Architecture As Language

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

Andreas Graf
www.itemis.de
graf@itemis.de
What is a language?

Set of well-defined terms
Stakeholders agree on meaning

Metamodel
Metamodel
Grammar

Metamodel
Grammar
Notation
A DSL is a focussed, processable language for describing a specific concern when building a system in a specific domain. The abstractions and notations used are natural/suitable for the stakeholders who specify that particular concern.
Architecture DSLs
As you understand and develop your Architecture...

Develop a language to express it!
Language resembles architectural concepts

We express the application(s) with the language.
An architectural DSL for embedded systems
Benefits

Clear Understanding from building the language
Unambiguous Vocabulary

Concepts independent from Technology
Programming Model can be defined based on Conceptual Architecture

Architecture „executable“
(i.e. more than rules and docs)
4

Why Textual?
... or: why not graphical?

Languages and Editors are easier to build
Languages and Editors are easier to build

Evolve Language and simple editor as you understand and discuss the architecture, in real time!

Integrates easily with current infrastructure: CVS/SVN diff/merge
adapting existing models as the DSL evolves

Model evolution is **trivial**, you can always use `grep`.

Many Developers prefer textual notations
When a graphical notation is better, you can visualize.
Tooling

Several tools available. Example: oAW Xtext
Specify Grammar

ANTLR Grammar and Parser is generated from this specification
Generated Metamodel

Specify Constraints
Generated Editor

The language-aware editor for our DSL
Generating Code
Since we already have a formal model....

Generate API
Maps Architectural Concepts to Implementation language (non-trivial!)
Implementation
Implementation only depends on the generated programming model API

Programming Model
Generated API + Usage Idioms Completely Technology-Independent
Runtime Infrastructure
Select based on fit wrt. to architectural concepts and non-functional requirements

Glue Code
Aka Technology Mapping Code
Maps API to selected platform
Glue Code
Contains Configuration Files for Platform
Might require „mix in models“

Several Platforms
Different Platforms, not Languages
Support for Scaling (non-functional req)
Benefits:
More Efficient Impl.
Technology Independent
Consistence/Quality
Architecture-Conformance

Code Gen Sequence

1) Generate API
2) Write Impl Code
3) Select Platform
4) Generate Glue Code
Separate Models for stuff relevant for the API vs. system/deployment stuff

- Model 1 (Types)
- Model 2 (System, Deployment)
- Programming Model API
- Glue Code

DEMO III
Generating C for the target device
Exposing Variability
Different Variants of the System for different customers.

How do I express this in the models?
Negative Variability: Conditionally taking something away

Feature Models
component DelayCalculator {
    provides default: IDelayCalculator
    requires screens[0..n]: IInfoScreen
    provides mon: IMonitoring feature monitoring
}
component DelayCalculator {
    provides default: IDelayCalculator
    requires screens[0..n]: IInfoScreen
    provides mon: IMonitoring feature monitoring
}

namespace monitoringStuff feature monitoring {
    component MonitoringConsole {
        requires devices: [*]: IMonitor
    }

    instance monitor: MonitoringConsole

    dynamic connect monitor.devices query {
        type = IMonitor
    }
}
Positive Variability: Conditionally adding something to a minimal core
Positive Variability:
Conditionally adding something to a minimal core

Aspects

namespace monitoring {
    component MonitoringConsole ...
    instance monitor: ...
    dynamic connect monitor.devices ...

    aspect (*) component {
        provides mon: IMonitoring
    }
}
component DelayCalculator {
  ...
}
component AircraftModule {
  ...
}
component InfoScreen {
  ...
}

aspect (*) component {
  provides mon: IMonitoring
}
component DelayCalculator {
  ...
  provides mon: IMonitoring
}
component AircraftModule {
  ...
  provides mon: IMonitoring
}
component InfoScreen {
  ...
  provides mon: IMonitoring
}
Weaver is **generic**: works with all (container) model elements

aspect (*) `<type>`
all instances of `type`

aspect (tag=bla) `<type>`
all instances with tag bla

aspect (name=S*) `<type>`
all instances whose name starts with S
namespace monitoring feature monitoring {

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

  aspect (*) component {
    provides mon: IMonitoring
  }
}

DEMO III

Adding Variability and connectivity to a feature model to the previous DSL
Based on actual practical experience
Currently in use with four of my customers

Benchmarked by suitability for use in today’s projects
THE END.
Thank you.
Questions?