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.

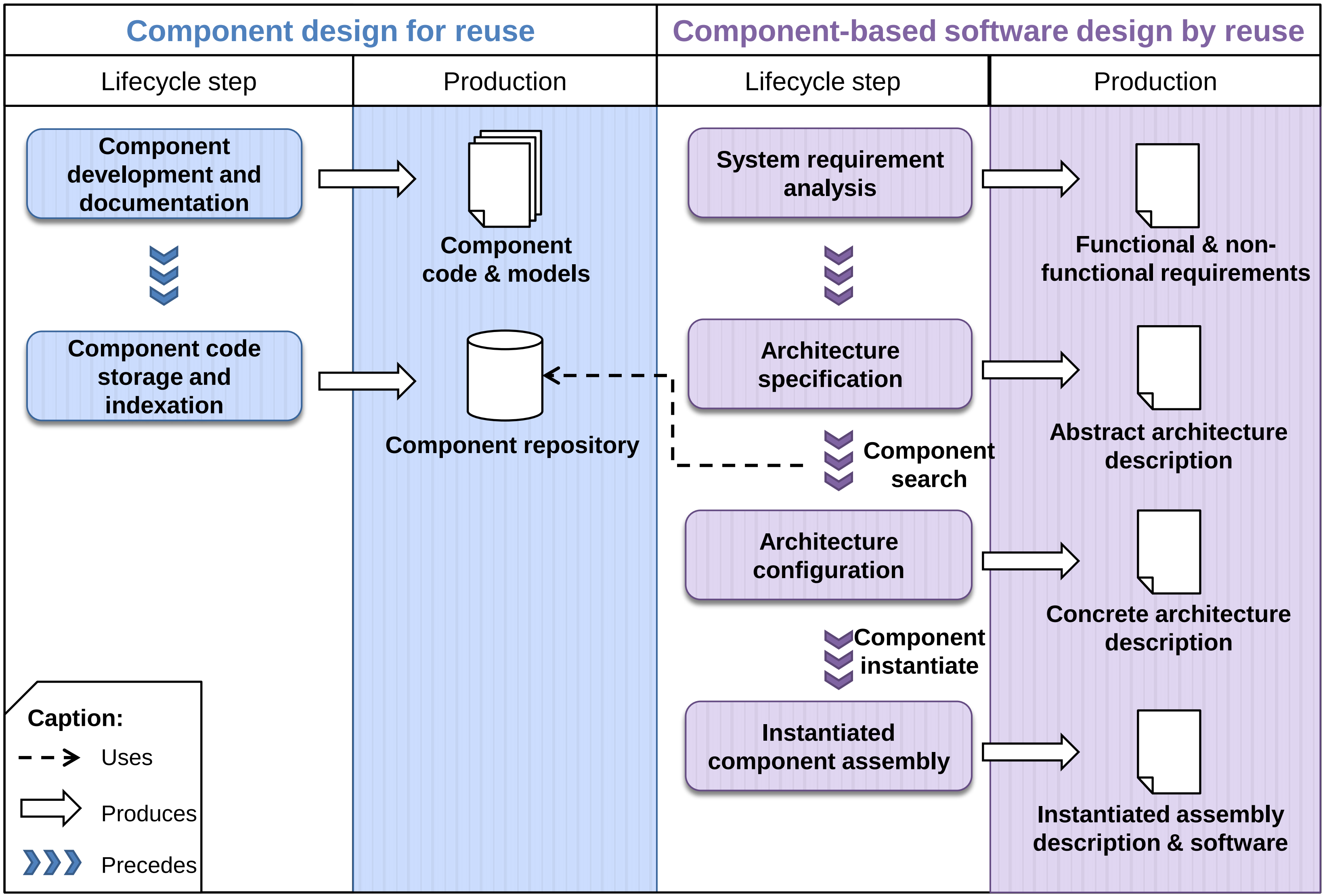
Outline

- 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, strcucture, 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]



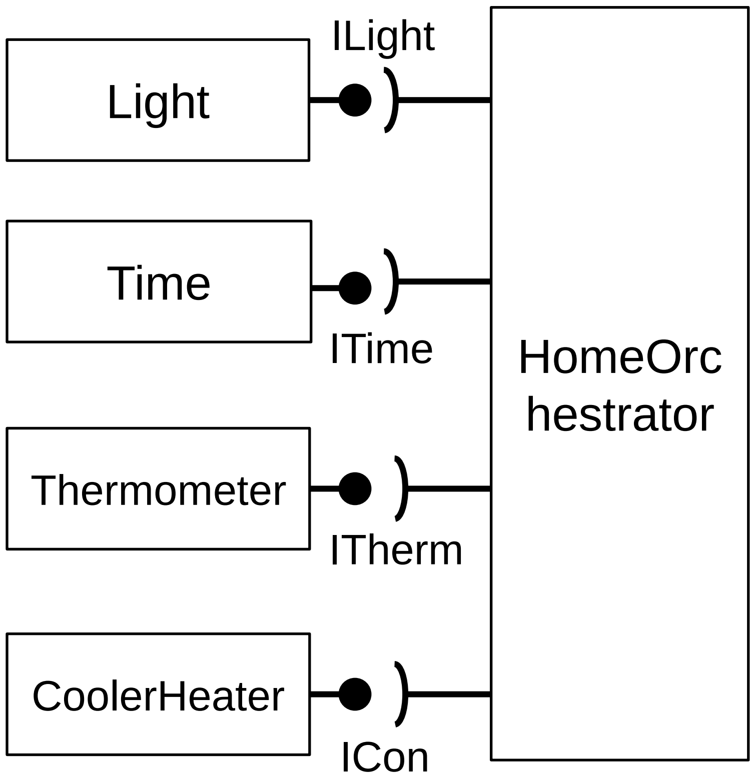
Architecture levels

- 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

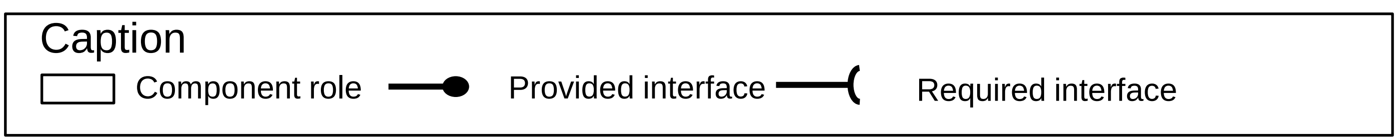
- Assembly level
  - Description of the architecture at runtime
  - Parameterized **component instances**

Running example : Home automation software

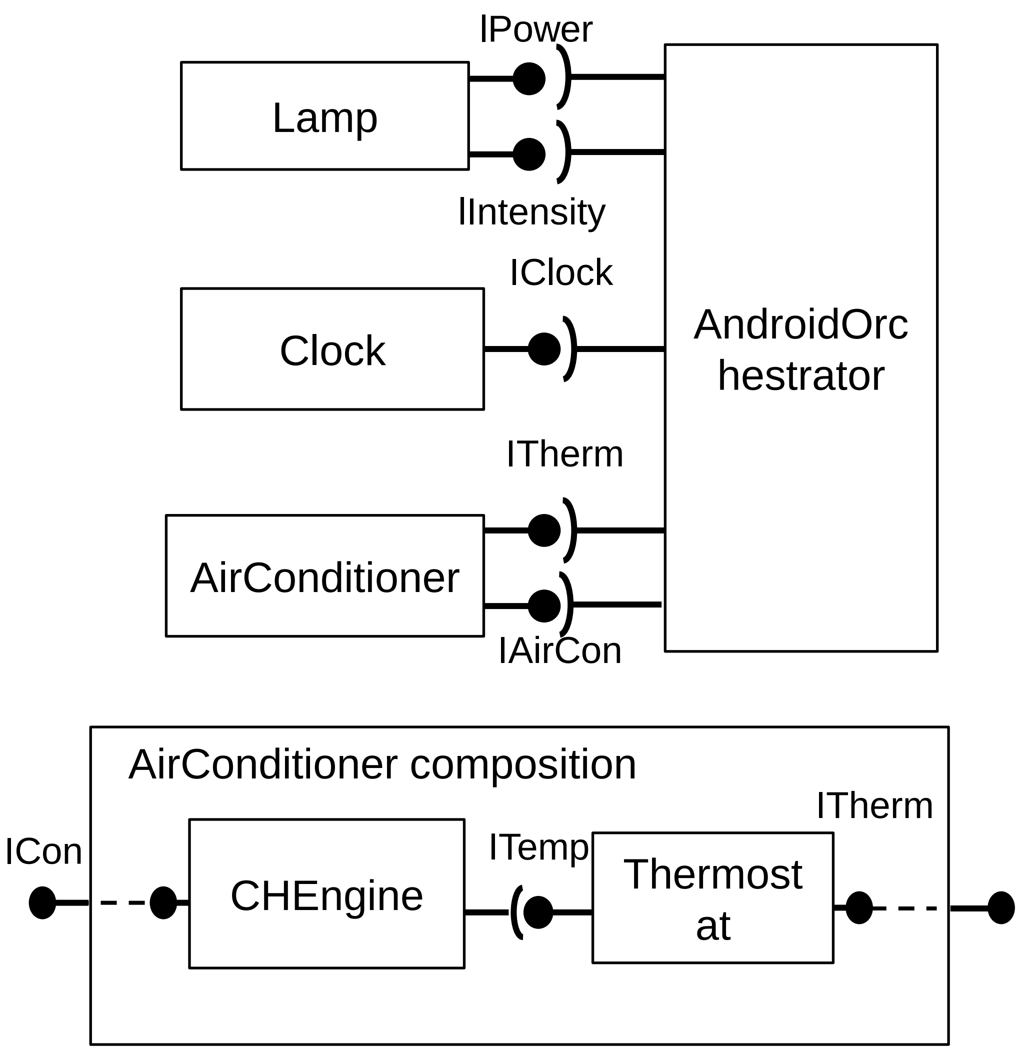


Interface types and their signatures:

ILight{
  void switchOn();
  void switchOff();
}
ITime{
  int getTime();
}
ITherm{
  int getTemp();
}
ICon{
  void setCondMode(CondMode mode);
  CondMode getCondMode();
}



Configuration level

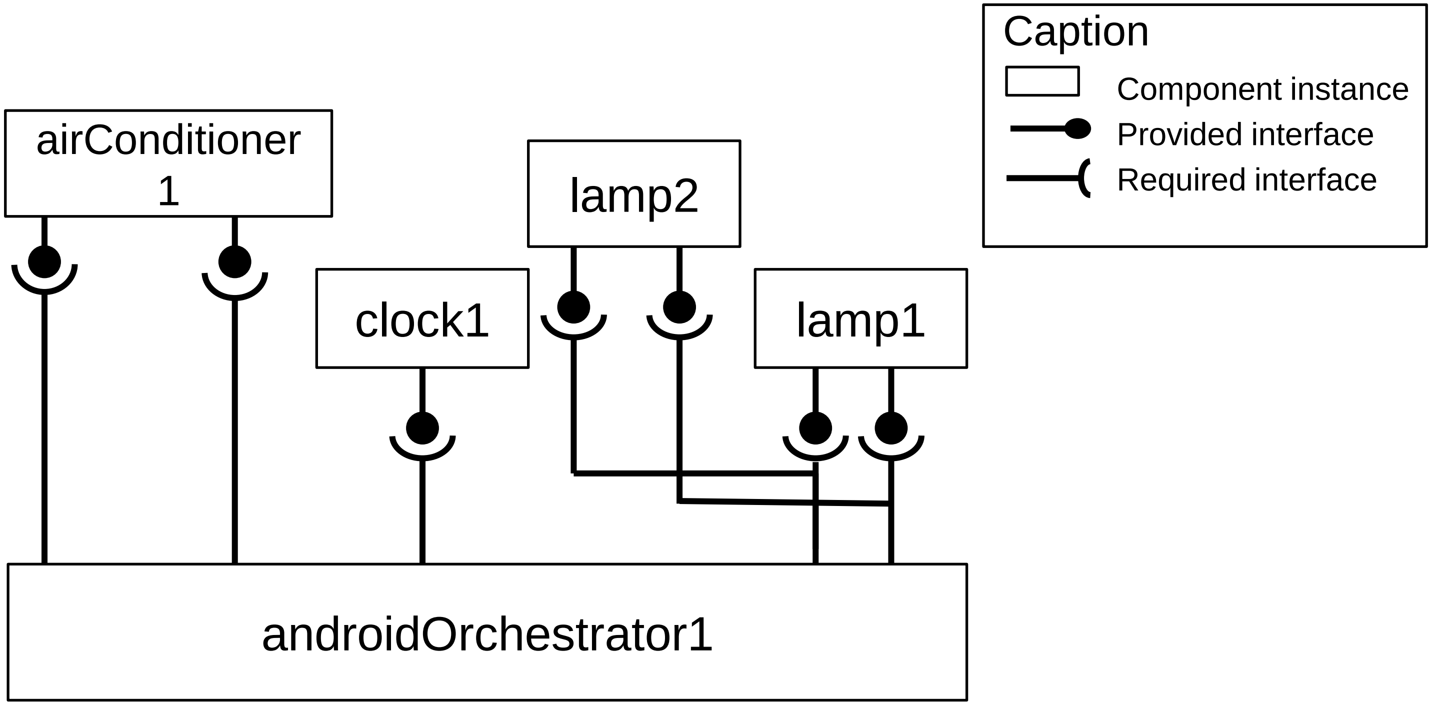


Interface types and their signatures:

IPower{
void switchOn();
void switchOff();
}
IIntensity{
void setIntensityLevel(int intensity);
int getIntensityLevel();
}
IClock{
void setDateTime(int time, Date date);
int getTime();
Date getDate();
}
ITherm{
int getTemp();
}
…
…

Caption
Component class — Provided interface — Required interface — Delegation link

Assembly level



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.

Example

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 */
$component \subseteq COMPS \land$
$comp\_name \in component \rightarrow COMP\_NAMES \land$
$comp\_interfaces \in component \rightarrowtail \mathcal{P}(interface) \land$
/* A client (resp. server)is a couple of a component and an interface */
$client \in component \leftrightarrow interface \land$
$server \in component \leftrightarrow interface \land$
/* A connection is a relation between a client and a server */
$connection \in client \leftrightarrow server \land$
/* An architecture has a set of components and connections */
$architecture \subseteq ARCHS \land$
$arch\_components \in architecture \rightarrow \mathcal{P}(component) \land$
$arch\_connections \in architecture \rightarrow \mathcal{P}(connection)$
/* Arch_clients (resp. arch_servers) lists the connected clients(reps. servers) within an architecture */
$arch\_clients \in architecture \rightarrow \mathcal{P}\ (client) \land$
$arch\_servers \in architecture \rightarrow \mathcal{P}\ (server)$

Specific B notations:
$\leftrightarrow$: relation $\rightarrowtail$: injection $\mathcal{P}$(<set>): powerset of <set>

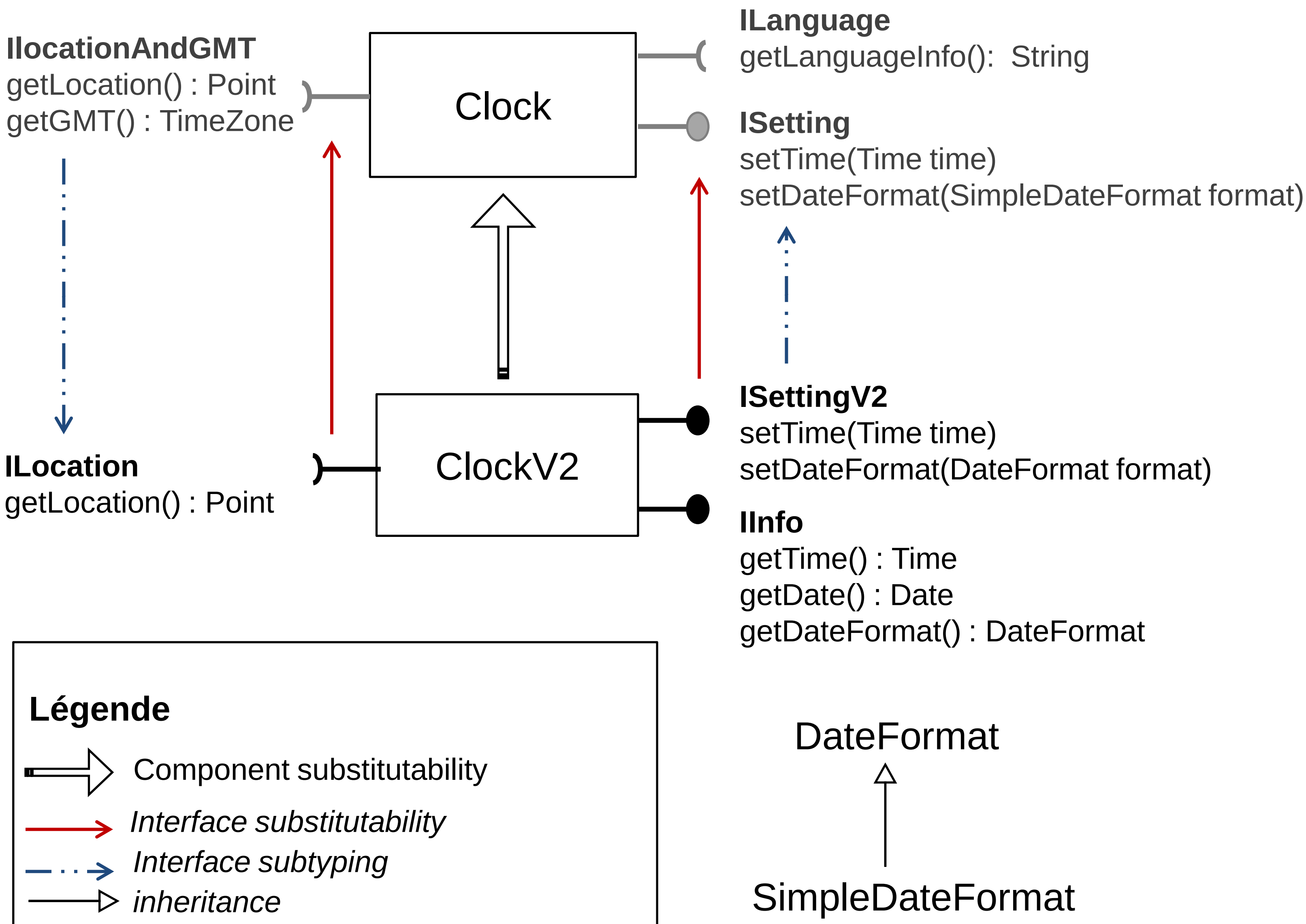
Outline

- 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


IlocationAndGMT
getLocation() : Point
getGMT() : TimeZone
Clock
ILanguage
getLanguageInfo(): String
ISetting
setTime(Time time)
setDateFormat(SimpleDateFormat format)
ILocation
getLocation() : Point
ClockV2
ISettingV2
setTime(Time time)
setDateFormat(DateFormat format)
IInfo
getTime() : Time
getDate() : Date
getDateFormat() : DateFormat
Légende
Component substitutability
Interface substitutability
Interface subtyping
inheritance
DateFormat
SimpleDateFormat

Consistency and completeness

- Based on the compatibility between interfaces

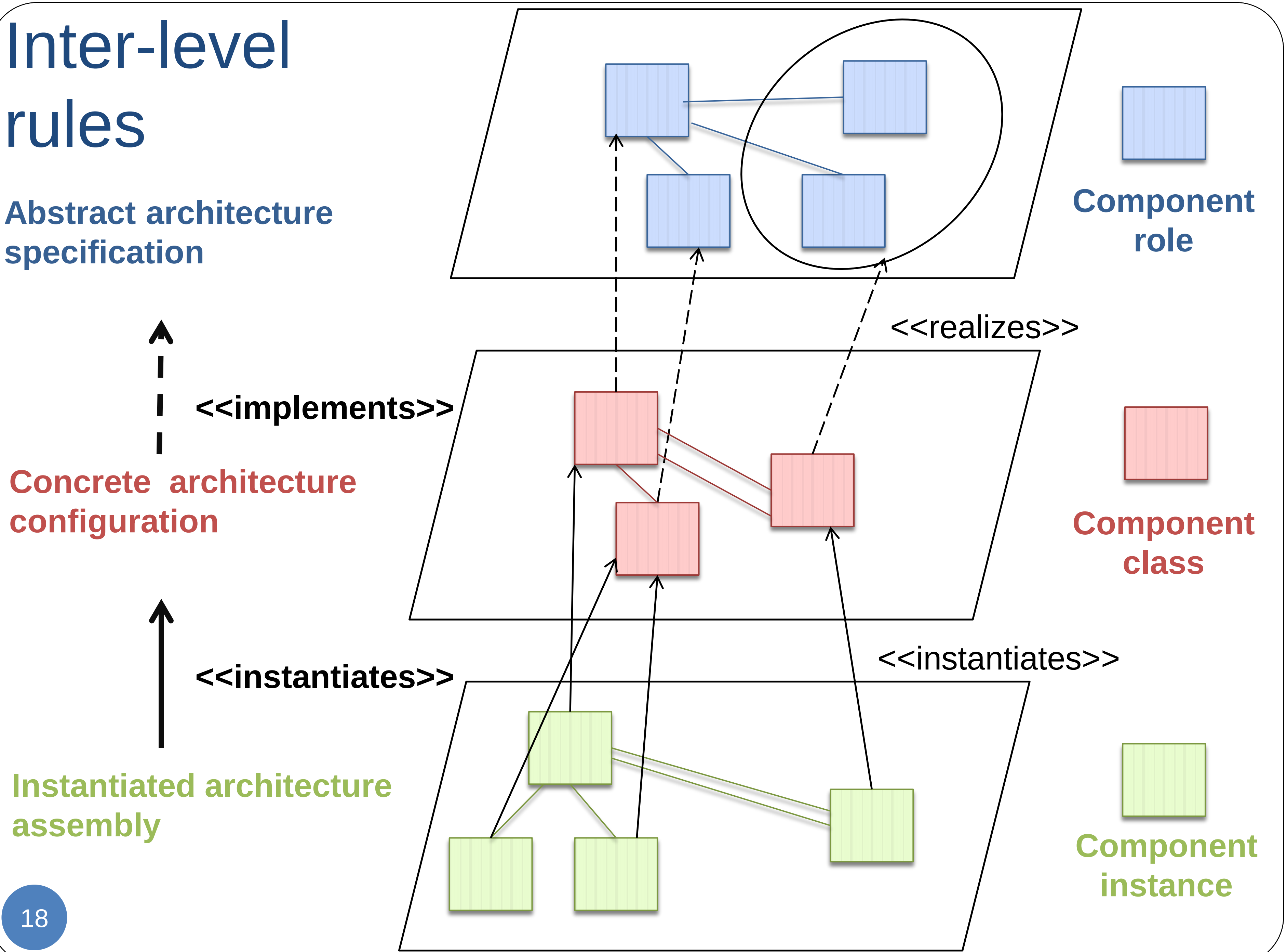
- Consistency:
  - Correct connections between components,
  - Connected architectural graph (no isolated components).

- 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



The realization rule

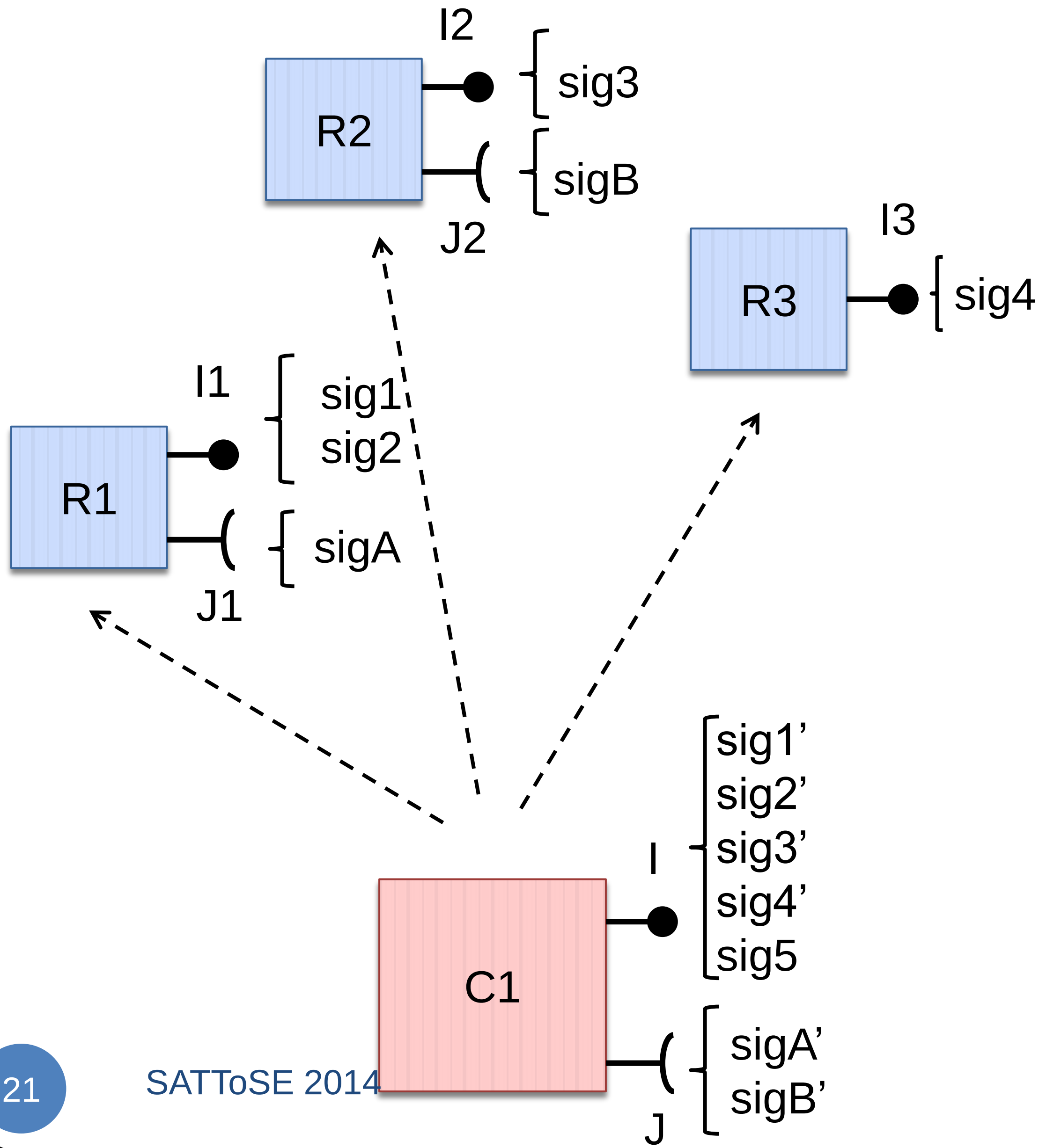
- Many-to-many relation,
- A component class may <<realize>> several roles at once,
- A roles may be realized by composing several component classes.

=> more flexibility while searching for implementation solutions.

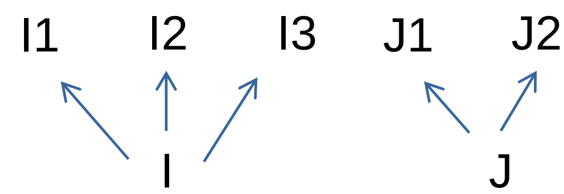
The realization rule

$$
\begin{aligned}
&realizes \in compClass \leftrightarrow compRole \wedge \\
&\forall (CL, CR).(CL \in compClass \wedge CR \in compRole \\
&\Rightarrow \\
&\quad ((CL, CR) \in realizes \\
&\qquad \Leftrightarrow \\
&\quad \exists\, CT.(CT \in compType \wedge (CT, CR) \in matches \wedge \\
&\quad\ (CL, CT) \in class\_implements)) \\
&)
\end{aligned}
$$

Example



Interface typing:



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 manger.

Example of evolution rule (Instance addition)

deployInstance(asm, $inst$, $class$, $state$) =

PRE

$asm \in assembly \wedge class \in compClass \wedge$

/* The instance is a valid instantiation of an existing component class*/

$inst \in compInstance \wedge class = comp\_instantiates(inst) \wedge inst \notin assm\_components(asm) \wedge$

/* The state given to the instance is a valid value assignment to the attributes of the instantiated component class*/

$state \in \mathcal{P}\ (attribute\_value) \wedge \mathbf{card}(state) = \mathbf{card}(class\_attributes(class)) \wedge$

/* The maximum nhumber of allowed instances of the given component class is not already reached*/

$nb\_instances(class) < max\_instances(class)$

THEN

/*initial and current state initialisation*/

$\mathbf{initiation\_state}(inst) := state \parallel$

$\mathbf{current\_state}(inst) := state \parallel$

/*updating the number of instances and the assembly architecture*/

$\mathbf{nb\_instances}(class) := nb\_instances(class) + 1 \parallel$

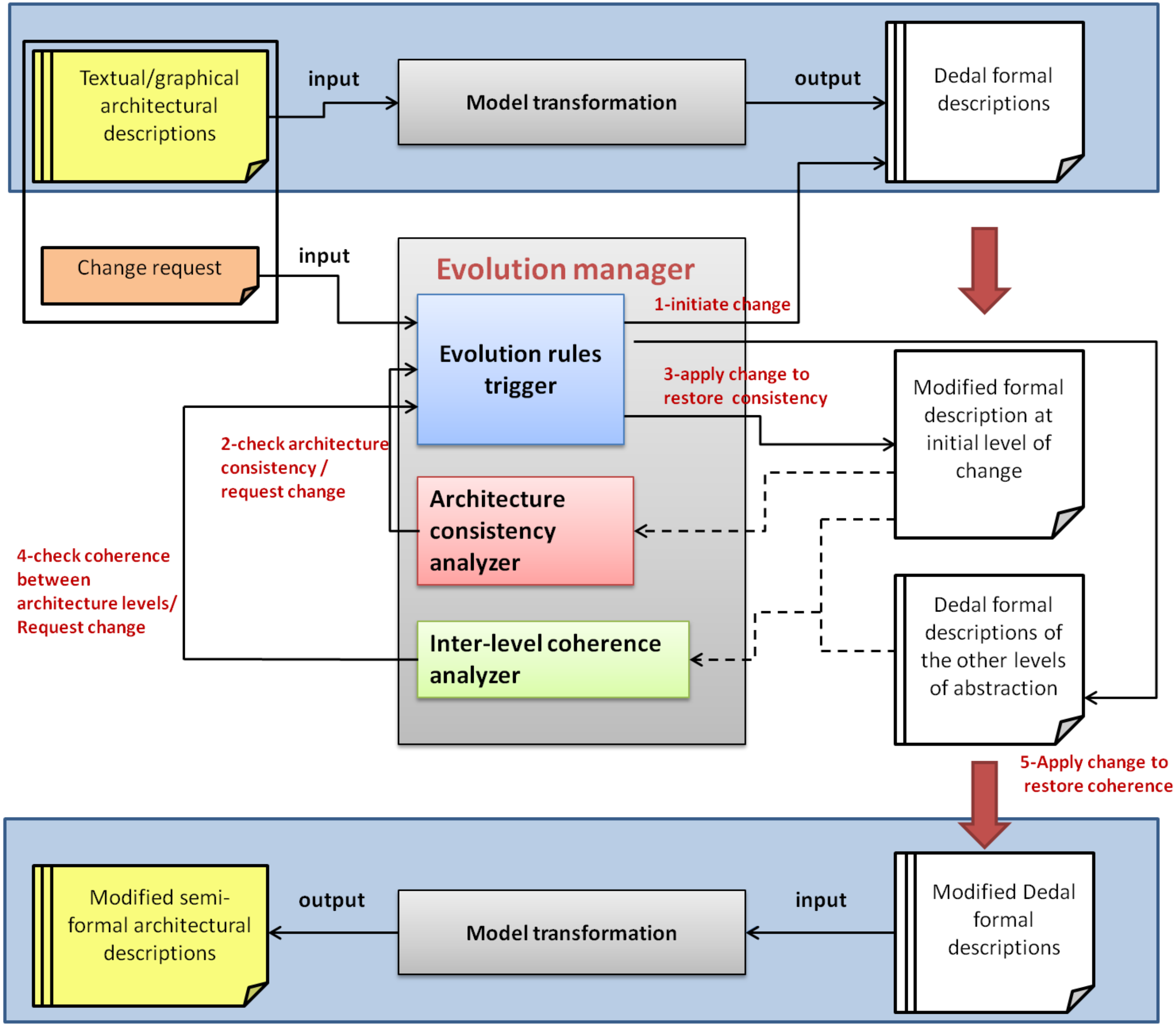
$\mathbf{assm\_components}(asm) := assm\_components(asm) \cup \{inst\} \parallel$

$\mathbf{assm\_servers}(asm) := assm\_servers(asm) \cup servers(inst) \parallel$

$\mathbf{assm\_clients}(asm) := assm\_clients(asm) \cup clients(inst)$

END;

Evolution process



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 environement within an eclipse-based platform,
- Study and manage software architecture versioning,
- Implementing a case study.