Software Product Line Engineering
Variability Representation with Feature Models

Jessie Carbonnel

Université de Montpellier, LIRMM & CNRS

15 November, 2016
Software product line engineering

"Software product line engineering (SPLE) refers to software engineering methods, tools and techniques for creating a **collection of similar software systems** from a **shared set of software assets** using a **common means of production**."


Software product line

"A **software product line (SPL)** is a set of software-intensive systems that share a common, managed set of features satisfying the specific needs of a **particular market segment or mission** and that are developed from a **common set of core assets** in a **prescribed way**."

Software product line engineering

Development paradigm to efficiently create and manage a collection of related software systems

→ Opposed to single system development

Application of mass customization in the software engineering domain

3 important concepts

- Similar software systems
- Sharing software assets
- In a prescribed way
Similar software systems

- Software systems from a **same domain** ...
  → Security, management, e-commerce, operating system, ...

- ... satisfying a **specific need** ...
  - ✖ A schedule management software & a supplies management software
  - ✓ Two antivirus softwares

- ... and sharing **commonalities**
  → Code, requirement, architecture...
Sharing software assets

4 software systems and their common set of assets:

- **Terminology**: Asset / feature / functionality
  - an important characteristic defined by domain experts to distinguish systems from one another
  - functional and non-functional aspects of a system

- **Different levels of granularity**
  - from low level code chunks to high level software functionalities
Asset's levels of abstraction

Different levels of abstraction of the shared assets

*Example*: A collection of e-commerce applications

- **High level** functionalities understandable by *final users*
  - payment methods, basket, newsletter, wishlist, ...

- **Low level** methods/algorithms implemented by *developers*
  - paypal authentication, connection to databases, form validation, ...

Why?

⇒ The basket functionality is implemented by several assets of lower level, but it is hidden from the user for understandability sake

⇒ A low level asset can be used in different high level functionalities (e.g., database connection)
Sharing software assets

Commonalities / variabilities

4 software systems and their common set of assets:

- The *green* asset is present in all softwares
- The *blue* and *orange* asset are shared by several softwares
- The *purple* asset is specific to the fourth software
**In a prescribed way**

- Documentation of what is common and what varies between the software systems

  → Defining the way assets vary/interact in the software systems

*Example.* A collection of e-commerce applications

  → All e-commerce application have a *catalog*
  → Some can optionally have a *wishlist* or a *newsletter* functionality
  → The *wishlist* functionality requires a *user account* management

⇒ **Organisation** of the set of assets in a *generic architecture*
Generic architecture

→ The generic architecture permit to describe several related software systems depending on a set of assets

→ The set of software systems comply with the generic architecture

Why ?

⇒ **Factorisation and exploitation** of common assets

⇒ **Delimits the scope of a software family**
⇒ Permits to derive several different software systems from the generic architecture
Domain engineering

Software product line engineering - phase 1

- **Domain engineering**
  - Domain analysis
  - Domain representation
  - Domain implementation
  - Development FOR reuse
Application engineering

Software product line engineering - phase 2

- **Application engineering**
  - → Product selection
  - → Product derivation
  - ⇒ Development BY reuse
Software selection and software derivation

**Software selection / configuration**

- A user specifies its requirements = **configures** the architecture
  - designate a product configuration
  - which has to comply with the architecture

**Software derivation**

- Implementation of the designated configuration
  - Leads to **(semi-)automated source code generation**
Benefits

What are the benefits of software product line engineering?

- Improved productivity by as much as 10×
- Increased quality by as much as 10×
- Decreased cost by as much as 60%
- Decreased labor needs by as much as 87%
- Decreased time to market (to field, to launch) by as much as 98%
- Ability to move into new markets in months, not years

Benefits
Variability representation

Central point of SPLE

Modelisation of the common parts and variants contained in the software systems

= **variability** of the software product line
Variability models

Software product line variability representation

⇒ Variability models

Basis for SPLE operations and SPL management:

- Software selection
- Software derivation
- Product line evolution
- Information retrieval
- ...

How to represent the variability of a software product line?
Several variability modelling approaches exist in the literature.

Two prevalent ones

- Decision modelling
- Feature modelling
Decision modelling

- List of possible decisions a user can make
- Focus on product selection/derivation

<table>
<thead>
<tr>
<th>decision name</th>
<th>description</th>
<th>type</th>
<th>Range</th>
<th>cardinality/constraint</th>
<th>visible/relevant if</th>
</tr>
</thead>
<tbody>
<tr>
<td>GSM_Protocol_1900</td>
<td>Support GSM 1900 protocol?</td>
<td>Boolean</td>
<td>true</td>
<td>false</td>
<td></td>
</tr>
<tr>
<td>Audio_Formats</td>
<td>Which audio formats shall be supported?</td>
<td>Enum</td>
<td>WAV</td>
<td>MP3</td>
<td>1:2</td>
</tr>
<tr>
<td>Camera</td>
<td>Support for taking photos?</td>
<td>Boolean</td>
<td>true</td>
<td>false</td>
<td></td>
</tr>
<tr>
<td>Camera_Resolution</td>
<td>Required camera resolution?</td>
<td>Enum</td>
<td>2.1MP</td>
<td>3.1MP</td>
<td>5MP</td>
</tr>
<tr>
<td>MP3_Recording</td>
<td>Support for recording MP3 audio?</td>
<td>Boolean</td>
<td>true</td>
<td>false</td>
<td>If Selected Audio_Formats, MP3 = true</td>
</tr>
</tbody>
</table>

GSM_Protocol_1900: one of (GSM_1900, NO_GSM_1900) {indicates whether support for making and receiving calls using GSM 1900 is available}
Audio: list of (WAV, MP3) {indicates the types of supported audio formats}
Camera: composed of
  Presence: one of (Camera, NO_Camera) {indicates whether camera support is available}
  Resolution: one of (2.1MP, 3.1MP, 5MP) {resolution of the camera}
MP3_Recording: one of (MP3, NO_MP3) {indicates whether MP3 recording is available}

Constraints
Resolution is available only if Presence has the value Camera
MP3_Recording requires that also MP3 Audio is supported

Czarnecki, Krzysztof, et al. “Cool features and tough decisions : a comparison of variability modeling approaches.”
Feature modelling (most prevalent one)

→ Distinguishable characteristics, dependencies
→ Focus on domain representation

How to model SPL variability in terms of features?
Feature models
Feature models: family of visual description languages

→ permit to describe a finite set of features and dependencies between them

⇒ depict a finite set of valid combinations of features = configurations

(1 configuration = 1 software system of the family)
Feature tree

→ structure hierarchically the set of features in a tree = **feature tree**

- root feature = name of the modelised system
- (top to bottom) from most generalised features to most specialised one
- describe the system in several level of increasing details
- express **refinement relationships**
Software selection

Start from the root feature
- select feature from more generalised to more specialised ones (graph search)
- while respecting the expressed constraints

Optional

Mandatory

Requires

Exclude

→ Software selection
→ 2 types of constraints

- graphical constraints expressed in the feature tree
  - Xor
  - Or
  - Optional
  - Mandatory

- textual constraints which cannot be expressed in the tree: **cross-tree constraints**
  - Requires
  - Exclude

**Feature tree:**

```
  e_commerce
     /   
  catalog payment_method basket
    /   
  grid list credit_card check
```

**Cross-tree constraints:**

- payment_method → basket
- basket → payment_method
Graphical constraints (1)

→ 4 different “graphical constraints” (1/2)

- between a parent feature and its child feature:

  ![Diagram showing graphical constraints]

  - Optional: if the parent feature is selected, the child feature can be selected, or not
  - Mandatory: if the parent feature is selected, the child feature is necessarily selected
→ **4 different “graphical constraints”** (2/2)

- between a parent feature and several of its child features:

  - **Or-group**: if the parent feature is selected, at least one feature involved in the group has to be selected
  - **Xor-group**: if the parent feature is selected, exactly one feature involved in the group has to be selected
Textual constraints (2)

→ 2 different “textual constraints”

- between two independent features:
  - Requires: if the premise is selected, the conclusion is also selected
  - Exclude: the two features are mutually exclusive

![Diagram showing relationships between e-commerce, catalog, payment_method, basket, grid, list, credit_card, check, and their relationships]
→ an e-commerce application necessarily possesses a catalog
→ this catalog can be displayed in a grid or in a list, but not both
→ it can eventually possess payment methods (credit card, check, or both)
→ it can also optionally have a basket
→ if the basket feature is selected, the application must possess at least one payment method (and conversely)
Feature model semantics

2 types of semantics

→ what do feature models define?

- a configuration semantics / logical semantics
- an ontological semantics
Configuration/logical semantics

Configuration semantics:

→ The list of valid configurations depicted by the feature model

1 \{e\_commerce, catalog, grid\}
2 \{e\_commerce, catalog, list\}
3 \{e\_commerce, catalog, grid, payment\_method, credit\_card, basket\}
4 \{e\_commerce, grid, payment\_method, check, basket\}
5 \{e\_commerce, catalog, grid, payment\_method, credit\_card, check, basket\}
6 \{e\_commerce, catalog, list, payment\_method, credit\_card, basket\}
7 \{e\_commerce, list, payment\_method, check, basket\}
8 \{e\_commerce, catalog, list, payment\_method, credit\_card, check, basket\}
Ontological semantics

→ *Domain knowledge depicted by the feature model*

**Example.**

- grid and list refine catalog
- catalog and payment_method are two independent features
- credit_card and check are independent but can coexist
Non-canonical representation

→ Same configuration semantics, but different ontological semantics
→ describe different domain knowledge
⇒ Non-canonical representation
"understandable" and "compact" way to express variability
- combinatorial explosion of the possible software variants
- potentially large number of represented software systems
(Example: Linux SPL = 41 features = $2 \times 10^7$ configurations)
⇒ enlarge the selection of products offered