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
  - Reduce development 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

The reuse approach [Zhang 2010]
Dedal 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

```plaintext
Interface types and their signatures:

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

ITime{
    int getTime();
}

ITherm{
    int getTemp();
}

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

<table>
<thead>
<tr>
<th>Light</th>
<th>Time</th>
<th>Thermometer</th>
<th>CoolerHeater</th>
<th>HomeOrc estator</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image" alt="Light" /></td>
<td><img src="image" alt="Time" /></td>
<td><img src="image" alt="Thermometer" /></td>
<td><img src="image" alt="CoolerHeater" /></td>
<td><img src="image" alt="HomeOrc estator" /></td>
</tr>
</tbody>
</table>

Caption:  
- Component role  
- Provided interface  
- Required interface
Configuration level

Interface types and their signatures:

```java
interface IPower {
    void switchOn();
    void switchOff();
}

interface IIntensity {
    void setIntensityLevel(int intensity);
    int getIntensityLevel();
}

interface IClock {
    void setDateTime(int time, Date date);
    int getTime();
    Date getDate();
}

interface ITherm {
    int getTemp();
}
```

Caption

- Component class
- Provided interface
- Required interface
- Delegation link

Assembly level

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.
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

```
ClockV2
| setTime(Time time) |
| setDateFormat(SimpleDateFormat format) |

Clock
| setTime(Time time) |
| setDateFormat() |

ILocation
| getLocation() : Point |

ILocationAndGMT
| getLocation() : Point |
| getGMT() : TimeZone |

IInfo
| getTime() : Time |
| getDate() : Date |
| getDateFormat() : DateFormat |

ILanguage
| getLanguageInfo() : String |

ISetting
| setTime(Time time) |
| setDateFormat() |

ISettingV2
| setTime(Time time) |
| setDateFormat() |
```

Légende

- Component substitutability
- Interface substitutability
- Interface subtyping
- Inheritance
Example: substitutability rule

\[ \text{comp}\_\text{substitution} \in \text{component} \leftrightarrow \text{component} \land \forall(C\_\text{sup}, C\_\text{sub}). \]
\[ (C\_\text{sup} \in \text{component} \land C\_\text{sub} \in \text{component} \land C\_\text{sup} \neq C\_\text{sub} \Rightarrow (C\_\text{sub} \in \text{comp}\_\text{substitution}[[C\_\text{sup}]]) \]
\[ \exists(inj1, inj2). \]
\[ (inj1 \in \text{providedInterfaces}(C\_\text{sup}) \overset{inj1}{\rightarrow} \text{providedInterfaces}(C\_\text{sub}) \land \forall(int). \]
\[ (int \in \text{interface} \land int \in \text{providedInterfaces}(C\_\text{sup}) \Rightarrow inj1(int) \in \text{int}\_\text{substitution}[[int]]) \land \]
\[ inj2 \in \text{requiredInterfaces}(C\_\text{sub}) \overset{inj2}{\rightarrow} \text{requiredInterfaces}(C\_\text{sup}) \land \forall(int). \]
\[ (int \in \text{interface} \land int \in \text{requiredInterfaces}(C\_\text{sub}) \land \Rightarrow int \in \text{int}\_\text{substitution}[[inj2(int)])]) \]

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
The realization rule

- Many-to-many relation,
- A component class may \(<\text{realize}>\) several roles at once,
- A roles may be realized by composing several component classes.

=> more flexibility while searching for implementation solutions.

The realization rule

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

- \texttt{<<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
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 manger.

Example of evolution rule (role addition)

```
addRole(spec, newRole) =
  PRE
  initialization = TRUE ∧ spec ∈ specification ∧ newRole ∈ compRole ∧ newRole ∉ spec_components(spec) ∧ changeOperation = none ∧
  ∀ er.(er ∈ compRole ∧ er ∈ spec_components(spec) ⇒ comp_name(er) ≠ comp_name(newRole))
  THEN
  spec_components(spec) := spec_components(spec) ∪ \{newRole\} ||
  spec_servers(spec) := spec_servers(spec) ∪ servers(newRole) ||
  spec_clients(spec) := spec_clients(spec) ∪ clients(newRole) ||
  selectedRole := newRole ||
  selectedArch := spec ||
  changeOperation := ADDITION
  END;
```
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.
Publications list

- Modélisation et vérification formelles en B des architectures logicielles à trois niveaux. CIEL 2014.
- A three-level formal model for software architecture evolution. SATToSE 2014.
- Fostering component reuse: automating the coherence verification of multi-level architecture descriptions. Submitted to ICSEA 2014.