A formal approach for fostering component reuse and managing software change

Abderrahman MOKNI,
Marianne HUCHARD, Christelle URTADO,
Sylvain VAUTTIER et Huaxi (Yulin) ZHANG
Context and problematic

- Component-based software engineering (separation of concerns, software in the large, complex systems, …)
  - Reduce development time and costs,
  - Reduce maintenance costs (usually takes 60%).

- Challenges:
  - A better reuse,
  - A better evolution handling (unanticipated changes),
  - A better software architecture documentation.

=> Need for formal mechanisms to improve software reuse and automatically handle architectural changes.
Outline

- The reuse approach
- The formal approach
- Intra-level rules
- Inter-level rules
- Evolution rules and process
- Conclusion and perspectives
Definitions

- **Software Architecture**: blueprint of the software system (design decisions, structure, interactions).

- **Components**: encapsulates data and functionalities.

- **Interfaces**: abstraction of component services (required and provided).

- **Connections** (connectors): connect components to each other.
The reuse approach [Zhang 2010]

<table>
<thead>
<tr>
<th>Component design for reuse</th>
<th>Component-based software design by reuse</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lifecycle step</td>
<td>Lifecycle step</td>
</tr>
<tr>
<td>Production</td>
<td>Production</td>
</tr>
<tr>
<td>Component development and</td>
<td>System requirement analysis</td>
</tr>
<tr>
<td>documentation</td>
<td></td>
</tr>
<tr>
<td>Component code storage and</td>
<td>Functional &amp; non-functional requirements</td>
</tr>
<tr>
<td>indexation</td>
<td></td>
</tr>
<tr>
<td>Component code &amp; models</td>
<td>Architecture specification</td>
</tr>
<tr>
<td>Component repository</td>
<td>Component search</td>
</tr>
<tr>
<td></td>
<td>Architecture configuration</td>
</tr>
<tr>
<td></td>
<td>Component instantiate</td>
</tr>
<tr>
<td></td>
<td>Instantiated component assembly</td>
</tr>
<tr>
<td></td>
<td>Instantiated assembly description &amp; software</td>
</tr>
</tbody>
</table>

Caption:
- Uses
- Produces
- Precedes
Architecture levels

• Specification level
  • Architecture as intended by the architect and conform to user requirements
  • **Component roles**: partial and ideal description of software components
  • Used to guide the search for concrete components.

• Configuration level
  • A concrete implementation of the software
  • **Concrete component classes** selected from repositories

• Assembly level
  • Description of the architecture at runtime
  • Parameterized **component instances**
Running example: Home automation software

Interface types and their signatures:

```java
ILight{
    void switchOn();
    void switchOff();
}

ITime{
    int getTime();
}

ITherm{
    int getTemp();
}

ICon{
    void setCondMode(CondMode mode);
    CondMode getCondMode();
}
```

Caption

- **Component role**
- **Provided interface**
- **Required interface**
Configuration level

Interface types and their signatures:

\[\begin{align*}
  \text{IPower} & \{} \\
  & \quad \text{void switchOn();} \\
  & \quad \text{void switchOff();} \\
  \text{Intensity} & \{} \\
  & \quad \text{void setIntensityLevel(int intensity);} \\
  & \quad \text{int getIntensityLevel();} \\
  \text{IClock} & \{} \\
  & \quad \text{void setDateTime(int time, Date date);} \\
  & \quad \text{int getTime();} \\
  & \quad \text{Date getDate();} \\
  \text{ITherm} & \{} \\
  & \quad \text{int getTemp();}
\end{align*}\]

Caption

<table>
<thead>
<tr>
<th>Component class</th>
<th>Provided interface</th>
<th>Required interface</th>
<th>Delegation link</th>
</tr>
</thead>
</table>
Assembly level

Diagram showing components and interfaces:
- airConditioner
- clock1
- lamp1
- lamp2
- androidOrchestrator1

Caption:
- Component instance
- Provided interface
- Required interface
Outline

- The reuse approach
- The formal approach
- Intra-level rules
- Inter-level rules
- Evolution rules and process
- Conclusion and perspectives
The formal approach

- Formalization based on set theory and first-order logic
  - B modeling language
- Generic concepts: architectures, components, interfaces, signatures, …
- Specific concepts: specification, configuration, component roles, component classes, …
- Invariants.
MACHINE Arch_concepts
INCLUDES Basic_concepts
SETS ARCHS; COMPS; COMP_NAMES
VARIABLES
architecture, arch_components, arch_connections, component, comp_name, connection, comp_interfaces, client, server
arch_clients, arch_servers
INVARIANT
// A component has a name and a set of interfaces */
component ⊆ COMPS ∧
comp_name ∈ component → COMP_NAMES ∧
comp_interfaces ∈ component ↦ P(interface) ∧
// A client (resp. server) is a couple of a component and an interface */
client ∈ component ↔ interface ∧
server ∈ component ↔ interface ∧
// A connection is a relation between a client and a server */
connection ∈ client ↔ server ∧
// An architecture has a set of components and connections */
architecture ⊆ ARCHS ∧
arch_components ∈ architecture → P(component) ∧
arch_connections ∈ architecture → P(connection) ∧
// Arch_clients (resp. arch_servers) lists the connected clients (resp. servers)
within an architecture */
arch_clients ∈ architecture → P(client) ∧
arch_servers ∈ architecture → P(server) ∧

Specific B notations:
↔: relation    ↦: injection    P(<set>): powerset of <set>
Outline

- The reuse approach
- The formal approach
- Intra-level rules
- Inter-level rules
- Evolution rules and process
- Conclusion and perspectives
Intra-level rules

- Substitutability rules
  - Syntactic definition of signatures (name, types, parameters),
  - Interface typing with respect to covariance and contravariance rules,
  - Interface substitutability,
  - Component substitutability.

- Compatibility rules
  - Between interfaces,
  - Between components.
Example

Clock

- setDateTime
- setDateformat

ClockV2

- setTime
- setDateformat

ISetting

- setTime
- setDateformat

IInfo

- getTime
- getDate
- getDateFormat

ILocation

- getLocation

IlocationAndGMT

- getLocation
- getGMT

ILanguage

- getLanguageInfo

Legend

- Component substitutability
- Interface substitutability
- Interface subtyping
- Inheritance

DateFormat

SimpleDateFormat
Consistency and completeness

- Based on the compatibility between interfaces

- Consistency:
  - Correct connections between components,
  - Connected architectural graph (no isolated components).

- Completeness (internal):
  - All required interfaces are connected
Outline

- The reuse approach
- The formal approach
- Intra-level rules
- Inter-level rules
- Evolution rules and process
- Conclusion and perspectives
Inter-level rules

Abstract architecture specification

Concrete architecture configuration

Instantiated architecture assembly

Component role

Component class

Component instance

<<implements>>

<<instantiates>>

<<realizes>>
The realization rule

- Many-to-many relation,
- A component class may <<realize>> several roles at once,
- A roles may be realized by composing several component classes.

=> more flexibility while searching for implementation solutions.
The realization rule

\[
\text{realizes} \in \text{compClass} \leftrightarrow \text{compRole} \land \\
\forall (CL, CR). (CL \in \text{compClass} \land CR \in \text{compRole} \\
\Rightarrow \\
((CL, CR) \in \text{realizes} \\
\iff \\
\exists CT. (CT \in \text{compType} \land (CT, CR) \in \text{matches} \land \\
(CL, CT) \in \text{class_implements}))
\]
Example

Interface typing:

```
I1  I2  I3  J1  J2
```

Signature matching:

```
sig1 <-> sig1'
sig2 <-> sig2'
sig3 <-> sig3'
sig4 <-> sig4'
sigA <-> sigA'
sigB <-> sigB'
```
Coherence between a specification and a configuration

- A configuration <<implements>> a specification if and only if:
  - Every role in the specification is realized by a component class in the configuration,
  - All the specified services in the specification are met in the configuration.
Coherence between assembly and configuration

- <<Instantiates>> is a many-to-one relation.

- An assembly is an instantiation of a configuration iff:
  - Each component class is instantiated at least once,
  - Each instance in the assembly is an instantiation of a component class in the configuration.
Outline

• The reuse approach
• The formal approach
• Intra-level rules
• Inter-level rules
• Evolution rules and process
• Conclusion and perspectives
Architecture-centric evolution

- A process to evolve software system by modifying its architecture.

- Issues:
  - Inconsistencies: (name, interface, behavior, …)
  - Architecture erosion: integrating architectural changes that violate higher level preconditions.
  - Architecture drift: integrating architectural changes that are not considered by the higher abstraction level.
Evolution rules

- Change operations guarded by preconditions,
- Three main operations: addition, deletion and substitution,
- Defined at:
  - Specification level to update user requirements,
  - Configuration level to update software implementation,
  - Assembly level to change software dynamically.
- Change can be initiated externally or triggered by the evolution manager.
Example of evolution rule
(Instance addition)

deployInstance(asm, inst, class, state) =

PRE
asm ∈ assembly ∧ class ∈ compClass ∧ /* The instance is a valid instantiation of an existing component class*/
inst ∈ compInstance ∧ class = comp_instantiates(inst) ∧ inst ∉ asm_components(asm) ∧ /* The state given to the instance is a valid value assignment to the attributes of the instantiated component class*/
state ∈ P (attribute_value) ∧ card(state) = card(class_attributes(class)) ∧ /* The maximum number of allowed instances of the given component class is not already reached*/

THEN
/*initial and current state initialisation*/
initiation_state(inst) := state ||
current_state(inst) := state ||
/*updating the number of instances and the assembly architecture*/

END;

SATTToSE 2014
11/07/2014
Outline

• The reuse approach
• The formal approach
• Intra-level rules
• Inter-level rules
• Evolution rules and process
• Conclusion and perspectives
Conclusion

- A formal model for multi-level software architectures,
- Intra-level rules to ensure architecture consistency,
- Coherence rules between architecture descriptions,
- Evolution rules to automatically handle software change and avoid architectural mismatches.
Perspectives

- Implement an evolution management environment within an eclipse-based platform,
- Study and manage software architecture versioning,
- Implementing a case study.