current subject*. So we now know `all roleverbs` is meant for Guest.

How about the current object? Well, a `perspective on` line sets the current object. So, putting it all together, for our lexical position `all roleverbs` we have

- Party as the current context
- Guest as the current subject
- Wishes as the current object.

Now it happens to be that a role definition that is not a user role, sets the current object, too. And we also have `perspective of`, setting the current subject. So the following model says exactly the same as our previous example:

```
domain Parties
  case Party
    thing Wishes
      perspective of Guest
        all roleverbs
    user Guest
```

Recapitulating: the current object is in scope in:

- the block following a `perspective on`;

[1] Assignment statements are not explained here. See the text Assignment. However, an assignment statement either changes a context by creating or deleting a role instance, or changes a role by filling it with a role (or remove that filler) or by changing the values of its properties.

- the body of a role definition that is not a user role

The current subject is in scope in:

- the block following a `perspective of`;
- the body of a role definition that is a user role; idem
- the block of assignments following `do for`, defining an action (we'll see examples below).

In each of these cases, exactly one user role type is specified. Hence the current subject is always a single named type (but notice this may be a calculated role type that defines the disjunction of two types).

Notifications and state

Often, we want to make sure that the users of an application take note of some changes. In our party example, it is important for the person throwing the party to note that his invited guests accept (or reject) the invitation. To make sure that some changes do not go unnoticed, we introduce the mechanism of notification.

A *notification* is something that draws the end users' attention. Usually, a notification also consists of some message (like a line of text). Crucially, a notification *should happen under specific circumstances*. In Perspectives, we model this with *state transitions*.

A context can be in a specific *state*, and so can a role. This state must be defined in terms of its roles (for a context), while the state of a role is defined in terms of its filling and/or property values. In Perspectives we can write an *expression* with some truth (Boolean) value to define a state[2]. Here is an example:

```
domain Parties
  case Party
    user Guest
      property FirstName (String)
      property Accept (Boolean)
      state Accepted = Accept
        on entry
          notify Organizer
            "{FirstName} has accepted."
    user Organizer
```

The state declaration line shows that role state Accepted is simply determined by the value of the Boolean property Accept. As soon as the end user playing the Guest role sets

[2] We do not explain expressions in this text in detail. In short, an expression can be some comparison whose left and right terms trace paths through the web of contexts, roles and properties.

that property to true (e.g. by ticking a box in the invitation screen), the Guest role *enters* the state Accepted, or *transitions* to that state.

But what was the previous state of Guest? By default, each context or role is in its resting, or *Root,* state. So by ticking the box, role Guest transitions from Root state to Accepted state.

Let's do some lexical analysis. `notify Organizer` is our lexical position of interest. What is the state transition it applies to? Following the nested scopes upward, we encounter an `on entry` line. This specifies a transition type (there is only one alternative: `on exit`). But entry of what state? Continuing our exploration of nested scopes upward, we encounter the state type declaration line for Accepted. Here's another rule: in the body of a state type definition, that state is the *current state*. This completes our quest: `notify Organizer` applies to entering state Accepted.

A little more on the nature of notifications. In the example above, we might be tempted to notify the Organizer with the text "I will come" rather than "X has accepted.". After all, it is the Guest who accepts and this could be his personal message to the Organizer.

But a notification is not a message from one user to another. It is more like an act of observation by the notified user. The difference is subtle: users should be observant and note what happens to contexts they play a role in. The notification mechanism merely aids them in actually noting that some changes have occurred. It is not a *conversation* mechanism; we have other means for that[3].

More on states transitions; automatic effects

Life is full of repetitious tasks. Automation can take care of them and Perspectives is no exception. But when we make something happen automatically using Perspectives, it will always be *delegated by some user role*. Here we mean by 'something happens' that the information recorded in terms of contexts, roles and their properties, changes; in other words, that the state (of some context or role) changes.

So to make this very clear: state changes are always traceable to some end user.

Let's consider a birthday party. On entering the Root state of the Party (which happens as we create it), we'll create a role PartyPig (to be filled later by a person). This is the model:

```
domain Parties
  case MyParties
    user Organizer
      perspective on Parties
        only CreateAndFill
    context Parties filledBy BirthDayParty
```

[3] Not yet at the time of writing, but in a future release.

```
      on entry
        do for Organizer
          createRole PartyPig in object >> binding >> context
  case BirthDayParty
    user PartyPig
```

The first thing to note is that all birthday parties (all instances of BirthDayParty) are embedded in the context MyParties. The context role Parties holds them. The user role Organiser can create a new one and fill it automatically with an empty embedded context (an instance of BirthDayParty) as well, due to the role verb CreateAndFill[4].

But this context is created empty and we always want to have an instance of the PartyPig role in it. This is where the `on entry` comes in. We then automatically create a role PartyPig, an automated task delegated by the Organizer user role.

You will notice the clause `in object >> binding >> context`. It traces a path from the new Parties role instance (the current object[5]) to the new embedded BirthDayParty context that it is filled with. This is where we create the new instance of the role PartyPig.

We might have written this model like this:

```
domain Parties
  case MyParties
    user Organizer
      perspective on Parties
        only CreateAndFill
        on entry of object state
          createRole PartyPig in object >> binding >> context
    context Parties filledBy BirthDayParty
  case BirthDayParty
    user PartyPig
```

There is no difference in meaning; just in the way we express it.

By now you will have inferred that the line `do for Organizer` sets the current subject to Organiser. But why do we write on `entry of object state` and not just `on entry`, like we did in the first formulation?

This has to do with the notion of current state. We've seen that in the body of a state definition, that state is the current state. But what is the state outside of such scopes?

[4] This is the *only* way we can create contexts. Thus, each context is always embedded through a context role in some other context. The role verb CreateAndFill governs this.
[5] Notice that the current object as we've defined it above is actually available as the value of the variable `object` in expressions. As a matter of fact, we could have left it out of this expression. Just `binding >> context` would do, as the expression is applied to the Parties role anyway. More on that below.

By construction, in the scope of a context definition, the current state is the Root state of that context. Similarly, in the scope of a role definition, the current state is the Root state of that role. Finally, we also have `in state` expressions that, unsurprisingly, set the current state.

So in our second formulation of the model, the current state in the lexical position at the start of the line `on entry of object state` is the Root state of the Organizer role. But we obviously do not want the automatic effect to take place on creating an instance of the Organizer role – it should happen when we create an instance of the Parties role! This is what `on entry of object state` does for us: it sets the current state that the `on entry` applies to, to the Root state of the current object. And this happens to be Parties (it is the first enclosing scope that sets the current object).

Looking back to the first formulation, we can understand why `on entry` works. The current state at this lexical position is given by the declaration `context Parties` - and thus is the Root state of Parties.

There are a lot of ways to set the current state. Summing up:

1. A state definition sets the current state for its body[6].
2. A context definition sets the current state to the Root state of that context.
3. A role definition sets the current state to the Root state of that role.
4. The `in state X` clause sets the current state to its substate X. When `of` and a state type (object, subject, context) is specified, the current state is set to the substate X of the current object, current subject or current context respectively.
5. The `on entry` and `on exit` clauses do by themselves not change the current state, but specify a state transition for the current state. When `of` and a state name is specified, the state transition is for the substate of the current state. Alternatively, you may think that state name changes the current state for the lexical position of `on entry X` and `on exit X` to substate X of the current state.
6. The `on entry` and `on exit` clauses can be augmented with three parts:
    a. `of object state`, optionally extended with a state name. It sets the current state to the Root state (or the named state) of the current object;
    b. `of subject state`, idem, for the current subject;
    c. `of context state`, idem, for the current context.

These definitions and clauses give us full control of specifying the conditions under which something may happen automatically, in various ways. Some examples:

```
on entry
on entry of Published
on entry of object state
on entry of object state Published
```

[6] In all rules we list below, 'setting the state' holds for the lexical scope following the declaration or clause lines.

```
  in state Published
  in state Published of object
  in state Published of context
```

Perspective and state

A user role might have different perspectives in various states. Let's revisit our first example:

```
domain Parties
  case Party
    thing Wishes
    user Guest
      perspective on Wishes
        all roleverbs
```

What is the current state in the lexical position `perspective on Wishes`? Its narrowest enclosing state giving scope is the body of the `user Guest` definition, so it is the Root state of Guest. The implication is that this perspective on Wishes is always valid.

Why always? Would Guest not lose the perspective on the very first state transition? No, because whatever state Guest would transition to, it *must* be a substate of its Root state. This means that Guest then would be in both the substate and the Root state. In other words, perspectives for the Root state are always valid.

In contrast, in this model:

```
domain Parties
  case Party
    thing Wishes
    user Guest
      property Accept (Boolean)
      state Accepted = Accept
        perspective on Wishes
          all roleverbs
```

Guest would only acquire a perspective on Wishes in state Accepted. That is, the state Accepted of the role Guest.

We might call this *subject state*: the perspective depends on the state of the subject. It is also possible to define a perspective dependent on object state:

```
domain Parties
  case Party
    thing Wishes
      property Finished (Boolean)
      state Published = Finished
    user Guest
```

```
      in object state Published
        perspective on Wishes
          all roleverbs
```

Now Guest can only see the Wishes when they are published. The perspective no longer depends on the state of Guest.

As of yet, we cannot make a perspective dependent on both object and subject state.

Obviously, we can also define a perspective to be valid in some context state. That means, in this case, that we can actually make the perspective depend on both object and subject state:

```
domain Parties
  case Party
    thing Wishes (Functional)
      property Finished (Boolean)
      state Published = Finished
    user Guest
      property Accept (Boolean)
      state Accepted = Accept
        state WishesPublished = context >> Wishes >> Finished
          perspective on Wishes
            all roleverbs
```

Subject role state Accepted now has a substate called WishesPublished. Its definition depends on the same property Finished of role Wishes as the Published state of Wishes itself (but we need a path via the context to reach it). So, whenever Wishes transitions to Published, a Guest user role instance in state Accepted will transition to its substate WishesPublished and thus be in both states at the same time. So we succeed in mimicking the effect of making the perspective depend on both object and subject state.

This works, however, only because Wishes is a *functional* role (you'll notice the Functional qualifier in parentheses on the role declaration line). Obviously, Guest is not a functional role and this means we cannot mirror this solution by reaching out from the role Wishes:

```
domain Parties
  case Party
    thing Wishes (Functional)
      property Finished (Boolean)
      state Published = Finished
        state GuestAccepted = context >> Guest >> Accept
          perspective for Guest
            all roleverbs
    user Guest
      property Accept (Boolean)
```

```
        state Accepted = Accept
```

Look at the declaration of GuestAccepted: exactly what Guest are we talking about? The expression `context >> Guest >> Accept` will return as many Boolean values as there are Guests. As a matter of fact, the Perspectives compiler will reject this state definition because the expression is not functional (can result in more than one value).

Summing up: only as long as at least one of subject and object are functional, can we mimic the effect of making a perspective depend on both object and subject state.

About expressions and variables

In the scope of an `on entry` or `on exit` clause we can write a `do` or a `notify` clause. In the `do` we write assignment statements that contain expressions; in the `notify` we can embed expressions in the sentence we want to notify with. Finally, actions consist of assignments and therefore have expressions, too.

An expression is like a function, applied to either a role instance or a context instance. Moreover, we may refer to the current object and current context in these expressions. So, for any given expression, we can ask ourselves the following questions:

1. What is it applied to?
2. What is the value of the `object` variable?
3. What is the value of the `currentcontext` variable?

The answers are dictated by the kind of state the expressions are in scope of. The difference is whether the current state of the lexical position of the expression is object state, context state or subject state.

1. Expressions in a lexical position where the current state is context state, are applied to the current context.
2. Expressions in a lexical position where the current state is object state, are applied to the current object.
3. Expressions in a lexical position where the current state is subject state, are applied to the current subject.
4. The above holds for the state conditions, too: context state condition is applied to the current context, an object state condition is applied to the current object, and the subject state condition is applied to the current subject.
5. Expressions can always refer to the variable `currentcontext`.
6. Expressions in object state can always refer to the variable `object` and it is always bound to exactly one role instance.
7. Expressions in context state may refer to `object` only if the expression is also in the scope of a `perspective on` clause embedded within the context state definition. In that case, the variable can be bound to zero or more role instances.

8. Likewise, expressions in subject state <u>may</u> refer to `object` <u>only if</u> the expression is also in the scope of a `perspective on` clause embedded within the subject state definition. In that case, the variable can be bound to zero or more role instances.

Note the similarities and differences between expressions in subject- and object state. In both cases, expressions are applied to a single role instance. For object state, `object` is bound to that role instance, too. For subject state, `object` is bound to a set of role instances and the instance that the expressions are applied to is **not** in that set!

In general, one must be careful when using subject state. Make sure that assignments are applied to the right roles!