Variability Modeling and Implementation with EASy-Producer

Klaus Schmid, Holger Eichelberger and Sascha El-Sharkawy

{schmid,eichelberger,elscha}@sse.uni-hildesheim.de
www.sse.uni-hildesheim.de
Vision of Product Line Engineering

Key Goal:
exploit commonality in externally (visible) properties of the software (system) in terms of commonality of the implementation

Product Line Engineering vs. Traditional Software Engineering

Project focus → Integrated development of a set of products

Complete Shift of Viewpoint

instead of producing a product and reusing parts produce a set of products in an integrated manner

⇒ Engineer differences
Product Line Engineering is..

..a systematic approach for developing a set of product variants
..the technological basis for software mass-customization
.. a comprehensive framework that consists of two different life-cycles for software engineering

Core idea:

.. variability management helps to organize this
.. crosscutting variability

**Ideally:** central model that supports configuration of all parts
What is Product Line Development?

- Many Systems – a single basis for implementation
- Selection of implementation using configuration

Transformation (bind variability)

Product Lines = binding often during development (but also later)
Challenges in Product Line Engineering

- Need to describe configurations
  - Very expressive
  - Easy-to-use (known concepts)
    - Decision Modeling
    - Similarity to programming

- Need to describe transformations
  - Very expressive
  - Flexible with respect to technologies
  - Open to integration of arbitrary third-party tools
EASy-Producer supports software product line ecosystems
Challenges in Variant-Rich Software Ecosystems

- Introduction of: Product Line Project
  - Derivation (from preceding) units
  - Provisioning of new variability
  - Combine variability and infrastructure / code

- Support
  - Composition operation (multi-product lines)
  - Staged derivation
  - Heterogeneous artifacts
  - Different instantiation mechanisms (even for the same artifact, based on origin of artifact)
  - Complex dependency management
Specific Characteristics of the EASy-Producer Approach

- Product Line Project (PLP) as a single, independent unit (separate configuration management)
  - Acts as product AND infrastructure
  - Supports independent product-specific parts
- Pull-only derivation to support decoupled evolution
- Rich, expressive variability-modelling language (IVML)
- Special language for configurable asset transformation (VIL)
  - Staged configuration support
  - Multi-project composition support
- Integrated template language
- Development and runtime support
- Fast reasoner
- Extendable asset model
Modelling and Implementation with EASy-Producer

The Toolset

Interactive

Configuration View

DSL-based

IVML Modelling Language

Instantiators

VIL & VTL Build Languages
Configuration View
Configuration

- Table-based editor
  - Supports defaults (and freeze)
  - Hierarchical structure
  - Arbitrary non-boolean values

- Supported by reasoning
  - Consistency checking
  - Value propagation
Configuration steps

- **Important:** Derive a new product

In the **new product**
- Change the configuration settings
- Validate the configuration
Instantiation View
Instantiation by Instantiators

- Use of known instantiation plugins
- Instantiation process:
  - Sequence of instantiators
  - Associated artifacts
- Composition:
  - Linking or copying
  - Conflict resolution: Namespace manipulation

- More flexibility: DSL
- Integration of complex instantiators:
  - System call
  - Programming a new plugin
Modelling and Implementation with EASy-Producer

• Produktlinien

Instantiation steps

• **Important**: Freeze the configuration

• Instantiate the product
IVML – A Textual DSL for Configuration
IVML Configuration Capabilities (1)

- Decision-modeling based
- Text-based
- Typed variables (Boolean, non-Boolean including compounds and container, may have default values)
- Derivation and extension for complex types
- User-defined operations
- Introduction of defaults to distinguish “must” vs. local decisions
- Multi-stage default-handling, including default constraints
- Expressive constraint language
- Meta-information: typed annotations

Note: every interactive change is mapped to IVML
Modelling and Implementation with EASy-Producer

IVML Configuration Capabilities (2)

- Project handling: composition, versioning
- Scalability mechanisms: information hiding
- Name-space capabilities to handle conflict-free composition
- Modularization through variability interfaces

⇒ IVML (Integrated Variability Modeling Language)
IVML-Support for handling multiple stages

- **PLP Px**
  - Unbound variability from P and lower-level projects is bound

- **Product Line Project P**
  - If not all values provided, will contain remaining variability
  - P provides values used in the transformation to derive instance (freeze)

- **PLP P0**
  - Defines x, y

- **a and y can be assigned**
  - Freezes x; defines a

---

19.10.2019 © SSE, University of Hildesheim
Modelling and Implementation with EASy-Producer

• Produktlinien

Creation of Configuration Approach

Phase 1: Creation of an integrated, formal approach
• Focus on enabling to express dependencies in a way that:
  – Evolution issues are minimized
  – Dependencies are expressed in a canonical source-independent way
• Requires:
  – Multi-stage default logic
  – Default constraints (vs. mandatory constraints)

Phase 2: Generalization
• Basis for IVML
• Used in several different projects

IVML: Basic capabilities

- Every configuration is a project
- Variability is structured by a rich type system including containers such as sets and sequences and variables are defined based on this
- Variables can have default values

```c
project contentSharing {
    enum ContentType {text, video, audio, threeD, blob};
    typedef Bitrate Integer with (Bitrate >= 128 and Bitrate <= 256);
    ContentType content;
    Bitrate contentBitrate = 128;
    contentBitrate = 128 implies content = text;
}
```

Type definitions

Note, this is a default value
Advanced Variability Modeling: Refinement of Structures

- Decision Variables may be structured in terms of compounds
- Inheritance from compounds is possible

```java
compound Content {
    String name;
    Integer bitrate;
}
compound ExternalContent refines Content{
    String contentPath;
    String accessPassword;
}
```
Example

```c
enum TransmissionType {Manual, Automatic};
compound Car {
    // Manual more fuel efficient
    TransmissionType transmission;
    Integer horsepower;
    Boolean airConditioning;
    airConditioning implies horsepower > 100;
}
Car car;
```
Advanced Variability Modeling: References

- A variability can reference another variability (with the meaning: whatever is configured by this)
- Multiple references may point to the same shared variable, in particular useful when references are stored in containers (not shown)

```java
compound ExternalContent {  
    String contentPath;  
    String accessPassword;  
}

ExternalContent myContent;  
refTo(ExternalContent) myRef = myContent;  
refBy(myRef).contentPath  
    = "http://anyserver.org/content";
```
Advanced Variability Modeling: References

```java
compound ExternalContent {
    String contentPath;
    String accessPassword;
}

ExternalContent myContent;
refTo(ExternalContent) myRef = myContent;
refBy(myRef).contentPath
    = "http://anyserver.org/content";
```
Advanced Variability Modeling: Project Handling (Composition)

- Arbitrary derivation chains
  (arbitrary deep derivation, arbitrary composition)
- Projects have versions
- While (re)using the projects it is possible to
  - require certain versions
  - exclude certain versions

```c
project contentSharing {
  version v0;
  import application;
  import targetPlatform with (targetPlatform.version>=v1.3);
  conflicts application with (application.version>=v2.0);
  application::name = "myApp";
  targetPlatform::name = "myPlatform";
}
```

Note, these are other (possibly external) projects
Advanced Variability Modeling: Annotations

- Variability description entities (and the corresponding assets) can be further annotated
- Goal: simple support for meta-variability
- Annotations may reuse any form of variability concept

```java
project contentSharing {
    enum BindingTime {configuration=0, compile=1, runtime=2};
    // Attaching an annotation to the entire project.
    annotate BindingTime binding = BindingTime.compile
        to contentSharing;
}
```
Advanced Variability Modeling: Annotations

Annotations can be attached to arbitrary sub-groups of variables

```java
compound Content {
    String name;
    Integer bitrate;
}
Content content;

enum BindingTime {compile, loadtime, runtime};
annotate BindingTime binding = BindingTime.compile to content;

content = {name="Video", bitrate=128,
            name.binding = BindingTime.compile,
            bitrate.binding = BindingTime.runtime};
```
More language capabilities

- Project interfaces
- Collection
  - Set
  - Sequence

```c
setOf(Type) variableName2;
sequenceOf(Type) variableName1;
```

- Derived types

```c
typedef AllowedBitrates setOf(Integer);
typedef Bitrate Integer with (Bitrate >= 128 and Bitrate <= 256);
```
More language capabilities

• Freezing variables

```c
freeze {
    contentSharing;
} but (v|v.binding == BindingTimes.runtime)
```

• Explicit evaluation (eval)

• Constraint variables

```c
Integer a, b;
Constraint x;
x = (a > b);
```

• Handling of undefined variables: constraints are not explicitly evaluated
Expression language

- Strongly based on OCL
- Rich set of base relations and functions
- Set and sequence operations
- Quantification

```
contents->forall (t | t.highBitrate <= 512);
contents->exists (t | t.highBitrate <= 512);
```

- A more complex example

```
parameters2->forall (p2 | parameters1->
  exists (p1 | p1.name==p2.name and
typeOf(p1)==typeOf(p2)));
```
Modelling and Implementation with EASy-Producer

EASy Producer: Syntax-driven IVML editor

```java
project PL_Content_Sharing {
  version v0;
  enum ContentType {Text, Video, Audio, ThreeD, BLOB};
  enum ContainerType {Tomcat, IIS, JBoss};
  enum DatabaseType {AzureSQL, AmazonS3, MySQL};
  enum DeploymentTarget {Traditional, Eucalyptus, Amazon, Azure};

  compound Content {
    ContentType type;
  }

  compound VideoContent refines Content {
    ContentType type = ContentType.Video;
    Integer bitrate;
  }

  compound ThreeD refines Content {
    ContentType type = ContentType.ThreeD;
    refTo(Container) threeDContainer;
  }

  compound BLOB refines Content {
    ContentType type = ContentType.Audio;
    refTo(Container) blobContainer;
  }

  compound Container {
    ContentType type;
  }

  compound Database {
    DatabaseType type;
  }
}
```
Experiences with IVML-based Modeling

Initial development

- Based on industrial experience
- Various „Challenge“-Workshops with industrial partners

Further evaluation in different projects:

- All relevant dependencies and configuration capabilities could be represented
- Improved documentation of dependencies
- Easy to learn

Current status

- Successfully evaluated for Klug IS (expressiveness / learnability)
- Is applied for real systems in prototypical manner
- Transition to production use ongoing

Projects:

- EasyCar with Robert Bosch GmbH
- ScaleLog (BMWi / Klug IS)
- EU-Project INDENICA
- EU-Project QualiMaster
- HIS eG
VIL –
A Textual DSL for Transformation
The Transformation Problem

Initial development

Complexity of transformation

- Different artifacts require different techniques
- Influence of configuration options
- Multiple levels of composition
- Various binding times
- Support for derivation networks:
  Assets must inherit their original instantiation mechanism

Evaluation of existing transformation and build languages unsuccessful

Development of a specific approach to transformation
Transformation language: VIL (Variability Instantiation Language)

Combines a number of different programming models

- Object-oriented artifact model
- Procedural Programming
- Rule-based Programming
  - Parameterized rules
  - Wildcard selectors
- Functional elements

+ Use of arbitrary external programs (black-box instantiators)

The language can be used as an arbitrary build language (among other things)
The realization perspective on product lines

Components

VTL

VIL

Configuration
Information
(Variability)

Transformation
(bind variability)

IVML

Product

Produktlinien
vilScript New_Product (Project source, Configuration conf, Project target) {  
  version v0;

  copySRCFiles() = "$target/src/**//*.java" : "$source/src/**//*.java" {
    RHS.copy(LHS);
  }

  copyRESFiles() = "$target/resources/**/*" : "$source/resources/**/*" {
    RHS.copy(LHS);
  }

  main(Project source, Configuration conf, Project target) = : {
    copySRCFiles();
    copyRESFiles();
    velocity("$target/**/*.java", conf);
  }
}
Modelling and Implementation with EASy-Producer

VIL Type Hierarchy (simplified)

Any

String, Real, Boolean, Integer

Configuration types

Map<K,V>, Collection<T>, Sequence<T>, Set<T>, Container

Configuration

Artifact, Path

FileSystemArtifact, RuntimeComponent

FolderArtifact, FileArtifact

VtlFileArtifact, JavaFileArtifact

Extending artefacts

Artefact (meta) model

~30 Types / Instantiators, 160 Operations
Variables / Types

- Types similar to IVM
- Additional Path type – arbitrary reference to transformable element
- Any type – subsumes all possible types
- Container types
  - Collection – abstract supertype of all containers:
    - Set – no duplicates, no order
    - Sequence – duplicates, order
  - Map – associative container (e.g., to represent mappings between different namings)
- const & protected modifiers
- new method – allows temporary artefacts

Not only filesystem
Basic ideas

- @advice: relate the script to the referenced variability model
- Typed language
- Basic script structure
  - `main` as implicit starting goal
  - `rules` → functions are regarded as a special case

```vilstonescript
vilScript New_Product (Project source, Configuration conf, 
  Project target) { 
  version v0; 
  method1(param) = postcondition : precondition { 
    statements; 
  } 
  ... 
}
```
Basic ideas

• Combine
  – Procedural
  – Rule-based
  – OO-approach

• Rules may rely on artifact relations:

```java
copySRCFiles() = "$target/src/**/*.java" : "$source/src/**/*.java" {
  RHS.copy(LHS);
}
```

• Benefits
  – Only do necessary rework
  – Let the infrastructure determine what to work on
Basic ideas

- Rules may rely on the explicit handling of logical expressions:

```java
Boolean processed;
Boolean compiled;

processSRCFiles() = processed==true : {
    ...
}

compileSRCFiles() = compiled==true : processed==true {
    ...
}
```

- Statements can also be handled like normal methods (no post-/pre-conditions)
**Commands**

- Commands given by artifacts
  - Depend on artifacts
  - Generic operations (on any artifact): `new`, `rename`, `delete`
  - `FileArtifacts` (e.g.): `copy`

- `execute` — start any system command
  
  if `cmd` is a path to an executable command:

  ```java
  cmd.execute(params)
  ```
Extensions

- Specialized commands that are treated like language primitives (but are externally realized)
  - Java compiler
    ```java
sEOF(FileArtifact) javac(Path s, Path t, ...)
    ```
  - Velocity
    ```java
setOf(FileArtifact) velocity(FileArtifact t, Configuration c)
    ```
  - Others: Maven, ANT, XVCL, AspectJ
  - Extendable by further bundles
  - Specialized template language VTL
    ```java
setOf(FileArtifact) vilTemplateTemplateProcessor(String n, Configuration c, Artifact a, ...)
    ```
Connecting Decisions and Scripts

- A configuration can be made known – advice-annotation

```java
@advice(ivmlName)
vilScript name (parameterList) extends name1 {
// scriptbody
}
```

This allows to access configuration variables arbitrarily, also in the editor.

- `join` – combine elements from configuration with elements from script:

```java
join(d:config.variables(), a:"$source/src/**/*.java")
with (a.text()).matches("${" + d.name() + "}")) {
// operate on decision variable d and
// related artifact a
```
Control-Flow (1)

- **if** – conditional execution

```java
if (expression) ifStatement else elseStatement
```

- **switch** – multiple alternatives

```java
switch (expression) {
    expression_1 : expression_2,
    expression_3 : expression_4,
    default : expression_5
}
```
Control-Flow (2)

- **for** – iterate over a number of items, collecting the result

```java
for (d = config.variables()) {
    // operate on the iterator variable d of type DecisionVariable (see Section 3.4.5.6)
}
```

- **map** – iterate over a number of items, collecting the results

```java
map(d = config.variables()) {
    // operate on the iterator variable d of type DecisionVariable (see Section 3.4.5.6)
}
```
**Script relations**

- A script may explicitly extend another one
- Explicit instantiate of a higher level script

```
instantiate name (argumentList) [with (version op vNumber.Number)]
```

- Explicit reference to higher-level method

```
super.operationName(argumentList)
```
An Example Script

Copy all sourcefiles and apply velocity on them

```java
vilScript New_Product(Project source, Configuration conf, Project target) {
    version v0;

    copySRCFiles() = "$target/src/**/*.*.java" : "$source/src/**/*.*.java" {
        RHS.copy(LHS);
    }

    main(Project source, Configuration conf, Project target) = : {
        copySRCFiles();
        velocity("$target/**/*.*.java", conf);
    }
}
```
Modelling data processing pipelines

1. Define elements as configuration types
2. Define specific configuration as values (development time)
Variability Model

(Type Model)

Legend

- Configurable Element
- Module
- Refinement
- Reference
- +,0..1Cardinality

- Hardware
  - Server
  - CoProcessor

- Algorithms
  - Algorithm

- Data Management
  - DataSource
  - DataSink

- Families
  - Family

- InnerNode
  - +

- ProcessorNode
  - +

- PipelineNode
  - +

- Pipeline
  - +

- Source
  - +

- Sink
  - +

- StoreNode
  - +

- Infrastructure
  - +

- Produktlinien
```java
project QM {

typedef Tuples setOf(Tuple);

// omitted: Server, CoProcessor, Tuple

compound Algorithm {
    String name;
    Tuples input;
    Tuples output;
}

typedef Algorithms setOf(refTo(Algorithm));

// omitted: DataSource, DataSink

compound Family {
    String name;
    Algorithms members;
    Tuples input;
    Tuples output;
    members->forAll(m|input == m.input
                         and output == m.output);
    members.size() > 0;
}

abstract compound PipelineNode {
    String name;
    Tuples input;
    Tuples output;
}

abstract compound InnerNode refines PipelineNode {
}

project QMCfg {

import QM;

// omitted: resources, algorithms

Source nFinancialSource = {
    name = "FinancialDataSource",
    source = refBy(nPreprocessor),
    next = {refBy(nCorrelation)}
};

FamilyNode nPreprocessor = {
    name = "Preprocessor",
    family = refBy(fPreprocessor),
    next = {refBy(nCorrelation)}
};

FamilyNode nCorrelation = {
    name = "FinancialCorrelation",
    family = refBy(fCorrelation),
    next = {refBy(nSink)}
};

Sink nSink = {
    name = "Sink",
    sink = refBy(pipSink)
};

Pipeline pip = {
    name="Example pipeline",
    sources={refBy(nFinancialSource,
              refBy(nTwitterSource)}
};
```
Domain-Specific Modelling
Modelling and Implementation with EASy-Producer

Application

- Domain-Specific Modeler
- Variability Model
- Code Generation
- Domain-Specific Infrastructure

Large parts of implementation “for free” (reusing EASy-producer)
- Powerful, but adequate modeling language
- Reasoner
- Transformation support

- Model with 9 pipelines
- Validation: 250ms
- Code generation
  - 4 Minutes
  - 30 KLOC in 195 artefacts
- Integration of algorithms
Summary
Summary-EASy-Producer

- Full support for typical product line problems
  - Interactively
  - Primarily as DSL (program your product line)

- Goals
  - Expressiveness
  - Possible to incrementally adopt
  - Representation: close to programming
  - Powerful reasoning and analysis

- Ecosystem extensions
  - (Partial) instantiation support
  - Composition
  - Openness & Modularization

- Has been applied to industrial problems
  ➔ but we are open to cooperate on more evaluations…
Summary IVML

- Very expressive approach
  - Expressiveness over analyzability
  - Can “simulate” feature models, but not restricted to this
  - Comparable to Ecore

- Representation
  - Similar to programming (ease of transition)
  - Includes concepts from OCL (constraints)
  - Constraints first class entities
  - Annotations: full expressiveness

- Reasoning
  - Very efficient forward reasoner
  - Aware of multi-step reasoning
  - Default logic (freezing)

- Ecosystem extensions
  - Interfaces
  - Modules
Summary VIL

• Configurable transformation language
• Language is a transformation language
  – Rule-based
  – Extensible
    • In various ways
    • Extend wrt. transformation operations
    • Extend wrt. artefacts
• May recur to inherited models

Summary VTL (optional)

• Template language
• Especially for artefact creation
Material

- EASy-Producer web page
- EASy-Producer release and documentation page
  - http://projects.sse.uni-hildesheim.de/easy/
- EASy-Producer on the SSE github page
  - http://ssehub.github.io/