Model versions and compatibility

Joop Ringelberg 08-01-21 Version: 1

Introduction

Perspectives parses an ARC text and transforms it into a DomeinFile (a JSON file format). On doing so, it **checks** the model. The purpose of this check is to prevent runtime errors and to ensure the intended meaning of the model.

As an example of the first category, consider a query that traces a path through the network of role and context instances. On each step, the query interpreter gathers resources from the database. If the modeller wrote a query with two consecutive `context` steps, the interpreter wrongly assumes it can read a property context from a context resource, which is a runtime error.

As an example of the second category, consider a query with a role-step in it for a role type that is not modelled. Runtime no error will rise, as the role instances getting function returns an empty sequence of instances. However, the program obviously cannot realise the intended meaning.

Of course model checking will never ensure intended meaning completely. In the end, that is the modellers' responsibility. More broadly, a model serves a purpose: to support a group of end users in their cooperation. It is the modellers craft to build a model that achieves that purpose.

In this text, I will reserve the word `consistent` to refer to a quality that a model is said to have, if and only if:

- it does not cause runtime errors
- it does not demonstrably violate the modellers' intended meaning.

This latter concept is, for now, operationally defined as the set of constraints that is checked.

Compatibility

A model may refer to another model. We call this second model a dependency of the first; it has a dependent. On checking a model with dependencies, we automatically establish the consistency of the model, extended with the types defined in those dependencies. If a model error is detected, the two models are said to be incompatible; otherwise we conclude them to be compatible.

Model versions

This notion of compatibility is of special significance in the light of model versions. It may happen that a newer version of a dependency is not compatible with a model. This is important information for the users of those models; they would be advised against updating the dependency, or, vice versa, would be advised to update the dependent. That is, if a compatible version is available! This is of concern to modellers; they should care about updates of dependencies of their model. To be more precise, they should check the compatibility of such updates with their models and preferably update their models to maintain consistency with the latest versions of those dependencies.

Referring back to the discussion above, they should also check if those latest versions do not compromise the intended meaning of the program in the sense of good support of the end user.

Backward compatibility with respect to user data

Up till now, we've only discussed compatibility of models. What about compatibility of a model and data? In Perspectives, data consists of context and role instances (collectively called `resources`). Obviously, resources that are constructed with respect to a particular model version cannot - indeed, should not! - cause problems. However, a new model version's types might not fit existing data.

Let's first examine what kind of problems can arise in this relation.

Resources turn out to be remarkably resilient with respect to types. First of all, notice that the **shape** of context and role instances does not depend on their type. This means that no problems can arise in encoding and decoding data when it is moved in and out of the database. In other words, the infamous problem of schema evolution does not arise in the context of Perspectives with respect to the database it makes use of.

Backward compatibility of role instances

How can a role type change?

A role type can

1. acquire or lose a property
2. have the type of a property change
3. have the allowed type of its binding change
4. acquire or lose a aspect
5. change into another kind (kinds are user, context, thing)
6. acquire or lose a view
7. change its cardinality (functional or not)
8. become indexed or no longer indexed
9. become linked or no longer linked

Some of these changes will cause the PDR to throw runtime errors in combination with existing representations of the instances of the type. We'll discuss each of them in turn.

Losing or acquiring a property

This causes no runtime trouble. A new property will, in time, lead to values be represented on role instances. Existing values represented on role instances will simply never be used.

Change the type of a property

This will cause problems for some types, depending on the change. As a String type is not interpreted, a change to that type is frictionless. However, change to Boolean, Number or Date may invalidate some existing string values. Errors would arise on interpreting a string representation as a typed value.

This can be detected automatically by comparing the new model version to the old model version.

We can prevent runtime errors automatically, too, by removing property values whose type has acquired an incompatible change.

Change the allowed type of a role type binding

If the new type is more general than the old type, no problems arise. Otherwise, some existing bindings may be invalid. This mostly boils down to 1, as properties are expected that are not represented. It may happen that the more specialised type shadows an existing property that has another type. This, however, will be detected by the model checker: it is a model time error.

Acquire or lose an aspect

This reduces to 1.

Change into another kind

This would be a huge semantic change. However, there seem to be no runtime errors that can arise because of such a change.

Acquire or lose a view

There can be no runtime errors as a consequence of such a change.

Change of cardinality

There can be no problem if a role no longer is functional. A role that becomes functional will give unforeseeable semantic effects, but there will be no runtime errors. There is no automatic way to fix this semantic problem. In the end, the user should select which of the role instances in a context should be selected as the one that may continue to be present.

Become indexed, or become not indexed

An indexed role is by definition functional. It means that we can refer in model texts to this role instance by a fixed name. Internally, this name is changed into a unique name for each end user. So, while references in the model text must change, there is no consequence for the runtime.

Become linked or unlinked

As a role becomes unlinked, the context instances that hold that role will no longer use the indices stored within them to retrieve the instances of the roles. This causes no particular problems.

However, the other way round is problematic. Without further action, each context instance would lose its role instances for the particular type. This problem can be fixed automatically.

Nevertheless, no runtime errors would occur.

Wrapping up: compatibility of role instances

We first notice that runtime errors only will occur for 2, property types that change their range type. Can such problems be diagnosed before models are put into use? Yes, but only by comparing a new model version with its predecessor. Such a check is really straightforward.

Second, we should notice that semantic errors may arise, too, in cases 7 and 9.

Backward compatibility of context instances

As with roles, let's explore the ways in which a context type can change.

A context type can

1. change its kind
2. lose or acquire aspects
3. lose or acquire roles, where we must distinguish user roles from context roles and thing roles
4. lose or acquire nested context types

5. move to another context type
6. become indexed or become not indexed

Change the kind of a context

Context kinds come in flavours like Domain, Party, Case, Activity and State. There are semantic restrictions on embedding context kinds in other context kinds. However, these are not yet checked. Anyway, these are model issues only.

Lose or acquire aspects

This reduces to losing or acquiring roles.

Lose or acquire roles

The situation is very similar to roles losing or acquiring properties. Again, this causes no runtime problems, in general terms. Role instances that are represented on context instances that have a type that is no longer modelled for that context, just are ignored, runtime.

There is one exception to this rule and that concerns user roles. When a user role is removed, it may be that a particular context instance refers to an instance of that role that represents the end user in that context. This value may no longer be used (as it no longer has a type!). However, just ignoring it means that the end user no longer has access to the context. This obviously is a big semantic problem. But, without handling it properly, there will be runtime errors, too.

Runtime errors can be prevented by automatic action (remove the `me` member from affected contexts).

Lose or acquire nested context types.

This is not a runtime problem. The consequence is that type names change; this should be handled in model time and such changes should be transparent in runtime.

Move to another context type

This is the same issue as 4, but now viewed from the other side.

Become indexed or become not indexed

As with roles, this is not a runtime problem.

Wrapping up: compatibility of context instances

We see only one issue for run time and that is 3, when a context loses a user role type. However, this problem can be handled automatically.

Conclusion: backward compatibility of user data

Based on this analysis, we will say that data is **compatible** with an updated model version, if and only if it causes no runtime errors.

If the updated model version requires changes to the data that can be performed automatically, we say that the data is **fixable** with respect to the new model version.

Existing data (created with a particular model version) may turn out to be either compatible or fixable with respect to a new version of that model. Remember that this applies just to the issue of runtime errors, not to semantic issues!

Compatibility issues with respect to data received from peers

In the above, we've focussed on data created by the end users' PDR. What about data created by the PDR of peers, who may have another version for a particular model?

Let's first consider the situation where the peer uses a *newer* model version. Three kinds of potential problem may occur:

- incoming Deltas may refer to types that are in the new model version, but not in the old model version;

This causes a runtime problem, because the authorisation process will reflect on that type. The problem arises during authorisation. We can catch these problems and choose to ignore the deltas that caused them.

- incoming Deltas may contain property values whose shape is incompatible with the type in the old model.

This does not cause immediate problems upon handling the delta, but will cause runtime errors later on.

We can add a check to the transaction handling and reject any Property deltas whose data have the wrong shape.

- a Context Delta arrives that puts a particular role instance in a context while the role type is unlinked in one of the models and linked in the other.

This turns out to cause no runtime problem. The PDR will either add or not add the instance to the context. The new instance will be found by query, or by following a link. Local logic is consistent!

Now let's consider the reverse case: peer data refers to types from an *older* version of the model (than that the user himself has installed). All of the problems signalled above may arise. However, all we can say is that the peers' model is incompatible with that of the receiving user; we cannot say which is older.

Handling the problems

We can handle both problems, to the cost of ignoring the problematic deltas. This may cause further delta handling problems, but all can be handled to the extent that no runtime error arises.

Semantic problems persist, however. We would like to handle these situations based on knowledge about whose model is older:

- if the receiving users' model is the oldest, one course of action may be to stop handling deltas and ask the end user if he wants to upgrade his local model.
- otherwise, we might want to advise the sender to update his model. However, this would require an entirely new form of communication between peers. As an alternative, we might simply wait. As soon as the user modifies a resource governed by this model himself, and the peer has a perspective on the modified resource, its PDR will find that it uses an outdated model.

Some special cases

Moving a role type to another context type

A modeller might decide that a particular user role is in a context but would be better of in the context that embeds contexts of the first type. As a consequence, the perspectives of that role would have to change - the model checker will signal any omissions.

Runtime errors can be prevented automatically for such a change, but semantic problems will arise. In this special case, however, a precise modification of the data would mend the semantic problems: move the role instances from one context to another, too. This can be done automatically but the modeller should decide if this action is desired.

Moving a property type to another role type

This case may be more frequent than the previous. It will probably be quite common for a modeller to move properties around in the role graph. As with roles, the required adaptation of the data could, in principle, be done automatically. Again, the modeller should decide whether that should be done or not.

Specialising a role's binding

There is no automatic solution to this problem. However, we might find all role instances whose binding has become illegal and present the end user with a means to re-bind them.

Change type names

We can change either the namespace (by embedding the type in another context or role) or the local name. These changes are quite important for models, but should be invisible in the end user data.

Solutions

Recognizing versions

If we include version identifiers with type names, we can quickly establish what situation we have at hand, when a delta arrives and we cannot find a type name. On looking up a type name, we currently split it into a model name and a local type name. Including version numbers requires us to split the name into a third part, its version number, which is not a big deal.

We would immediately recognise a version mismatch. But we can do better. Compatibility over versions can be established per type, with each type recording the earliest (oldest) compatible version (this would need to become part of the model parsing and checking process). Upon recognising a model type mismatch, we can then establish whether the type in our model version is compatible with that of the incoming version (notice this may require us to retrieve a newer version).

Second, the problem of the wrong shape for an attribute value. We would immediately establish whether the problem exists, by checking if the incoming version of the property is compatible with our local version.

Unique model names

The issue of type name changes ties in with another, related potential problem and that concerns the uniqueness of model names. We need our model names to be both readable by modellers and to be unique - worldwide. That should either be solved by a global register (such as the DNS) or by GUIDs. Barring a global register, we must rely on GUIDs. As GUIDs are not readable, we must use a (local) name translation system that gives the modeller the illusion of uniqueness of readable names, while protecting them from name conflicts if they arise. We will equip a DomeinFile with both a readable and a unique name. The modeller can write his text in terms of the readable name: on parsing, the PDR substitutes the unique name. Now if a modeller wants to use two dependencies that have the same readable name, he must use a substitute in his text. The issue is: how to

associate a substitute with a model? The association must be in the model text. The straightforward way would be to include the unique name in the text. For example:

```
use: sys for model:System
use: esys for model:System (<guid>) -- Electrical system modelling.
```

IDE support might assist in inserting the guid.

Handle type name changes

Modellers might change the name of a type from one version of the model to the next.

We check a dependent model, however, by parsing its source again, using the new DomeinFile as dependency. This will cause any reference to a renamed type to be thrown up as an error. An error that could be prevented, if only we knew how to substitute an old name for its new value!

So we need, as the result of parsing an adapted model text, a table that maps old names to new ones.

There appears to be no robust way of building such a table automatically. Instead, we offer the modeller a way to provide hints, so reference errors for dependent models can be avoided. A hint consists of the old name in square brackets immediately behind the new name. The parser builds those hints into a full table. Notice that this must handle changing namespaces, as well.

It should be noted that this is a solution for consecutive model versions only. Clearly, a name that changes in two successive versions, needs a different translation table for dependents that refer to different versions.

How to adapt a source file given a translation table

Given a table that maps, for a given dependency, the old names to new names, how can we adapt the source file to the new situation?

First, notice that references to types outside the model must be fully qualified. Such names come in two forms: expanded and prefixed.

Expanded names are easy to map to their new versions. Replacing them in the source file is just a matter of string replacement.

Prefixed names are more difficult. The main problem is that prefixes only hold for a given syntactical block. A modeller may use the same prefix as a substitute for *different* context names. If the source then contains the *same local name*, in different blocks, with the same prefix (but another mapping) we can no longer substitute globally. Luckily, we can detect that situation and warn the modeller that he must change his source before dependency name substitution can be carried out.

Unique names to protect instance data

Were we to substitute internal names for model type names that would persist through model versions, we could protect references in instances to types to those changes. Succinctly: instead of using the readable name as the type reference in instances, we would use the unchanging internal name.

This is very important. Without this facility, a model type name change must be followed by type reference changes in all instances of that type.

We can achieve this by including in the DomeinFile a translation table from readable names to internal names. Notice that this is a *different* table from the one that maps old names to new names. On parsing the source text again, we look up each readable name in this table and use its internal substitution in the data structures we're building.

By running each name through the old-new table, we can find the internal name for a new name.

A diff function for model source files

By comparing DomeinFiles, we can construct a structure of data that details the differences between a model's types. This structure could be encoded as JSON and we could generate a readable report from that, to be consulted through the InPlace GUI.

We can put this diff structure to good use by providing hints for the modeller that can be shown in the IDE. Particularly, we might draw his attention to references to type names he failed to update after renaming the type (e.g. a property name in a view). We can do this because the DomeinFile refers using the internal name and we can look up the new readable name and compare it with the name in the text.

Another service is that we can detect a new name in a particular namespace, notice that another name has disappeared, and suggest that the modeller may have forgotten to include a renaming hint for the parser. E.g. when a role with local name `MyRole` may have changed to `SomeRole`, the modeller **should** have included the name change in the text (`SomeRole [MyRole]`). We can provide the following hint, in the form of a question: "Did you rename `MyRole` to `SomeRole`?" Obviously these hints are not infallible.