Parsing, compiling and using models

Joop Ringelberg 10-04-20 Version: 1

Introduction

A model is in effect a collection of types. A model also introduces a namespace. Contexts and roles introduce namespaces, too:

- a context gives its name to the roles and actions defined in it;
- a role gives its name to the properties and views defined in it.

A context can also contain other contexts (notice we're talking types here!) and so provides a namespace to them. This might lead us to think that a model is just another context. And it is. However, a model is a context *without a context*. In other words, it is the root of a namespace hierarchy. We call such a context a *Domain*. So a Domain is a context that contains all other types in the model, recursively.

However, a DomeinFile – derived from a Model's source text – is not just a context. To start with, it is represented differently. In Perspectives, it plays the role of a *package*. All types (contexts, roles, properties, views, actions) are pulled up to the surface and are available in maps as members of a record. The PDR uses this representation to quickly look up any type.

Furthermore, Instances can be packaged with a DomeinFile, indeed, an instance of `model:System$Model` *must* be packaged with it. The external role of this context instance has properties that give its readable name and description, and the url where it resides (a repository on the open internet).

Last, but not least, a DomeinFile/Package lists the *other* DomeinFiles/Packages it's definitions depend on, in the sense that they refer to types defined in them.

To sum up:

- a *Model* is a collection of types;
- a *Domain* is the root context of a Model, having no context itself;
- a *DomeinFile* is a *Package*, the physical transport format of the represented types constituting the model.

Constructing a DomeinFile

Perspectives models are written in the Perspectives Language[1]. A model file is parsed and transformed into a DomeinFile in a process consisting of three phases. A DomeinFile is a data structure that lends itself to easy translation into JSON. It contains the type

[1] Previously called ARC language, from Action Role Context. This name is frequently used in the Perspectives Source code files.

definitions as defined in the model. It also contains an unparsed CRL file holding the instances that belong to the model, such as the model's description.

In this text we give a high level overview of

- the various transformation phases;
- the interdependencies of models;
- how to put a model in operation for a PDR installation;
- the treatment of the model instances, emphasising indexed names.

Transformation phases

Parsing

The raw source text of a model is parsed with the functions in the module `Perspectives.Parsing.Arc`. The output is an Abstract Syntax Tree in terms of the data structures defined in the module `Perspectives.Parsing.Arc.AST`.

Expression parsing

Query expressions form a substantial subset of the Perspectives Language. They have their own parsers, in `Perspectives.Parsing.Arc.Expression`. These functions create an AST in terms of the data structures defined in `Perspectives.Parsing.Arc.Expression.AST`.

Assignments (as in the right hand side of bot rules) are treated by the expression parser, too.

Phase Two

The next step – simply called 'Phase Two' – is performed by functions in module `Perspectives.Parsing.Arc.PhaseTwo`. It is only now that a DomeinFile is created. During this phase, the identifiers of the definitions that have been declared, are qualified with namespaces. Contexts and roles are both namespace-giving (embedded definitions are scoped with the name of their embedder). However, *references* to definitions are left untreated, because a reference may be made in the text before that what it refers to, is fully qualified (the so-called *forward reference problem*). In this phase we also collect the models that the model under treatment depends on (its 'referredModels').

During this phase, we also expand all *prefixed names*. Prefixed names are shorthand for qualified names, making life easier for modellers. Each prefix must be defined for some (context) scope. Phase Two expands all these names[2]. Prefixes as such are lost beyond Phase Two (but the namespaces they refer to are saved in referredModels).

[2] A full treatment of this subject is given in the text: *Expanding prefixed names*.

Phase Three

It is only in the third phase that references are qualified. These are:

- the object and indirect object of actions
- the binding of role definitions
- references to properties (in views)
- references to views (in actions)
- the type of value that is returned from a computed role.

Furthermore, Phase Three is responsible for several other processes:

- expression compilation
- inverted queries for properties and roles
- qualification of binding to a Calculated role
- rule compilation

Expression compilation

Expressions in the source text have, up till Phase Three, been stored as abstract syntax trees. Now that all identifiers are fully qualified, we can look up, for any identifier used in an expression, what its type is. Only now can we compile an expression into a *description of a function that can be executed runtime*.

Let's step back. A query expression is used to define a so-called *Calculated Role* or *Calculated Property*. When an end user requests the value of a Calculated Role through the Api, a function is executed on a context instance and a set of (Enumerated) role instances is returned. It is these functions whose descriptions are compiled in Phase Three.

A `QueryFunctionDescription` is a data type (defined in `Perspectives.Query.QueryTypes`) that gives details of a function's domain and range and how it should be executed. This can either be a primitive (implemented as a Purescript function) or a composition of `QueryFunctionDescription`s. As queries consist for a large part of traversing from contexts to roles and vice versa, we need to be able to look up the type of each role. Hence this compilation can only be done in Phase Three.

The module responsible for this compilation is `Perspectives.Query.DescriptionCompiler`.

From the description of a query function, a Purescript function is computed with the functions in `Perspectives.Query.Compiler`. This, however, is *just in time compilation*: each function is compiled just before it is applied for the first time during a PDR session (the compiled function is cached, so it is compiled only once during each session). In other words, actual query function compilation is a run-time process and does not happen in compilation time (the subject of this text).

Inverted queries for properties and roles

A big responsibility of the PDR is to make information available to those users that are modelled to have a perspective on it. Concretely, if user A adds a role instance to a context instance (as an example), all peers in that context with a perspective on that role should be informed. To this end we calculate, in Phase Three, *inverted queries* that compute in run time, from a role instance, the peers that should be informed. Whenever a role instance is added (run time), we run the appropriate queries and then add a Delta to a Transaction to be shipped to those peers. In order to prevent misunderstanding:

- inverted queries are *constructed* during Phase Three (compilation time)
- inverted queries are *executed* during run time.

Qualification of binding to a Calculated role

A role can be defined with a restriction on the type of roles that can be bound to it (that can fill it). This restriction is often in terms of another Enumerated Role, but may be in terms of a Calculated Role. But the type of a Calculated Role is nothing but a composition of types of Enumerated Roles (it is an *Abstract Data Type* consisting of Sum and Products). So, really, we need to record the type of the Calculated role as the restriction on the binding of such an Enumerated role.

But the type of a Calculated Role is the *range of its compiled expression*. Hence, only after the expressions of a model are compiled, can we really qualify the binding of an Enumerated Role restricted with a Calculated Role.

This step finds all such roles and sets the restriction on their bindings.

Rule compilation

Rules consist of a condition and assignments (where these assignments may be in the body of a let-expression). In this step we compile the condition (just an expression) and the assignments. As with expressions, an AST representing an assignment is turned into a QueryFunctionDescription[3].

Interdependencies of models; packages

We've mentioned in the introduction that a package (DomeinFile) lists the models that the model it represents depends on. How do we deal with these dependencies?

[3] Strictly speaking we should separate expressions (that just have a value) from assignments (that change state). However, their treatment is sufficiently similar to justify lumping them together in the same modules.

Consequences of interdependencies for modellers

A modeller is a PDR user who creates models. He uses functions of the PDR that 'ordinary' end users do not: to parse and compile a model source text to a DomeinFile. With special powers come special responsibilities. In this case that means that a modeller must take care that he has installed, for his PDR, for local use *all models that his model depends on*. During compilation these models are referred. Indeed, when we have IDE support, these models will be consulted to provide type information on referred types.

Consequences of interdependencies for end users

An end user cannot be expected to inspect a model's source text, find the other models that it depends on, and download them from a repository *before* downloading the original model. We have to take care of these dependencies for them. So, on putting a model into operation for some PDR installation, all dependencies must be downloaded automatically first. We will see a little more detail of this in the next chapter.

Putting a model into operation for a PDR installation

To be able to use a model, a user should add the package that contains the models' types to his PDR installation. That is, he should download the package file from some repository and store it in his local database of packages. The function that takes care of this is `addModelToLocalStore` from the module `Perspectives.Extern.Couchdb`[4]. We highlight several facets of this function.

Add model instances

As stated before, a package contains (the text of) a CRL file. This file holds a number of instances, usually of the types defined in the package. It **must** contain an instance of `sys:Model`, describing the model itself. The embedded CRL file is parsed and the instances are added to the users' local database.

Customize indexed names

A model may both declare and use *indexed names* (see the text *Indexed Names* for more detail). Indexed names, such as `sys:Me`, must be replaced by personalised identifiers before they can be used. We must distinguish the occurrence of an indexed name in the model source text from that in a model instance.

In the type definitions, indexed names *only* occur in queries (calculated roles and properties, conditions of actions, right hand sides of binding in a let expression, right hand

[4] This module makes functions available in the namespace `model:Couchdb`, as external core functions that can be executed by `callExternal` and `callEffect`.

side of an assignment expression). Replacement is done when the query is executed – run time.

However, an indexed name in instances that come with a model, must be replaced before that instance is stored in the users' local database. As an example, think of a role instance that is filled with `sys:Me`. Before that instance is stored, its binding should be replaced by whatever identifier is used to identify the user who 'owns' the PDR installation.

Moreover, the replacement should be remembered in run time. As the query interpreter executes a step that is, for example, `sys:Me`, it should look it up and find the unique local replacement for it. How do we set up the lookup table?

This is achieved by the requirement that the *modeller* should bind an instance of each indexed type in the model description (remember that the model description is an instance of `sys:Model`, a mandatory instance packaged with a model). Here is an example, take from `model:System` itself:

```
sys:Model usr:PerspectivesSystemModel
  extern $Name = "Perspectives System"
  extern $Description = "The base model of Perspectives."
  extern $Url = "http://127.0.0.1:5984/repository/model%3ASystem"
  $IndexedContext => sys:MySystem
    $Name = "MySystem"
  $IndexedRole -> sys:Me
    $Name = "Me"
```

Notice how `sys:MySystem` (an instance of the indexed context type `sys:PerspectivesSystem`) and `sys:Me` (an instance of the indexed role type `sys:PerspectivesSystem$User`) are bound in `usr:PerspectivesSystemModel`.

Run time, the PDR retrieves all instances of `sys:Model$IndexedContext` and `sys:Model$IndexedRole` from the local database and constructs a table from the indexed names to the customised identifiers (remember that the indexed names themselves have been replaced by unique identifiers *before* the instances were added to the users' local database!).

Add new indexed names

Indexed contexts and roles introduced by the model will be added to the table of all indexed names that the PDR has compiled on startup.

Set `me` and `isMe`

The PDR keeps track of a number of computed properties for the sake of efficiency. One of these is the `me` member of a context instance representation; related is the `isMe` Boolean property of a role instance representation. The latter is true if the role represents

the current (owning) user; the former is the role in the context instance that has an `isMe` value of `true`. (for more details, see the text *Overview of state change mechanisms*).

After parsing the CRL text with instances, the PDR computes the value of `me` and `isMe` for each of them.

Retrieve dependencies

A model may list other models whose types are used in its definitions. These *dependencies* should be loaded before the models' instances are processed. Currently, the PDR supposes that model dependencies are available from the same repository as the model itself. We may need to change that in the future. The most likely way to do so seems to be to store the repository location with the model name as a dependency (the repository name must be known to the PDR of the modeller, as it must have a local copy of each of the dependencies).

Developing and testing

Tests based on model:System

A test that requires types defined in model:System should be run in an environment where the test user has that package in his local model database, and the model instances in his instances database. So the test needs to put that model in operation by calling `addModelToLocalStore`. We will assume that this model is stable and is stored in the repository. Cleaning up after the test should involve removing the package from the local model database and removing its instances, too[5].

We introduce the function `withModel` to that purpose, defined in Test.Perspectives.Utils. `withModel` will fetch the model from the repository, add and customise model instances, run the test and clean up afterwards.

Testing models

Instead of relying on a (supposedly stable and functional) model, some tests involve aspects of a model itself. The tester can work with various functions to load a source file. Roughly, there are two independent dimensions that these functions vary on:

- to load instances, or not;
- to save to the database, or just cache.

Notice that the tester needs to load model dependencies in his test code. The functions that parse and compile ARC and CRL do not load dependencies!

[5] We have two variants: `withModel` will clear the user database, `withModel'` leaves it as it is.

The full set of functions is:

	Load CRL	Do not load CRL
Cache	`loadCompileAndCacheArcFile`	`loadCompileAndCacheArcFile'`
Save	`loadCompileAndSaveArcFile`	`loadCompileAndSaveArcFile'`