Software Intensive Systems

- Importance of non-functional properties
  - distributed reactive systems, parallel & asynchronous
  - quality of service: security, reliability, latency, performance...

- Variations in functional aspects
  - notion of product lines (space, time)

Example: Nokia
1.1 billions phones in circulation
Hundreds of networks
~10,000 variants of software
Software Reuse

- drastically decrease cost of software development and maintenance
- increase quality of software
- reuse of existing software is one of the most promising approaches
- construct applications by composing reusable software pieces

(cost $\Rightarrow$ TTM $\Rightarrow$ staff)

Two approaches to reuse

- Opportunistic:
  - the software engineer reuses pieces of software that fit the current problem and adds them to the software.

- Planned:
  - the organization puts explicit effort in developing reusable artifacts that provide the ‘right’ abstractions, ‘right’ level of variability and that fit into an higher level structure.

- Opportunistic reuse does not work in practice
  - As in automotive and other industries, build on the notion of Product Line
Software Product Lines

- A software product line consists of:
  - product line architecture
  - set of reusable components
  - set of products, where each product has
    » product architecture derived from PLA
    » instantiated and configured components
    » product-specific code
Many Issues Around SPLs

- **Assets [Jacobsen]:**
  - architecture
  - components
  - systems
- **Views [SEI]:**
  - business
  - organization
  - process
  - technology
- **Lifecycle [Bosch]:**
  - development
  - usage
  - evolution

Technical Issues in SPLs:
Managing variability
### Initiating a Product Line

<table>
<thead>
<tr>
<th></th>
<th>Evolutionary</th>
<th>Revolutionary</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Existing product line</strong></td>
<td>Develop vision for PLA. Develop one comp. at a time by evolving existing components</td>
<td>Develop PLA, components and products based on requirement super set.</td>
</tr>
<tr>
<td><strong>New product line</strong></td>
<td>PLA and components evolve based on requirements posed by new PL members.</td>
<td>PLA and components developed to match requirements of all PL members.</td>
</tr>
</tbody>
</table>

### Evolving Existing Products

- **advantages**
  - reduced risk due to
    - small up front investment
    - early return on first shared components
  - relatively small effect on production schedule

- **disadvantages**
  - larger total investment
Replace Existing Products

- **advantages**
  - shorter conversion time
  - smaller total investment

- **disadvantages**
  - higher risk
  - negative effect on production schedule

role of hardware and mechanical parts

Applicability of SPL concepts

- context: consultancy company or IT department performing projects with partially overlapping requirements
- define common architecture and common components/subsystems
- develop (slightly) more general components in normal projects
- increase the variability and generality of the component in subsequent projects
- **balance investment and risk!**
SPL Overview

- development
  - design product line architecture
  - develop software components
- deployment
  - develop members of product line
  - Ideally, "just" configure and specialize the product line
- evolution
  - evolve all assets in the product line
  - Software configuration management

Model Driven Engineering for SPL

- How to model PL with the UML?
  - Variability in the UML class diagrams
- Constraints on PL models
  - The use of OCL (Object Constraints Language ) to specify PL constraints
- The PL derivation
  - The decision model
  - The PL derivation as the UML models transformation
The 3 Dimensions of Software Configuration Management:

- The Variant dimension
  - Handle environmental differences
- The Revision dimension
  - Evolution over time
- The Concurrent Activities dimension
  - Many developers are authorized to modify the same configuration item
- Even with the help of sophisticated tools, the complexity might be daunting
  - Try to simplify it by reifying the variants of an OO system

Variants in Software Systems

- Hardware Level
  - \( V_i = 16 \)
- Heterogeneous Distributed Systems
  - \( V_p = 4 \)
- Peculiarities in Target Operating System
- Compiler Differences
- Range of Products
  - \( V_n = 8 \)
- User Preferences for GUI
  - \( V_g = 5 \)
- Internationalization
  - \( V_l = 24 \)

\[ \text{Number of Variants} = V_p * V_n * V_g * 2^{V_i+V_l-2} \]
- That's 43,980,465,111,040 possible variants
Traditional Approaches

- Patch the executable
- Device Drivers
  - source level, link time, boot time, on demand at runtime
- Static Configuration Table
- Conditional Compilation / Runtime Tests

```c
if (language == "french") {
    #ifdef MSW
    io_puts(0,"Bonjour",7);
    #elif TEXT
    printf("Bonjour\n");
    #endif
} else {
    #ifdef MSW
    io_puts(0,"Hello",5);
    #elif TEXT
    printf("Hello\n");
    #endif
}
```

- Static and Dynamic configuration information intermingled
- Hard to change your mind on what should be static or dynamic...

Basic Idea

- Abstract the Intent
  - `io.write_line(language.hello)`
- Rely on Dynamic Binding for the Details
  - Don’t care now for static/dynamic distinction

```
IO
  \- MSW
  \- TEXT

Language
  \- French
  \- Basic
```

- Uncouple the variations from the selection process
  - Automatically derive a product using OCL 2 meta-model transformation
The Mercure PL as example

- A family of SMDS\cite{Jezequel96} (Switched Multi-Megabits Data Service) servers.
- Delivering, forwarding, and relaying messages from and to a set of network interfaces.
Variability in UML class diagrams

Abstraction
Inheritance, Abstract Factory

Parameterization
UML class templates

Optionality
UML extensions mechanisms (Stereotype « Optional »)

Alternatives
Optional + Constraint
Product Line Constraints

- **Generic Constraints (GC):** Constraints for ALL product line architectures (architecture coherence).

- **Specific Constraints (SC):** Constraints for a SPECIFIC product line architecture (relationships between specific elements).

**GC as OCL meta-level constraints**

- $M_2$ (UML meta-model level)
  - extended UML Meta-model for PL
  - Defined on
  - Meta level OCL Constraints

- $M_1$ (UML model level)
  - SPL architecture 1
  - SPL architecture 2
  - SPL architecture N
  - Evaluated on
**Examples**

Dependency constraint

context Dependency

\[
\text{inv : self.supplier} \rightarrow \exists (S: \text{ModelElement} \ S.\text{isStereotyped('optional')}) \\
\text{implies self.client} \rightarrow \forall (C: \text{ModelElement} | \\
C.\text{isStereotyped('optional')} )
\]

**SC as OCL meta-level constraints**

M₂ (UML meta-model level) → M₁ (UML model level)

- Extended UML Meta-model for PL
- Defined on PL architecture
- Associated to Product N architecture
- Evaluated on SPL specific constraints
Examples

- **Presence constraint**
  
  context `Model_Management::Model`
  
  inv:
  
  ```
  self.presenceClass('ENGINE1') implies
  self.presenceClass('NETDRIVER1')
  ```

- **Mutual exclusion constraint**
  
  context `Model_Management::Model`
  
  inv:
  
  ```
  (self.presenceClass('GUI1') implies not
  (self.presenceClass('LANGUAGE_CAT2')) and
  (self.presenceClass('LANGUAGE_CAT2') implies not
  self.presenceClass('GUI1'))
  ```

Product Line Derivation

<table>
<thead>
<tr>
<th>Products</th>
<th>VPs</th>
<th>GUI</th>
<th>Language</th>
<th>Manager</th>
<th>NetDriver</th>
<th>Engine</th>
</tr>
</thead>
<tbody>
<tr>
<td>CustomMercure</td>
<td>GUI1, GUI2</td>
<td>Language 2-1</td>
<td>Manager1</td>
<td>NetDriver1</td>
<td>Engine1</td>
<td></td>
</tr>
<tr>
<td>MiniMercure</td>
<td>GUI1</td>
<td>Language 1</td>
<td>Manager1</td>
<td>NetDriver1</td>
<td>Engine1</td>
<td></td>
</tr>
<tr>
<td>FullMercure</td>
<td>all</td>
<td>all</td>
<td>all</td>
<td>all</td>
<td>all</td>
<td></td>
</tr>
</tbody>
</table>

GUI 1 mutual-exclusion LANG Cat 2
The decision model

- The Abstract Factory Design Pattern
  - [Gamma et al 95]

```
$\text{Mercure}_{\text{Factory}}$
\begin{itemize}
  \item new$\_\text{gui}()$ : GUI
  \item new$\_\text{language}()$ : Language
  \item new$\_\text{network\_manager}()$ : Manager
  \item new$\_\text{netdriver}()$ : Net Driver
  \item new$\_\text{engine}()$ : Engine
\end{itemize}
```

Dynamic Configuration

```
\text{aMercure}_{\text{Concrete\_Factory}} \quad \text{Mercure} \\
\text{read()} \quad \text{aDynamic\_Configuration}
```

```
\text{get\_gui\_type()} \\
\text{get\_language\_types()} \\
\text{get\_manager\_type()} \\
\text{get\_netdriver\_types()} \\
\text{get\_engine\_type()} \\
\text{run()} \\
```
The PL derivation

- Manipulate the UML meta-model to automatically derive a product model using a Model Transformation Language (MTL)

DeriveProductLine

**Input:** PL_model: Model

aConcreteFactory: Class

pre: -- check Generic Constraints on PL_model

**Output:** Product_model: Model

post: -- check Specific Constraints on the PL_model

---

Model Transformation

- By limiting the range of variants available from a given Concrete Factory:
  - The transformer may know the set of living classes
    » special case of Partial Evaluation
  - Generate a specialized model for the product
    » When only one living class for an abstract varying part:
      ■ Dynamic binding replaced with direct call (and even inlining)
    » When only a few living classes
      ■ Dynamic binding replaced by if then ... else
      ■ Implemented in e.g., GNU SmallEiffel
  - All static configuration issues kept encapsulated and do not pollute the model
The PL derivation

- The Variants selection:
  - Using operation factory stereotypes

- The Model specialization:
  - Removes all optional classes which have not been selected

- The Model optimization:
  - Deletes unused factories, Optimize inheritance

DerivePL (PL_model: Model, aConcreteFactory: Class) : Model {
    selectedVariants: Set;
    Result := clone(PL_model);
    selectedVariants := getSelectedVariants(aConcreteFactory);
    // Model specialization
    for each optional class C in PL_model do
        if (the class name of C not in selectedVariants)
            and (names of all subclasses of C not in selectedVariants)
            then
                delete the class C from Result;
        endif
    done
    // Model optimization
    replace abstract classes with only one subclass S by S
    delete all other factories
}
Code size

Type inference score for MiniMercure is 99.7%

Runtime Performances

Dead code removal  Automatic static binding
A collection of events rather than that collection of messages.

- A combined SD: refers to a set of interactions and composes them by means of operators:
  - **Sequence** (seq): weak sequential composition.
  - **Alternative** (alt): choice between interaction operands.
  - **Loop** (loop): iteration of an interaction.
- Extended with operators to model variability
  - **Optional, variation, virtual**...
**STEP 1: Behavioral derivation**

<table>
<thead>
<tr>
<th>Product</th>
<th>Decision model instance (DMI)</th>
</tr>
</thead>
<tbody>
<tr>
<td>BS1</td>
<td>DM1 = {(settingLimit, TRUE), (settingCurrency, FALSE), (withdrawAccount, 1), (fromEuro, FALSE), (toEuro, FALSE)}</td>
</tr>
<tr>
<td>BS2</td>
<td>DM2 = {(settingLimit, FALSE), (settingCurrency, FALSE), (withdrawAccount, 2), (fromEuro, FALSE), (toEuro, FALSE)}</td>
</tr>
<tr>
<td>BS3</td>
<td>DM3 = {(settingLimit, FALSE), (settingCurrency, FALSE), (withdrawAccount, 2), (fromEuro, TRUE), (toEuro, TRUE)}</td>
</tr>
<tr>
<td>BS4</td>
<td>DM4 = {(settingLimit, TRUE), (settingCurrency, TRUE), (withdrawAccount, 1), (fromEuro, TRUE), (toEuro, TRUE)}</td>
</tr>
</tbody>
</table>

RESD = [[PL-RESD]] DMI
Behavioral derivation rules

- [[optional name [ E ]]]_DMI = \begin{cases} 
E \text{ IF } (\text{name, TRUE}) \in DMI \\
E_\emptyset \text{ IF } (\text{name, FALSE}) \in DMI 
\end{cases}

$E_\emptyset$ is the empty SD: neutral element for seq, alt; idempotent for loop

- [[variation name [ E1, E2, ]]]_DMI = E_i \text{ IF } (\text{name, i}) \in DMI

- [[virtual name [ E ]]]_DMI = Ereff \text{ IF } (\text{name, Ereff}) \in DMI, E \text{ ELSE }

DM2 = ((settingLimit, FALSE), (settingCurrency, FALSE), (withdrawAccount, 2), 
(fromEuro, FALSE), (toEuro, FALSE))

$E_{BS2} = [[E_{BS2}]]_{DM2}$

$E_{BS2} = \text{loop } (\text{Deposit alt }): \text{(CreateAccount seq)}$

$\begin{aligned}
&\text{seq } (\text{CreateAccountOk seq E_\emptyset seq CreateAccountFailed) alt (WithdrawWithoutLimit seq (WithdrawOk alt WithdrawFailed)) alt (E_\emptyset) alt (E_\emptyset))}
\end{aligned}$

$E_\emptyset$ is the empty SD: neutral element for seq, alt; idempotent for loop

$\Rightarrow$ expression reduction
\[ E_{BS2} = \text{loop } (Deposit \ \text{alt} \ (\text{CreateAccount seq } (\text{CreateAccountOk }) \ \text{alt} \ \text{CreateAccountFailed}) \ \text{alt} \ (\text{WithdrawWithoutLimit seq } (\text{WithdrawOk alt} \ \text{WithdrawFailed}))) \]

**Step 1 result**: One expression (RESD) for each product

### STEP2: UML Sequence Diagrams (SDs):

=> Statecharts

- **Inter-object view**: Many objects, one example.

- **Intra-object view**: Single object, a complete behavior.

**Related work**:
- Kriss et al [UML 98],
- Koskinies et al [IEEE Software 98]
- Whittle et al [ICSE 00],
- Mäkinen et al [ICSE 01] etc.
From combined SDs

Statecharts operators

- 1) Sequence (seqₕ)

<table>
<thead>
<tr>
<th>SDs</th>
<th>STs</th>
</tr>
</thead>
<tbody>
<tr>
<td>seq</td>
<td>seqₕ</td>
</tr>
<tr>
<td>alt</td>
<td>altₕ</td>
</tr>
<tr>
<td>loop</td>
<td>loopₕ</td>
</tr>
</tbody>
</table>
Statecharts operators (contd.)

2) Alternative \((\text{alt}_a)\)

![Statechart Diagram]

Statecharts operators (contd.)

3) Iteration \((\text{loop}_a)\)

![Statechart Diagram]
From combined SDs (contd.)

\[ E = \text{loop} \ (\text{UserArrives} \ \text{seq} \ (\text{loop} \ (\text{EnterPassword} \ \text{seq} \ \text{BadPassword}) \ \text{seq} \ \text{EnterPassword} \ \text{seq} \ (\text{BadAccount} \ \text{alt} \ \text{UserCancel}) \ \text{alt} \ \text{UserCancel})) \]

\[ E = \text{loop}_5 \ (\text{P(UserArrives, ATM)} \ \text{seq}_s \ (\text{loop}_5 \ (\text{P(EnterPassword, ATM)} \ \text{seq}_s \ \text{P(BadPassword, ATM)}) \ \text{seq}_s \ \text{P(EnterPassword, ATM)} \ \text{seq}_s \ (\text{P(BadAccount, ATM)} \ \text{alt}_5 \ \text{P(UserCancel, ATM)}) \ \text{alt}_5 \ \text{P(UserCancel, ATM))}) \]

---

Result: Bank StateChart for BS2

![Bank StateChart for BS2]
Result: Bank StateChart for BS4

Conclusion & Perspectives
Conclusion

- Handling variability in SPL is complex
  - Exponential number of configurations
- Use models (& aspects) to manage complexity
  - Abstract away from #IFDEF & diff
- Executable Meta-Modeling to Automate Product Derivation from SPL Models
  - Both from Static & Dynamic aspects

Perspectives on SPL Research

- Composition of models
  - New ways of composing software from modeling elements
    » at both model and meta-model levels
    » Unifying MDE, AOSD, SPL, Generative Programming...
  - Composing models at runtime: dynamic adaptation
    » FP7 STREP: DIVA (03/2008-03-2011)