A Family of Languages for Architecture Description

Markus Voelter

Independent Consultant, Grabenstrasse 4, 73033 Goeppingen, Germany
voelter@acm.org, http://www.voelter.de

Abstract

In this paper I describe how product line engineering and variant management can be applied to domain-specific languages. I introduce concepts and a tool prototype for describing a family of DSLs used for architecture description. I want to make two points in this paper: First, I want to introduce the idea of product line engineering for domain-specific languages, and second, I want to illustrate why and how this approach is especially useful for DSLs that describe software architectures. The paper is based on practical experience and not on academic research.

Categories and Subject Descriptors D.2.2 Design Tools and Techniques, D.2.11 Software Architectures, D.2.13 Reusable Software, D.3.3 Language Constructs and Features

General Terms Documentation, Design.

Keywords software architecture; domain-specific languages; variability management; product line engineering

1. Overview

Architecture DSLs

Architecture is typically either a very non-tangible, conceptual aspect of a software system that can primarily be found in Word documents, or it is entirely driven by technology ("we use an XML architecture"). Both are bad: the former makes it hard to work with, and the latter hides architectural concepts behind technology hype.

What can be done? As you develop the architecture, evolve a language that allows you to describe systems based on this architecture. Based on my experience in a number of real-world projects, this makes the architecture tangible and provides an unambiguous description of the architectural building blocks as well as the concrete system while still staying away from technology decisions (which then can be made consciously in a separate step).

In other words, I am advocating the use of DSLs to describe the architecture of a specific system or product line.

The beauty of textual languages

I like to use textual languages for this endeavor, for the following reasons:

- First of all, languages as well as nice editors are much easier to build compared to custom graphical editors (e.g. those built with Eclipse GMF)
- Textual artifacts integrate much better with existing developer tooling compared to graphical models based on some kind of repository. You can use the well-known diff/merge tools, and it is much easier to version/tag/branch models and code together.
- Model evolution (i.e. the adaptation of the models in cases where the DSL evolves over time, something you'll always have in practice) is much simpler. While you can use the standard approach – a model-to-model transformation from the old version to the new version – you can always use search/replace or *grep* as a fallback, a technology familiar to basically everybody.
- Lastly, textual DSLs are often more appreciated by developers, since "real developers don't draw pictures".

Graphical notations are useful, of course. Whenever you want to show the relationship between entities a graphical notation is potentially better suited. Also, whenever you want to communicate to non-technical people, graphical languages are typically preferrable because they are perceived to be "easier to understand".

However, there is a different between graphical *notation* and graphical *editing*! Using tools like Graphviz [13] or Prefuse [14], you can easily render a textual model in a graphical way – without being able to *edit* in the graphical environment. Since the model contains the relevant data in a clear and unpolluted form, you can easily transform the model data into a form that tools like GraphViz or Prefuse can process.

The following is an example of a graphviz-generated diagram. It shows namespaces, components, interface, datatypes as well as the dependencies between those.

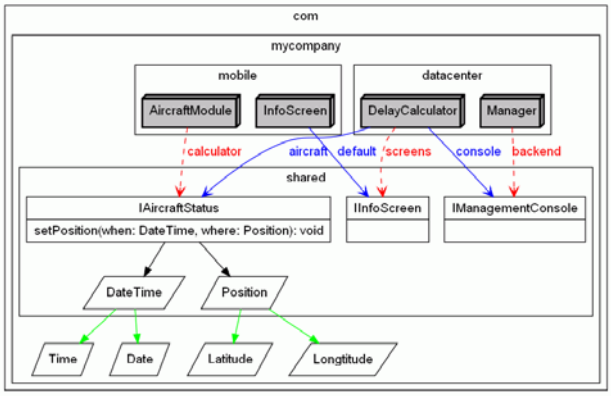


Figure 1. Visualization via Graphviz

The challenge of reuse

I argue above that it is essential that the architecture DSL is developed as you understand your architecture, i.e. that it is specific to the system at hand. Just using an existing, generic architecture description language (such as UML or one of the many ADLs) does not reap the same benefits because you have to shoehorn your domain's architecture into the existing modeling language.

However, that does not mean that there aren't a number of architectural features that are found in many different systems or projects. It is not sensible to put all of them into a "generic architecture DSL", but it is sensible to make it trivially simple to add the respective feature to the DSL once you've identified it as being relevant to your architecture.

Enter product lines: I advocate building a product line of architecture DSL where we use feature modeling to capture the variations. So, instead of doing feature model-based variability management on the concrete system level, we do it on language (grammar) level.

Note that the approach to variant management for languages is of course not technically limited to architecture DSLs. However, specifically in case of software architecture there's a high degree of similarity between different systems, hence the potential and need for reuse is especially high.

Code Generation

Once we have defined a modeling language that accurately reflects the architecture of a software system, and once we have described actual systems with this language, we have to decide what to do with the models. Generally it is my firm belief that if you *don't do something* with your formal models, they are pretty useless (actually, in this case, the process of describing the architecture and the systems has a value in itself, since it helps you understand your own systems much better).

Hence, we will generate two kinds of code models: API code is used by developers to implement manually written business against. Glue code adapts the API code and the manually written code to some kind of implementation/middleware platform.

A central point of this paper is that we describe variants of the architecture DSL. Consequently we also need to vary the code generator, typically based on the same configuration features that are used to define variants of the language.

openArchitectureWare includes a number of features useful to this end. For example, you use aspects for code generation templates, model transformations and workflow specifications to define variants of code generators. The deployment of the aspects can be made to depend on configuration features, too, making the language as well as it's processors depend on the same configuration model. I don't describe this any further in this paper, but you can read more about it in [15].

The structure of this paper

The rest of the paper is structured as follows. Section 2 contains a discussion of how to build an architectural meta model for component-based architectures and also shows a set of typical variations that I came across over the years. It is those variations we'll capture in the feature model. Section 3 looks at how language tooling can be implemented to be able to express the variability discussed in section 2. Section 4 then looks at how users use this variability to configure and customize their own language. Section 5 looks at the current state of the prototype, and section 6 contains an evaluation as well as directions for future work.

2. A Product Line for Component Meta Models

This section introduces a set of typical features of a component DSL. I'll start with defining a set of basic viewpoints and their meta models. The rest of the section then looks at variations of those viewpoints/meta models. These have been extracted from years of work building component-oriented architectures, most of them using formal modeling as a basis.

Viewpoints

A viewpoint describes a specific aspect or concern of a system. It has a limited number of connections to other viewpoints, making each of the viewpoints reusable and well modularized. We use three viewpoints as the foundation for component architectures: type viewpoint, composition viewpoint and system viewpoint. Again, those viewpoints are based on experience gained in many projects over the years. They can also be found in various industry standards (such as EJB or SCA), even if they are not necessarily explicitly distinguished and given names.

Type Viewpoint

The type viewpoint describes component types, interfaces, and data structures. A component provides a number of interfaces and references a number of required interfaces. An interface owns a number of operations, each with a return type, parameters, and exceptions. Alternatively, for message oriented systems, an interface can also be a collection of messages, where a message is named and has a number of parameters. In this case, the interface also defines the direction (in/out) of messages, or even message interaction patterns (*oneway, request-reply, publish-subscribe*)

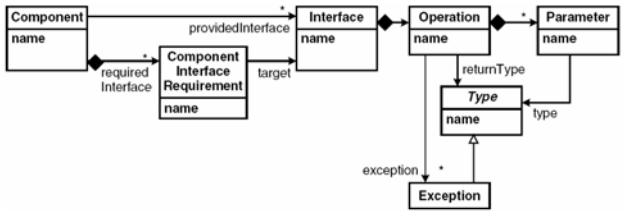


Figure 2. Type viewpoint, components

To describe the data structures with which the components work, we start out with the abstract type Type. We use primitive types as well as complex types. A complex type has a number of named and typed attributes. There are two kinds of complex types. Data transfer objects are simple *structs* that are used to exchange data among components. Entities have a unique ID and can be made persistent (this is not visible from the meta model). Entities can reference each other and thus build more complex data graphs. Each reference has to specify whether it is navigable in only one or in both directions. A reference also specifies the cardinalities of the entities at the respective ends.

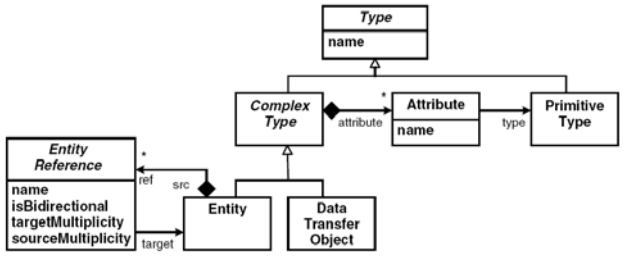


Figure 3. Type viewpoint, data

This data meta model is not much different from the entity relationship model or the SDO standard. In many scenarios, the data meta model can probably be simplified quite a bit, basically reducing it to the equivalent of Java beans or C structs.

The Composition Viewpoint

This viewpoint, illustrated in the following diagram, describes component instances and how they are connected. A *configuration* consists of a number of component *instances*, each referencing its type. An instance has a number of *wires*: a wire is an instance of a *component interface requirement* and hence a connector between component instances. Note the constraints defined in the meta model:

- For each component interface requirement defined in the instance's type, we need to supply a wire.
- The type of the component instance at the target end of a wire needs to provide the interface which the wire's component interface requirement references.

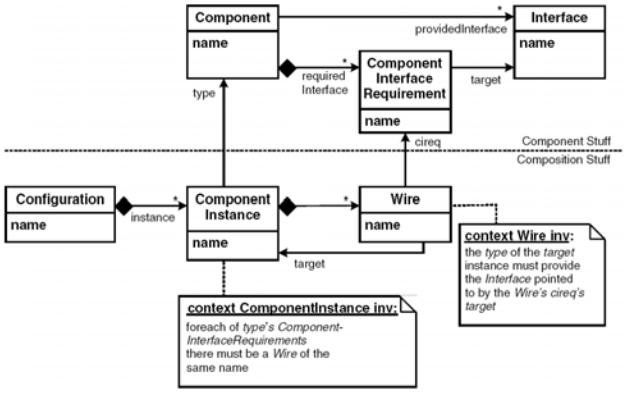


Figure 4. The composition viewpoint

Using the type and composition viewpoints, it is possible to define component types as well as their collaborations. Logical models of applications can be defined. From the composition viewpoint, you can generate or configure a container that instantiates the component instances. Unit tests that verify the application logic can be run in an infrastructure-free environment.

The System Viewpoint

This third viewpoint describes the system infrastructure onto which the logical system defined with the two previous viewpoints is deployed. A *system* consists of a number of *nodes*, each one hosting *containers*. A container hosts a number of component *instances*. Note that a container also defines its kind – this could be things like CCM, J2EE, Eclipse, Spring or a proprietary runtime infrastructure.

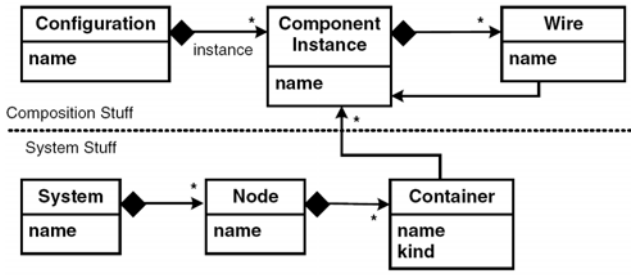


Figure 5. System viewpoint

Based on this information, you can generate the necessary glue code to run the components in that kind of container. The node information, together with the wires (connections) defined in the composition model, allows you to generate all kinds of things, from remote communication infrastructure code and configuration to build and packaging scripts.

Viewpoint Dependencies

You may have observed that the dependencies among the models (and meta models) are well-structured. Since you want to be able to define several compositions using the same components and interfaces, and since you want to be able to run the same compositions on several infrastructures, dependencies are only allowed in the directions shown in the next diagram.

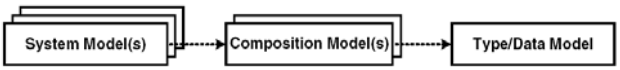


Figure 6. Viewpoint dependencies

Variations

The meta models we describe above cannot be used in exactly this way in every project. Also, in many cases the notion of what constitutes a component needs to be extended. As explained earlier, it is essential that the DSL for describing an architecture evolves and grows with the architecture itself. However, there are common variations. In this section we illustrate some of these.

Separate Interfaces

You might not need separate interfaces. Operations (or messages, respectively) could be owned directly by the components. As a consequence, of course, you cannot reuse the interface "contracts" separately, independently of the supplier or consumer components.

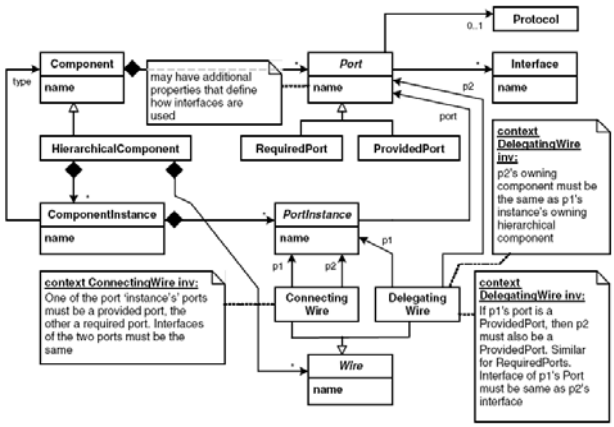


Figure 7. Hierarchical Components

Hierarchical Components

Hierarchical components are a very powerful concept: a component is internally structured as a composition of other component instances. Ports define how components may be connected: a port has an optional protocol definition that allows for port compatibility checks that go beyond simple interface equality. While this approach is powerful, it is also non-trivial, since it blurs the formerly clear distinction between type and composition viewpoints.

Configuration Parameters

A component might have a number of configuration parameters – comparable to command line arguments in console programs – that help configure the behavior of components. The parameters and their types are defined in the type model, and values for the parameters can be specified later, for example in the composition or the system models, or through configuration files.

Component Kinds and Layering

Often you'll need different kinds of components, such as domain components, data access (DAO) components, process components, or business rule components. Depending on this component classification you can define constraints that check whether certain component dependencies are valid or not. You will typically also use different ways of implementing component functionality, depending on the component types. In effect, this gives you a way of layering application functionality.

Another way of managing dependencies is to mark each component directly with a layer tag, such as *domain*, *service*, *gui*, or *facade*, and define constraints on how components in these layers may depend on each other.

State, Threads and Lifecycle

You might want to specify something about whether the components are stateless or stateful, whether they are thread-safe or not, and what their lifecycle should look like (for example, whether they are passive or active, whether they want to be notified of lifecycle events such as activation/passivation, and so on).

Communication Paradigm

Even if a decision has been made for RPC-style communication, it is not always enough to use simple synchronous communication. Instead, one of the various asynchronous communication patterns, such as those described in the *Remoting Patterns* book [16], might be applicable. Because using these patterns affects the APIs of the components, the pattern to be used has to be marked up in the type model.

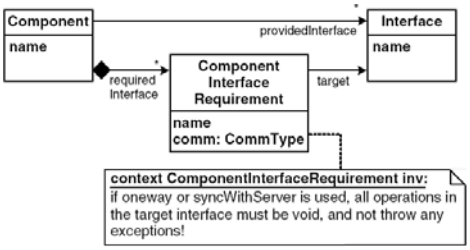
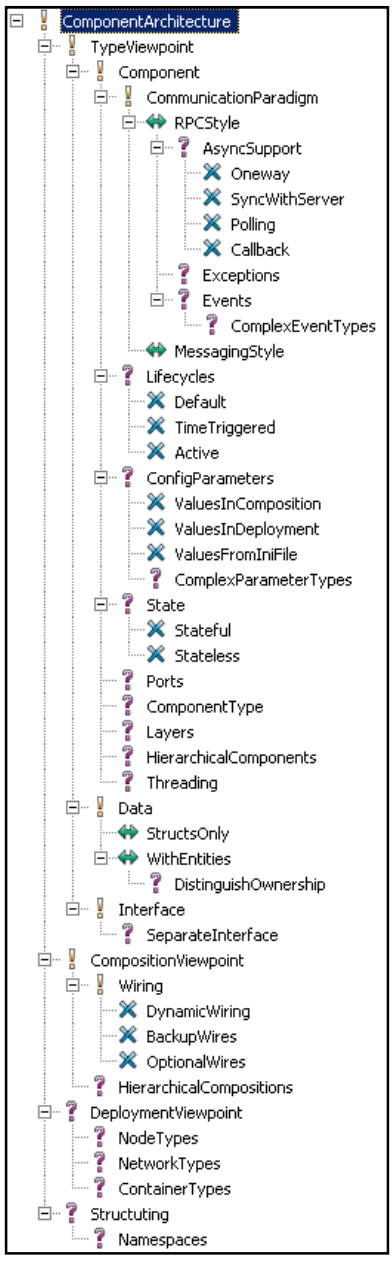


Figure 8. Communication Alternatives

Of course when messaging is used the communication is asynchronous anyway. However, even in that case it makes sense to capture a set of predefined communication paradigms such as *oneway*, *request/reply* or *publish/subscribe*.



Events

In addition to the communication through interfaces, you might need (asynchronous) events using a static or dynamic publisher/subscriber infrastructure. It is often useful that the "direction of flow" of these events is the opposite of the *uses* dependencies discussed above, i.e. they propagate from the *used* entity to the *using* entity.

Static vs. Dynamic Connection

The composition model connects component instances statically. This is not always feasible. If dynamic wiring is necessary, the best way is to embed the information that determines which instance to connect to at runtime into the static wiring model. So, instead of specifying in the model that instance A must be wired to instance B, the model only specifies that A needs to connect to an instance with a specific set of properties: it needs to provide a certain interface, and for example offer a certain reliability. At runtime, the wire is "dereferenced" to a suitable instance using a repository/naming/lookup/trader service.

Higher Level Structures

Finally, it is often necessary to provide additional means of structuring complex systems. The terms *business component* or *subsystem* are often used. Such a higher-level structure consists of a number of components. Optionally, constraints define which kinds of components may be contained in a specific kind of higher-level structure. For example, you might want to define that a business component always consists of exactly one façade component and any number of domain components.

3. PLE for Languages – Tool Implementation

This section explains how to conceptually implement the architecture DSL product line approach for a textual language. We use Eclipse [1], EMF [2], openArchitectureWare [3] and pure::variants [4] as tooling. Specifically, openArchitectureWare's Xtext is used for the textual editor. The way it works is that you specify the grammar for your language, and the meta model, parser and editor are automatically derived from that grammar. In addition, you also have to specify constraints.

Feature Modeling

Feature modeling is used to describe the variability of the architecture DSL. The tall diagram on this page shows a pure::variants feature model with some of the variability mentioned in the previous section.

Based on this feature model, the architecture DSL can be adapted to the needs of a specific architecture as it arises.

Of course, there are facilities that allow for custom configuration, i.e. to put features into the architecture DSL that are not available as a simple configuration option from the feature model.

Variability Mechanisms for Textual Languages

It is not enough to describe conceptual variability in feature models. It is similarly important to actually implement the variability in the artifacts for which we define variations.

In the case described here we want to vary the definition of the architecture DSL (grammar, constraints) as well as the respective editor (code completion, outline, etc.).

In a scenario where the respective artifacts are built with openArchitectureWare's Xtext, this requires variation of the following artifacts: Xtext grammar definitions, check files and extension files. As of now, none of those kinds of files can contain explicit feature dependencies – those artifacts do not "know" they are being varied.

Consequently, we have to use low level "text modification" based on the features. This is similar to Gears' [5] way of implementing variability and is basically a generalized C *#ifdef*. Feature-dependency is expressed with special comments:

```
//# SomeFeature
context Component ERROR "error message":
  here.is.the.actual.contraint.condition;
context Configuration ERROR "another message":
   constraint > 0;
//#~ SomeFeature
```

A preprocessor takes the files marked up with those comments and removes everything for which the corresponding feature is not selected. The marked up file itself contains all possible alternatives (hence this is a form of negative variability).

In the current implementation of our tooling, there is some integration between the text editors and pure::variants:

- Feature names mentioned in artifact files are statically cross-checked with the feature model. If you mention a feature name that is not in the feature model, you'll get an error in Eclipse's Message view.
- Also, you can select any feature in the feature model and see in pure::variants Relations view in which of your artifact files it is referenced.

Customization

Again let me emphasize that is it important to be able to directly represent the architectural concepts of a *specific* system in the architecture DSL. It is therefore not enough to "just" configure a DSL from a set of predefined configuration options, even if these are typical, and hence likely to be a good starting point for your specific system. It is still necessary that the DSL developer can customize the DSL with arbitrary additional grammar.

This is easily possible. The grammar derived from the feature model shown above will contain hooks in various places where customization can happen. It is again based on "text mangling". We show an example in the next section.

4. Using the tools

This section explains how to use the tooling from the perspective of a DSL developer or architect.

Configuring your language

Open the configuration model and select the options you want for your language. In the example here, we want to be able to express stateful components and use exceptions for the operations of our interfaces. We select the *Stateful* and *Exceptions* features as shown in the illustration below.

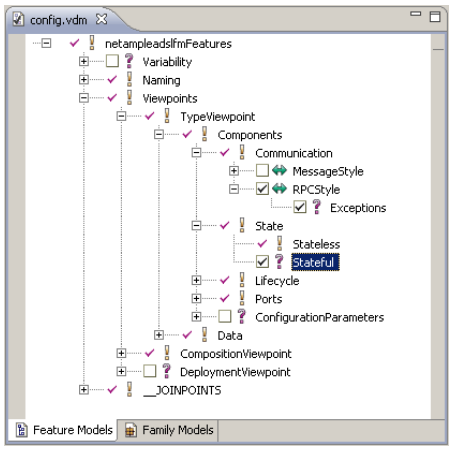


Figure 9. Selecting Stateful Components with Exceptions

Then we regenerate the language (by running the *configure-MyLanguage.oaw* workflow) and rebuild the tooling (running *generateDSLAndTooling.oaw*). We are now able to use the following notation in the language-aware editor:

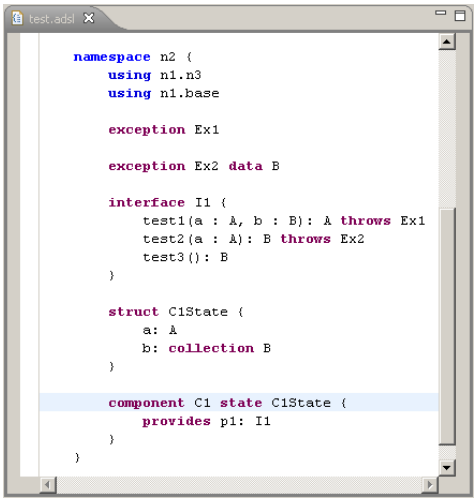


Figure 10. Resulting Language and Editor

Customizing the Language

We call *configuration* the activity of adapting your language by selecting features from the configuration model. This is in contrast to *customization*, by which we mean the extension of the ADSL with your own, project specific features.

There are two fundamentally different ways to go about this: you can define an external viewpoint, i.e. a completely separate language that "annotates" an ADSL model. Technically, this is not an adaptation of ADSL, but can serve the same purpose. Al-

ternatively you can also extend the actual ADSL by adding custom code to a set of predefined hooks.

External Viewpoint

An external viewpoint is especially useful if you want to describe something that relates to the abstractions defined in the ADSL model, but is sufficiently different for it to be expressed with a different notation or by a different role in the development process.

Since Xtext models can be treated as an EMF resource, you can, with the facilities provided by EMF, reference an Xtext model element from any other EMF model and hence "annotate" it. The following example shows how to define another textual DSL to annotate the ADSL model.

Assume you want to define some kind of database mapping for your data structures. To do that, you define a separate DSL using the following piece of code as the grammar:

```
importMetamodel "platform:/resource/
   net.ample.adsl.language/src-gen/net/ample/adsl/
   language/adsl.ecore" as adsl;

DBMapping:
        (imports+=Import)*
        (structMappings+=StructMapping)*;

Import:
        "adsl" file=URI;

StructMapping:
        "map" struct=[adsl::ComplexType|ID] "{" "}";
```

In this grammar we import the generated meta model of the ADSL language and reference elements from it (in the *StructMapping* rule). Once you generate the editor and run it the editor provides code completion and constraint checking into the ADSL file, thereby providing tight integration between the two languages.

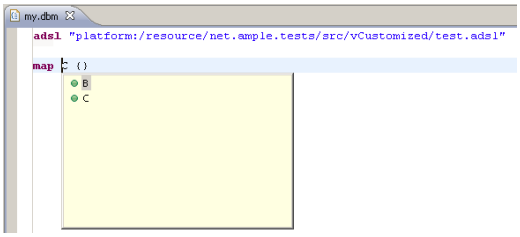


Figure 11. Code-completing into the ADSL file

By loading the persistence model as well as the ADSL model, you can now generate code from both models together – effectively customizing the ADSL model without touching the language itself.

In-Language Extension

The other approach to customization is the custom extension of the ADSL language itself. You can extend the language in all respects, you can even change parts of the configured language. Language customization happens in three steps:

- In the base language (the one you get via configuration) a hook must be defined in all the location where customization is intended. Borrowing from AO, we call these hooks joinpoints.
- You can then define advices (again, borrowing from the AO terminology) that contribute additional code *before*, *after* and *instead of* predefined joinpoints.
- Step three is the execution o the weaver, which actually contributes the advices to the joinpoints.

Let us look at an example. Imagine you want to be able to embed a statemachine in a component. The grammar for a statemachine is probably relatively straight forward. Here is one way of integrating state machines into component grammar:

- We need to add a reference to a statemachine inside a component
- And we'd need to embed the actual statemachine as a top level content in the namespace.

Defining the Joinpoints: We start by defining those two joinpoints in the original overall grammar. In the following piece of grammar, the joinpoints are highlighted in bold (the lines beginning with //>).

```
Namespace:
      "namespace" name=ID
   //# NamespaceFeatureDependencies
                   (featureClause=FeatureClause)?
        "{"
              (usings+=Using)*
              ( subNamespaces+=Namespace
                | components+=Component
                | datatypes+=DataType
                | interfaces+=Interface
                | compositions+=Composition
                //# DeploymentViewpoint
                     | systems+=System
                //# Exceptions
                     | exceptions+=Exception
                //> AdditionalNamespaceContents
                //~> AdditionalNamespaceContents
               )*
      "}";

Component:
      "component" name=ID
      //# Active (isActive?="active")?
      //# Periodic (isPeriodic?="periodic"
                 "(" period=INT ")")?
      //# Stateful ("state" state=[ComplexType|ID])?
      "{"
              (ports+=Port)*
          //> ComponentContentsAfterPorts
          //~> ComponentContentsAfterPorts
      //# ConfigurationParameters
              (configuration=
                   ComponentConfiguration)?
      "}";
```

These joinpoint markers will end up in the generated, configured grammar; note how we use the comment to make the joinpoints invisible to the grammar generator. Note also, that the tooling provides checks against the feature model, so if you refer to a joinpoint that is not defined, you'll get an error message:

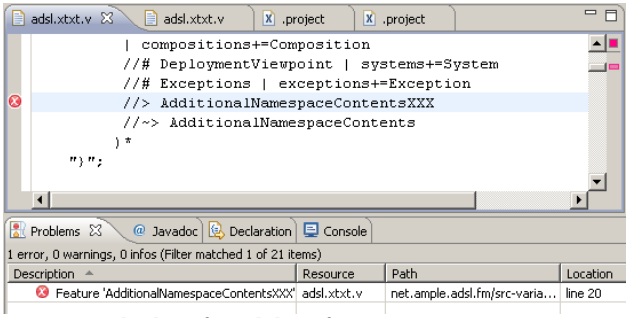


Figure 12. Checking for validity of joinpoints

Note that these joinpoints are not really *configuration* features – however, we still use the feature model to uniquely define the *names* of those joinpoints.

Defining the advice. For the statemachine example, we have to define a number of advices. First of all, we need to define the statemachine itself in the grammar. Note in bold the actual advice syntax. In this case, we put something after (i.e. at the end of) the *TopLevelContents*.

```
//+ after:TopLevelContents
Statemachine:
      "statemachine" name=ID "{"
              (states+=State |
               events+=Event)*
      "}";

State:
      "state" name=ID "{"
              (transitions+=Transition)*
      "}";

Transition:
      event=[Event|ID] "->" target=[State|ID];

Event:
      "event" name=ID ":" operation=[Operation|ID];
//~+
```

To add a statemachine to a component, we need to advice the *ComponentContentsAfterPorts* joinpoint:

```
//+ after:ComponentContentsAfterPorts
      (statemachine=Statemachine)?
//~+
```

Both of these advices are located in a separate file */demo.config/src/adsl.xtxt.v*. The file has the same name as the file into which it is woven into, plus the *.v* extension which is used for all variant files.

We also define a constraint that checks the uniqueness of state names in a statemachine. These need to be contributed to the *net::ample::adsl::language::Check.chk* file, which defines a joinpoint *TopLevelContents* for this purpose. Here's the advice, which, as you might expect, is in the */demo.config/src/net/ample/adsl/language/Checks.chk.v* file:

```
//+ before:TopLevelContents
context State ERROR "State name not unique" :
    ((Statemachine)eContainer).states.
          select(s|s.name == name).size == 1;
//~+
```

After running the "text file weaver", the result is an ADSL version that supports embedded statemachines:

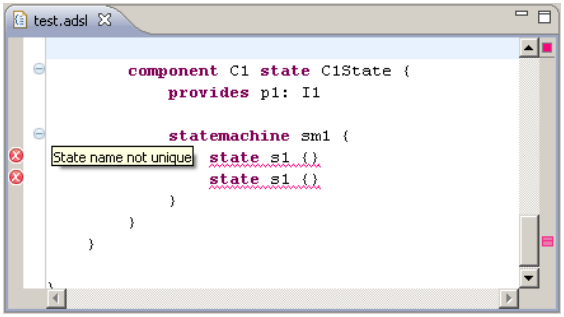


Figure 13. Resulting editor, with state machines

5. The state of the prototype

The prototype has been developed as part of the AMPLE project and is in the process of being made open source. We're looking for interested parties to help develop it further. If you're interested, please contact the author.

The tooling is generally done, but the set of configuration features is limited (ca. 25 options as of now).

There is also a simple Java API generator that generates a mapping of the selected language features to Java; however, there is no generator yet for any specific target platforms.

We also have integrated visualization facilities using Graphviz (for printing) and Prefuse (for interactive visualization).

6. Evaluation, Related Work and Future Work

Since implementing the toolkit, I have used the toolkit for two other customers. We have selected the language features necessary for their architecture, generated the tooling, and used it for real project work. It is fair to say the approach works in practice.

Isn't a generic language good enough?

Describing architecture with formal languages is not a new idea. Various communities recommend using Architecture Description Languages (ADLs) or the Unified Modeling Language (UML) for describing architecture. Some even (try to) generate code from the architecture models. However, all of those approaches advocate using existing generic languages for documenting the architecture (although some of them, including the UML, can be customized).

I don't see much benefit in shoehorning your architecture description into the (typically very limited) set of constructs provided by predefined/standardized languages. The idea is to actually build your own language to capture your system's conceptual architecture. Adapting your architecture to the few concepts provided by the ADL or UML is not very helpful.

So this raises the general question about standards. Are they important? Where? And when? In order to use any architecture modeling language successfully, people first and foremost have to understand the architectural concepts they are dealing with. Even if the UML standard is used to do this people will still have to understand the concepts and map them to the language – in case of using UML that would be an architecture-specific profile. Of course, then, the question is whether such a profiled UML is still

standard. Also, I am not proposing to ignore standards generally. The tools are built on MOF/EMOF, which is an OMG standard, just like the UML, just on a different meta level.

A specific note on UML and profiles: yes, you could use the approach explained above with UML, building a profile as opposed to a textual language. I have done this in several projects and while it does work, my conclusion is that it doesn't work very well in most environments. Here are some of the reasons:

- Instead of thinking about your architectural concepts, working with UML requires you to think more about how you can use UML's existing constructs to more or less sensibly express your intentions. That's the wrong focus!
- Also, UML tools typically don't integrate very well with your existing development infrastructure (editors, CVS/SVN, diff/merge). That's not much of a problem if you use UML during some kind of analysis or design phase, but once you use your models as source code (they accurately reflect the architecture of your system, and you generate real code from them) this becomes a big issue.
- In today's tools, a UML profile cannot remove things the UML provides out of the box. Consequently, the meta model of the model you create is a superset of the (already non-trivial) UML meta model, making it even more complex. Since you want to process your models with generators or transformers, this meta model complexity is an issue to reckon with.
- Finally, UML tools are often quite heavyweight and complex, and are often perceived as "bloatware" or "drawing tools" by "real" developers. Using a nice textual language can be a much lower acceptance hurdle.

Related Work

Architecture Modeling. Using formal languages to describe software architectures is of course nothing new. UML is often used for this purpose, as are the many ADLs that are available on the market [6,7,8]. The approach of defining a domain-specific ADL can also be found elsewhere, an example is AUTOSAR [9] in the automotive world.

However, the approach advocated in this paper is based on defining an architecture DSL that is much more specific to the platform or system being built. The process of defining the language is integral to defining the architecture – architecture definition and language creation cross-polinate each other.

Language Customization. Many general purpose modeling languages provide some kind of customization. Many ADLs allow you to define new "component types" – basically a type label that can be associated with a component. This is a very simplistic approach that does not allow the definition of new architectural abstractions that come with their own structure, constraints and syntax. The approach described in this paper supports arbitrary configuration and customization of languages.

The best known example for language customization is of course the UML with its profile mechanism. I have already discussed this in the *Isn't a generic language good enough?* section above. The approach advocated in this paper tailors a language by actually *removing* features you don't need in a given scenario. Hence the editor, the meta model and all other subsequent model processing is simplified along with the language.

Language Modularization. Being able to define language modules and then integrate those modules into "composite languages" is of course an active area of research. This is a non-trivial problem, because you'll have to somehow combine the parsers. In some cases you'll have to regenerate a new parser based on the combined grammars (that will be the approach available in oAW 5, see below). In other environments (such as SDF, [10]) languages can be combined without regeneration of the composite parser.

Other language engineering environments support the modularization of languages without the need for a parser. Examples include MetaEdit+ [11] (which supports mainly graphical DSLs, where the editor creates the AST directly) and the Intentional Domain Workbench [12], which uses projectional editing even for languages whose concrete syntax looks textual.

Work to be done

More architectural features. Obviously, more architectural features will be added to the ADSL toolkit over time. Based on the more recent customer projects there is already a set of additional features we would like to support.

Java Generator. Also, the Java API generator is not yet completely up-to-date with regards to the variability of the language itself. More work needs to be put into the generator.

openArchitectureWare 5 The facilities for composing Xtext artifacts are limited. For example, there is not much support for grammar modularization in oAW Xtext 4.3. The same is true for composition and modularization of constraint files or other oAW artifacts. As a consequence, we have to do all the variability on text level, using those //# and //> comments in textual artifacts.

In the upcoming oAW 5 framework, the facilities for modularizing oAW artifacts, especially Xtext grammars, will be far more sophisticated.

References

[1] Eclipse platform, www.eclipse.org

[2] Eclipse Modeling Framewok, eclipse.org/emf

[3] openArchitectureWare, openarchitectureware.org

[4] Pure Systems GnbH, pure::variants, http://www.pure-systems.com

[5] BigLever, Software Gears, http://biglever.com/

[6] xADL, http://www.isr.uci.edu/projects/xarchuci/

[7] ACME ADL, http://www-2.cs.cmu.edu/~acme/

[8] AADL http://www.aadl.info/

[9] Autosar, http://www.autosar.org/find02_07.php

[10] SDF, http://www.syntax-definition.org/

[11] MetaEdit+, http://www.metacase.com/

[12] Intentional Domain Workbench, http://intentsoft.com/technology/IS_OOPSLA_2006_paper.pdf

[13] http://Graphviz.org

[14] http://prefuse.org/

[15] Markus Völter, Iris Groher, Product Line Implementation using Aspect-Oriented and Model-Driven Software Development, SPLC 2007, or at http://www.voelter.de/data/pub/VoelterGroher_SPLEwithAOandMDD.pdf

[16] Völter, Kircher, Zdun: Remoting Patterns, Wiley 2004