State and Notification

Joop Ringelberg 26-03-21 Version: 2

State: definition

Perspectives describes part of the world in terms of contexts, roles and their properties. But we do not consider such a description to be universally valid or useful. Hence we limit access to its parts by *perspectives* of user roles. Only those participating in a context have access to it and the tacit assumption is that they will know how to interpret its representations[1].

The Perspectives universe is not God-given. Rather, the participants build it piece by piece. There are at least two reasons to do so: to cover larger part of the world by a description and because the world itself changes, so the description has to follow.

Anyhow, it is useful to think about the *state* of a Perspectives representation. This state consists, obviously, of the state of its parts. The state of a context instance is determined by its role instances; that of a role instance by its filler(s) and the roles it fills; and the values of a property type for a particular role instance could be seen as property state. However, for practical reasons we collapse role- and property state together into role state.

Given a particular set of Perspectives types (context-, role- and property types), the number of states a certain description of the world in terms of those types can assume can be very large indeed (if the number of role instances is not limited, the number of states is infinite). Therefore, rather than thinking in terms of individual states, it is useful to think in terms of *state collections*[2].

So when is a state member of a state collection? It turns out that for role state, this is governed by a *proposition*; a sentence in propositional logic. Or, in less fanciful words: when a set of equalities or comparisons of properties, combined with 'and' and 'or', is true. In Perspectives terms, this is a *property query* with a Boolean value.

For context state we use a sentence in predicate logic. That is, we apply quantification, such as 'for all' and 'exists' to roles and contexts. Nevertheless, in Perspectives terms, this is again a property query.

Now from this point on we will use 'state' instead of the more correct 'state collection' and we will say that it is defined by a *state query*. We also associate a *state label* with the query and use it to identify that state.

---

[1] I use description and representation as synonyms in this text.
[2] Some authors speak of micro- and macrostates.

The use of states

What good is the notion of state? It turns out there are three good uses we can put states to:

1. Sometimes we want to be *notified* if a context or role enters a particular state. A prime role state example concerns *presence*. We say an end user is *present in a context instance* if he or she has opened that context on screen (that is, if there is a presentation of his/her perspective on that context instance on screen). Think of a Chat. We want to be notified whenever our conversational partners 'enter the room', so to say.
   Notification is a user interface event.
2. There may be things that we want to happen automatically whenever a context or role enters or leaves a state. These are precisely the 'bots' we make part of our Perspectives models. We can think of bots in terms of *rules* with a condition (left hand side) and action (right hand side); but we can also view that condition as a state query and think of the action as something to be executed as the context enters that state[3].
   Rule firing is a state change event.
3. Last, but not least, we may want to model that a perspective only holds in a given state. An example: in a medical context, some information is only useful when the patient is male (so we want to suppress certain form fields for females).
   We can think of this as user interface state and changes.

Role state versus context state

A modeller might want to specify that a particular user is notified when a new instance of another role is created. At first sight we'd think this is about *context* state, as that is determined by its constituent parts. However, it turns out not to be possible to write a first order logic sentence that captures the notion of 'a new role instance'. This is because the 'newness' is relative to a previous state only: we say an instance is new when it was not in existence earlier. But to recognise that situation, we'd have to write an expression that *accesses a previous state*. However, a Perspectives description of the world does not capture the flow of time; it is a timeless description. We can update that description to reflect a change in the world, but the updating itself is outside of the logical domain. A state condition cannot see it. That would require a whole level of description, and correspondingly a new mechanism to realise it.

However, we can think of this phenomenon in terms of *role state*. That is, if we interpret the coming-into-being of a role instance as a state transition that we can act upon. In

[3] As a matter of fact, thinking in terms of state uncovers something that has eluded us until now: that we might have use for a rule variant that fires when its condition switches from true to false, instead of when it switches from false to true. This corresponds, obviously, to leaving and entering a state.

other words, we can specify that our user must be notified in the *on entry* section of the role *root state*.

Another example of role state would be an invitation situation, where a Boolean property indicates rejection of the invitation. Again, notifying a user of such rejection can only be expressed in terms of state of the role instance, in this case not the entry of the root state but of some substate.

Automatic actions on Role state transitions

We want to be able to prescribe certain automatic actions to be carried out when a Role enters or exits a particular state. These automatic actions are *assignments* that will change state, possibly leading to new transitions. But what are these actions *on*?

For context states, we may have the `object` variable we can use in statements. This variable will be bound to the current object set and is defined when the state transition is described in the lexical context of an expression that gives a role (we have two such lexical contexts: within a role definition and within the `perspective on` expression). We *may have* object; for in the `onEntry` and `onExit` expressions written in a top level state definition, no object is available and it is an error to refer to `object` in assignment statements.

For role states, we can always refer to the `object` variable. It is bound to the role instance whose state changed.

Moreover, unless stated otherwise, a property assignment statement always changes the property values of the role instance whose state changed; i.e. the instance bound to `object`. So these two statements are equivalent:

```
PropertyType =+ 10

PropertyType =+ 10 for object
```

Finally, we can always use the variable `currentcontext` in our expressions.

How to use state in models

Defining state

We introduce into the written form of the Perspectives Language (PL) a new construct:

```
state <identifier> = <booleanQuery> [notify (onentry | onexit)
<notificationLevel>]
```

Here, one can optionally include a *notification level indication* in the state definition. Modified by the `onentry` or `onexit` keyword, this will cause the system[4] to bring this state change to the end users' attention with the given level of urgency[5].

This governs the first use of states: notification. No more is needed to bring state changes to the end users attention.

Using state in rules

Currently, we write a rule like this:

```
rule: <identifier>
     if <booleanQuery> then
       <assignment>+
```

We will keep this syntax, as it is a perfectly legitimate way of introducing anonymous state (the rule name is used for tracing during development)[6].

However, we will make it possible to write this:

```
entering <state name>:
     <assignment>+
```

and

```
exiting <state name>:
     <assignment>+
```

where, obviously, `<state name>` must refer to a defined state (tracing during development will show what state(s) is(are) entered and exited).

Such declarations are part of a user perspective. Automatic actions are only carried out on behalf of a user!

State in user perspectives

Finally, we use state in the specification of a user perspective:

```
perspective on: <RoleExpression>
     in state <state name> only Consult
     in state <another state> all except Delete
       action <identifier>
         <assignment>+
```

[4] I.e. the combination of the Perspectives Distributed Runtime (PDR) and the end user application, InPlace.
[5] Levels to be defined elsewhere; presumably ranging from very much 'in your face' to 'not at all'.
[6] Actually, we can also specify the right hand side as a let-expression: a series of assignments wrapped in a collection of variable bindings.

This shows how some verbs are available to the user in just some states. It also illustrates an *action* available in just a single state. An action is a series of assignments that can be triggered as a whole by the end user. So, to prevent misunderstanding, an action is never carried out automatically, in contrast to rules and on-entry and on-exit assignments.

Alternative modelling

It may be useful to devise another syntax for state and perspectives. Traditionally, states are modelled as syntactical units, with parts specifying entry- and exit actions. In Perspectives we might have states as containers within contexts, defining some roles in some states and not in others. There are problems to be solved, like unifying roles that occur in two or more states but with different perspectives, for example. We consider this to be future extensions.

How to make it work

Representing state in instances

State holds for particular instances. How to represent it? First, we must ask ourselves whether state should be persistently stored. Should state be recomputed on each new session, or should it survive the end of a session?

Ending a session does itself not change the state of a context or role or property as we have defined it here. Hence, there is no *need* to recompute it on session start. Because we want to be able to present the user with a list of notifications that have a certain duration (a notification can be *valid* for some time) and the user can switch off his computer in the meantime, it would mean we would have to recompute state for all context instances on startup. That is clearly undesirable. Hence, state must be persisted.

At first sight, we have two opportunities to represent (and persist) context instance state:

1. as an external property (holding a list of strings representing the state types);
2. as a new member of the context representation.

When we represent state as properties, it will be automatically shared between those who play a role in the context (assuming every user role will have a perspective on the state of the context). Is that what we want? Let's explore some examples.

Consider a medical examination related to a serious disease, having a physician, a laboratory technician and a patient. Suppose a blood test is involved. At some point, the test results are available and the physician should interpret them. The physician should be notified of this state, but the patient should only receive a notification after the interpretation has been added to the test results.

Consider a financial transaction system involving two business parties and an intermediate party. The latter should perform fraud checks. The situation is modelled such that some

broad checks are performed automatically on behalf of the intermediate party. When alarm bells go off, manual intervention is required before further action is taken. Obviously, the alarm bells should not ring for the two business parties *before* the human audit.

We conclude that indiscriminately sharing state would *leak information* that we've carefully kept away from some roles, using perspectives. Notice we're not talking about actual notification, as we can choose to not notify some user roles of some state changes. However, these changes would be sent to their computer and this opens up, in principle, a way for the receiver to get access to it.

In other words: state should not be shared among participants; each should recompute state given the information available according to his perspective[7].

This analysis allows us to decide on state representation in terms of a new internal member of the context instance representation, rather than as external properties.

We add to the context instance representation an Array of the current states that instance is in (and do a similar thing to role representation).

Working with Properties and Verbs

We provide an API function that returns, for a given role instance, an Array of Property-Verb combinations given the state(s) of the role and context and the type of the role the owning user plays in the context. As with other queries, we support the functional reactive programming pattern for these functions. This means that on state change, the user interface program is notified by an invocation of the callback that it provided on requesting the Property-Verb combinations.

This makes it very easy to adapt our user interfaces automatically to changing state, as the visual representation of each Perspective is built on Properties and Verbs.

The underlying mechanism is the same as for ordinary queries: based on dependencies. However, the computation of Property-Verb combinations depends on the states of a role and its context. Hence we record a new type of dependency, the 'state-dependency'.

When that state changes (see below) we record the correlation identifier of the API request for the Property-Verb combinations in the current Transaction in Perspectives State. On subsequently running that transaction, we look up the corresponding effects and apply them (recomputing the combinations using the new states and sending them to the client).

[7] In other words, users do not have an implicit perspective on the state condition.

Applying the inverted-query pattern to state queries

But how does state change? As a state definition consists of a boolean query, we can invert it and thereby make sure that relevant assignments lead to re-evaluation of such queries (just as we do with the previous Bot Action implementation). Actually, this is a two-phase mechanism. On changing some context, role or property, we follow inverted queries to the contexts (or roles) where they are state queries and record these in the current Transaction in Perspectives State.

Then, when we run that Transaction, we re-evaluate the state queries for each context or role that is affected. Whenever a state query evaluates to true, but the associated state label is not in the current states of the context or role instance, we add the label. Conversely, we remove the label if the query evaluates to false. On doing so, we record the correlation identifiers whose computation depends on those states, in the current Transaction.

In a way, a state query is like a rule whose right hand side adds or removes a state label (and also executes entry- and exit automatic actions, see the next paragraph and also the last).

When we then later re-evaluate queries that came in through the API, the relevant state-dependent requests are re-computed.

Automatic actions on entering and exiting states

When we re-evaluate a state query and add a label (or conversely remove it), we also look up all entry automatic actions for the newly added state (or the exit actions when it was removed instead) for the role played by the owning user. These will be executed, triggering state change that may lead to a new round of evaluation of state queries.

Notification

We want to notify the user about some roles and contexts when they enter (or exit) designated states. Being in a 'notified state' is, in some cases, a phenomenon that should persist for some time (see next paragraph). For that reason we record those roles and contexts in specific role types in `sys:PerspectivesSystem`[8].

If the modeller specified, say, state entry notification for state S of context type C, at level L, for user role U, an instance I of C that enters S when the owning user is in role U will be added to the role `ContextNotification` of `MySystem`, with property `Level` having value L.

[8] Thus, this becomes Perspectives State and survives individual sessions.

In other words: we keep a list of contexts annotated with notification level (and another for roles). A client program can request these role instances through normal API calls; so when I is added to `ContextNotification`, the client will be updated.

It is up to the end user program to determine how to actually alert the end user. It may throw up a screen alert, for example. Handling notifications is part of the framework provided by InPlace; it is not the responsibility of the screen programmer of a particular app (model).

Notification life cycle

What happens to a notification when it has been shown to the end user? For some notifications, just showing it once may be good enough. This may suffice for notifying the end user that a chat partner has entered a context. This means that the PDR does not remember contexts that entered the triggering state; after the end user program has received updates, they are discarded.

But for others it might be better to keep them in a list the end user can choose to inspect, until he actively dismisses them. This may be appropriate for reminders to reply to an email; indeed, the very idea of a to-do list is modelled this way. Such notifications should survive the end of a session with InPlace.

So, for notifications we have two dimensions:

- How urgent a notification is brought to the end users' attention;
- Whether it is dismissed automatically, or by hand (or after some time, etc.).

We must research whether these two dimensions can be collapsed into a single set of categories, or need separate representation.

State and Notification mechanism considered to be the old rule system

Currently[9] the PDR transforms Actions for the Bot into rules. When we consider the state query to be the left hand side of such a rule, the right hand side is precisely this:

1. Add the state label to the state set of the context when the lhs evaluates to true and the label was not in the set before (and remove it when false etc. Similar for roles).
2. When an automatic entry is specified with the state for the owning user, run that set of statements (similarly run the on exit statements when applicable).
3. If a notification is specified for when the context enters the state, add the context to the right role in `MySystem` (similarly for notification on exit and for roles).

[9] Version v0.9.0 of InPlace.

This will cause notification sets in the client to be updated and requested Verb-ViewType sets as well.