Architecture as Language, Part 2: Expressing Variability

Version 0.2, Jan 27, 2008

Markus Völter
voelter@acm.org
www.voelter.de

Abstract

This second part of the Architecture As Language series looks at implementing variability in the DSL. This open up the Architecture As Language approach as a foundation for product line architectures. We explain how to express positive and negative variability in the DSL and how integrate it with feature modeling. Like in the first paper, we look at the story, the concepts, and – in the last section – the necessary tooling.

Variability Management

A platform

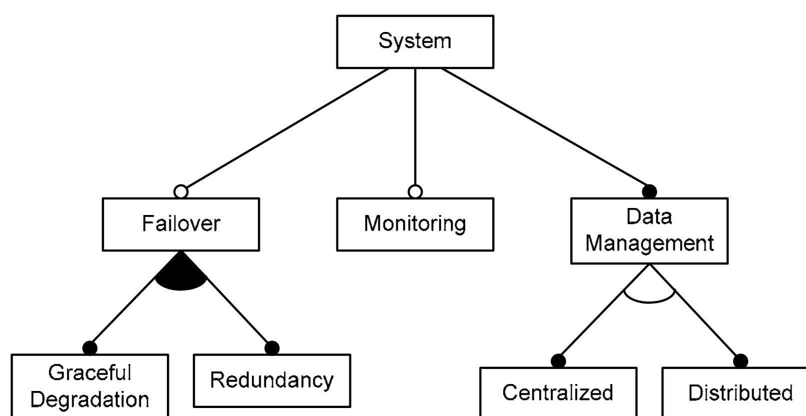
I did not mention the whole story about my customer's system, although you have maybe guessed it: They will be deploying the system to many customers of theirs. Those various systems will contain system-specific components and they will also reuse existing, standardized components. In

addition, they will also need to define variants of their components and systems.

As a consequence we needed to address variability in our models.

Background: Variability w/ Feature Models

Variability management is a well-known activity in product line engineering. A best practice is to manage the variability with formal variability models. For example, feature modeling is a good formalism to do this. The following diagram shows a part of the overall feature model for the kind of system we've been describing in the first part of this paper:



Let us look at the meaning of the notation. The notation describes a system (in the sense of the previos sections). It has an optional feature *Monitoring* (optional is described using the outlined circle). It has mandatory feature *Data Management* (mandatory is shown via the filled circle) which can either be *Centralized* or *Distributed* (either/or is represented via the arc between the lines with the filled circles). Finally, *Failover* can include *Graceful Degradation* or *Redundancy* or both (the filled arc represents n-of-m).

This notation is useful because it only shows the features and their conceptual relationship. The diagram says nothing about how the features are implemented.

However, based on this notation you can easily define combinations of features that describe a valid system. For example a system that includes *Failover* via *Graceful Degradation* and *Centralized Data Management* is valid. Another valid example would be a system with *Monitoring* as well as *Distributed Data Management*.

Expressing the variability on a conceptual level is one thing. However, we also need to be able to connect the implementation artifacts to those conceptual features. We have two kinds of implementation artifacts: the

models expressed via the architecture DSL as well as the manually written implementation code. The next two sections look at connecting those to the conceptual variability models.

Variability in manually written code

Since our systems are completely described via the DSL as well as manually written code, we need to think about expressing variability in those two artifacts.

In manually written code we use special comments to delineate areas of the text file are only present if a specific feature is present. This approach is also used by product line tools such as pure::variants or Gears.

Note that since only some aspects of the overall business logic is written manually, the amount of programming language code that needs to be "marked up" like this is limited. All the technical code is generated.

Variability in the models

Negative Variability

The models are described with a textual DSL that is specific to our architecture. If we want to express parts that are only present in case a certain feature is selected, we can simply extend our grammar accordingly:

```
component DelayCalculator {
  provides default: IDelayCalculator
  requires screens[0..n]: IInfoScreen
  provides mon: IMonitoring feature testing
}
```

This notation describes that the component *DelayCalculator* only provides the *IMonitoring* interface through the *mon* port if the feature *testing* is selected in the configuration model.

The idea behind this is that we can extend the grammar to support the optional addition of a feature dependency for more or less any grammar element.

Let us look at the next example, where we mark up a region of the model to be dependent on the *testing* feature. This is of course much more useful where several aspects of the overall systems depend on certain features.

```
feature testing {

  component MonitoringConsole {
    requires devices:[*]: IMonitor
  }

  instance monitor: MonitoringConsole
```

```
  dynamic connect monitor.devices query {
    type = IMonitor
  }

}
```

We noticed subsequently that instead of having separate *feature* sections, you could reuse namespaces for this:

```
namespace com.mycompany {
  namespace datacenter {

    // … as before …

    namespace monitoting feature testing {

      component MonitoringConsole {
        requires devices:[*]: IMonitor
      }

      instance monitor: MonitoringConsole

      dynamic connect monitor.devices query {
        type = IMonitor
      }

    }
  }
}
```

Positive Variability

Consider again the example above. The monitoring feature can be modularized for the *MonitoringConsole* component and the *monitor* instance, as well as the connector. However, there's also a cross-cutting concern, where each component should provide a port called *mon* associated with the *IMonitoring* interface. We can also modularize this concern using AO technology:

```
feature testing {

  component MonitoringConsole …
  instance monitor: …
  dynamic connect monitor.devices …

  componentaspect * {
    provides mon: IMonitoring
  }
}
```

The component aspect queries for all components in the system and adds a new provided port to them – of course only if the feature *testing* is selected in the product configuration.

Another example for aspects arises from the need to optionally include graceful degradation policies in the system. Let us look at the following piece of model code:

```
component InfoScreen tags (degrade) {
  …
}

componentaspect tag(degrade) feature gracefuldegradation {
  provides shutdown: IShutdown
  requires health: IHealthStatus
}
```

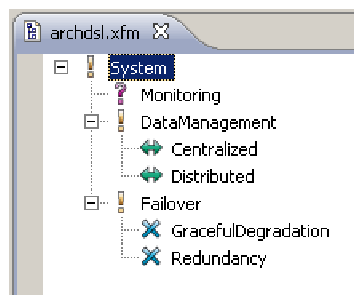
In this example we add the required port *health* through which a component can report local or system-wide health issues to a global system coordinator. Through the provided port *shutdown* the component instance can be told to shut down. Note how the aspect works here: it applies only to components who have the *degrade* tag, and it is only applied to the system if the feature *gracefuldegradation* is selected in the product configuration.

Tooling

Compared to the tooling described in the first part of this paper where Xtext was sufficient to describe the grammar and generate metamodels and editors, we now need additional tooling.

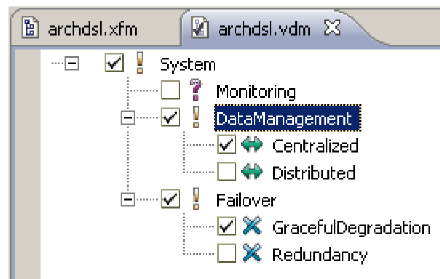
Describing Feature Models and Configurations

First of all, we need a tool to describe the feature models as well as the configurations defined based on them. There are several tools available (fmp, XFeature, pure::variants). While the tooling can be built based on any of them, we would probably choose pure::variants because it is the most mature, the most scalable and it also integrates nicely with an EMF world (see next section). Here is the feature model shown above represented by pure::variants.



They do not use the graphical notation shown above (because it does not scale very well) but they use the same constraints: *mandatory*, *optional*, *1-of-m* and *n-of-m*.

Once you have defined your feature model, you can change to another mode and create any number of configurations. For example, the following illustration shows the *Failover* via *Graceful Degradation* and *Centralized Data Management* product configuration. Note that it is of course not possible to create invalid configurations: the constraints expressed in the feature model are enforced by the configuration editor.



It is also possible in pure variants to express non-local constraints between features and to associate configuration values with them (strings, integers).

Feature Model Integration

The *feature* clauses in the textual model obviously have to mention features that exist in the feature model. You cannot make a piece of model depend on features that don't even exist. This mandates two kinds of integration:

- A constraint check needs to be put in place that validates that feature names that are mentioned in feature clauses do actually exist in the feature model. Xtext supports the validation of such constraints in real time as the user is editing the model.
- You also want to provide code completion for the architecture model. The proposals should only be those features available in the feature model.

Luckily pure::variants provides an automatic EMF export whenever you change something in a feature model. Based on this export, the implementation of the constraint checks and the code completion is a matter of hours and is hence is available in the tooling.

It is also easily possible to get support for the other way round: selecting a feature, show all the location in the DSL files (and the manually written code) that depend on that feature. pure::variants provides an extension point where you can easily plug in to achieve this goal.

Feature Clauses

The *feature* clauses are a way of expressing negative variability. Negative variability means that you code the overall artifact and then selectively remove those parts whose associated feature is not selected for a given product. So the challenge is to implement the actual removal.

This technically very easy: you simply grab all the model elements that result from *feature* clauses. Then we check whether their associated feature is selected and if not, we delete the node that owns the feature clause, including all its children.

Aspect Weaving

The aspect language is of course specific to the DSL itself, hence you cannot write a generic aspect weaver. However, writing, a DSL-specific weaver is trivial, since you define the granularity of the joinpoint model and the expressiveness of the pointcut expressions. In any case, the weaving happens on the level of the EMF model that is parsed from the textual source. Implementing a custom weaver there is a matter of a couple of tens of lines of transformation code.

Summary

I consider this approach for product line architecture extremely powerful. It raises the abstraction level for variation points from the (low level and detailed) implementation code up to (domain specific and more expressive) models. However, it does so while still using textual notations: this has a lot of advantages wrt. to integration with existing developer tooling (CVS/SVN, diff/merge).