Automatic Translation of Architecture Constraint Specifications into Components

Sahar Kallel, Bastien Tramoni, **Chouki Tibermacine**, Christophe Dony & Ahmed Hadj Kacem.

LIRMM, CNRS and Montpellier University, France

Outline

Context

- Architecture constraints are predicates that formalize design rules (instantiation of patterns, styles, ...)
- They are used to complement some architecture descriptions with invariants to enforce design rules (during evolution)
- They are checked by analyzing architecture descriptions

Context -Ctd

Architecture vs. Functional Constraints:

- Checked by analyzing: static architecture descriptions vs. states of running components
- Used in: Evolution Assistance vs. Design by Contract
- Specified in the context of a: metamodel vs. model
- Examples (in OCL):
  - Functional Constraint:

```
--Employees must have the legal age to work
context Employee inv: self.age >= 16
```

  - Meta-level (∼Architecture) Constraint: part of UML spec.

```
--Only binary associations can be aggregations
context Association inv: self.memberEnd
->exists(aggregation <> Aggregation::none)
implies self.memberEnd->size()=2
```

Problem Statement

- Many architecture constraints have been formalized for design, architecture and SOA patterns
- These constraints are "gross" unstructured specifications
- They do not offer enough reusability and parametrization
- They are composed of many independent parts that have their own semantics, and which can be reused with other architecture descriptions
- In the past, we proposed a component model for the specification of architecture constraints: CLACS

Problem Statement

- Many architecture constraints have been formalized for design, architecture and SOA patterns
- These constraints are “gross” unstructured specifications
- They do not offer enough reusability and parametrization
- They are composed of many independent parts that have their own semantics, and which can be reused with other architecture descriptions
- In the past, we proposed a component model for the specification of architecture constraints: CLACS

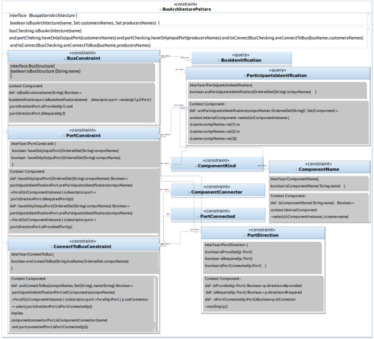
There is no automated way to translate all existing architecture constraints into CLACS

Goals of this Work

- **General goal:** Make existing constraint specifications *reusable*, *customizable* and *composable* entities by automatically translating them into components
- In this work:
  - We specified, implemented and experimented an automatic translation process
  - This process takes into account a concrete language for architecture constraint specification: OCL
  - It generates CLACS components

Input and Output of the Process


```
context Component inv :
let bus:Component
= self.realization.realizingClassifier
->select(c:Classifier | c.oclIsKindOf(Component)
 and c.oclAsType(Component).name='esbImpl')
customers : Set(Component)
= self.realization.realizingClassifier
->select(c:Classifier | c.oclIsKindOf(Component)
 and (c.oclAsType(Component).name='cust1'
  or c.oclAsType(Component).name='cust2'
  or c.oclAsType(Component).name='cust3'))
producers : Set(Component)
= self.realization.realizingClassifier
->select(c:Classifier | c.oclIsKindOf(Component)
 and (c.oclAsType(Component).name= 'prod1'
or c.oclAsType(Component).name='prod2'
or c.oclAsType(Component).name='prod3'))
in
-- The bus should have at least one input port
-- and one output port
bus.ownedPort->exists(p1,p2:Port|
   p1.provided->notEmpty() and p2.required->notEmpty())
and
--Customers should have output ports only
customers->forAll(c:Component|
  c.ownedPort->forAll(required->notEmpty()
    and provided->isEmpty()))
and
--Customers should be connected to the bus only
customers->forAll(com:Component|
  com.port->forAll(p:Port|p.end->notEmpty()
    implies
      self.ownedConnector ->exists(con:Connector |
         bus.ownedPort->exists( pb:Port|
           con.end.role->includes(pb)) and
             con.end->includes(p.end))))
and
--Producers should have input ports only
producers->forAll(c:Component|
  c.ownedPort->forAll(provided->notEmpty()
    and required->isEmpty()))
and
--Producers should be connected to the bus only
producers->forAll(com:Component|
  com.port->forAll(p:Port|p.end->notEmpty()
    implies
      self.ownedConnector->exists(con:Connector|
        bus.ownedPort->exists( pb:Port|
          con.end.role->includes(pb)) and
            con.end->includes(p.end))))
```



A Translation Process composed of two Steps

- **Refactoring of Constraints:** Decomposing and transforming “gross” OCL constraints into parameterized *OCL definitions*
- **Generation of CLACS Components:** Grouping and wrapping OCL definitions in constraint components

- *OCL definitions:* named and (possibly) parameterized expressions part of a model/metamodel [used by invariants]

Example:

```
context Component
def: getBusComp(busName:String): Component
=self.realization.realizingClassifier
->select(c:Classifier | c.oclIsKindOf(Component)
 and c.oclAsType(Component).name = busName)
```

Constraint Refactoring

A multi-step micro-process whose input is the constraint's AST:

- Extraction of declarations
- Decomposition of the constraint
- Redundancy Removal
- Parameterization of definitions
- Recontextualization of definitions

Result:

- A set of basic OCL definitions whose context is the meta-class Component, ready for the second step
- A set of recontextualized OCL definitions associated to different meta-classes, usable for invariant specification

Extraction of Declarations

- *Let* expressions enable to declare variables initialized with the values of repeated expressions

In this step:

- If let expressions exist, extract them from the constraint
- Declare them as OCL definitions (queries):

```
context Component
def:letCustomers():Set(Component)=self.realization
.realizingClassifier->select(c : Classifier |
c.oclIsKindOf(Component) and
(c.oclAsType(Component).name='cust1' or
 c.oclAsType(Component).name = 'cust2' or ...))
```

- Call these generated OCL definitions in their appropriate places in the constraint

Constraint Decomposition

- A recursive process where the constraint is decomposed into expressions based on logical operators
- Stopping condition: size & no logical operators
- Declare these pieces as OCL defs returning a boolean value
- Refactor the constraint for every generated def
- Example:

```
context Component
def: def1(c:Classifier):Boolean=c.oclIsKindOf(Component)
def: def2(c:Classifier):Boolean=c.oclAsType(Component).name
= 'esbImpl'
def: letBus():Component=self.realization.realizingClassifier
->select(c:Classifier|def1(c) and def2(c))
def: def3(c:Classifier):Boolean=c.oclIsKindOf(Component)
def: def4(c:Classifier):Boolean=c.oclAsType(Component).name
 = 'cust1' ...
def: letCustomers():Set(Component)=self.realization.
realizingClassifier->select(c:Classifier|def3(c) and def4(c))
...
```

Redundancy Removal

- After the decomposition step, we obtain a bag (multiset) of OCL definitions
- Remove all redundant (syntactically identical) definitions

```
context Component
def: def1(c:Classifier):Boolean=c.oclIsKindOf(Component)
...
def: def3(c:Classifier):Boolean=c.oclIsKindOf(Component)
...
```

- Refactor the constraint

Constraint Parametrization

- Create a parameter in the signature of the definition when finding a literal value of a given data type
  - Types of parameters are obtained from the AST
  - Example:

```
context Component
def: def2(c:Classifer, name:String): Boolean
= c.oclAsType(Component).name = name
```

- Identify similar definitions (by ignoring the signatures), and remove them (considered as redundant):

```
context Component
def: def17(c:Classifier, name1:String): Boolean =
c.oclAsType(Component).name = name1
def: def18(c:Classifier, name2:String): Boolean =
c.oclAsType(Component).name = name2
...
def: def4(c:Classifier,name1:String,name2:String,name3:String):
Boolean = def17(c,name1)and def18(c,name2) and def19(c,name3).
```

Constraint Parametrization -Ctd

- Optimize parameters:

```
def:def4(setofcustomers:OrderedSet(String)):Boolean=
...
```

instead of:

```
def:def4(name1:String,name2:String,
         name3:String):Boolean=
...
```

- Refactor the definition and the constraint accordingly

Component Generation

- A CLACS component is stereotyped *query* or *constraint*
- A CLACS component declares ports; each one has an interface specifying a set of query/checking operation signatures
- The generation of components is also a multi-step transformation micro-process:
  - OCL Definition (Operation) grouping
  - Metamodel migration
  - CLACS arch. description generation

Operation Grouping

- CLACS query-components embed OCL defs (not boolean)
- CLACS constraint-components embed OCL boolean defs
- Putting together the OCL definitions that check/query similar aspects (properties of the same kind of architectural elements)
- How similarity between defs is measured?
  - Sub-trees of the AST are compared
  - They should share a common root and a minimal sub-tree (obtained in a breadth-first traversal)
  - For the remaining part, an *edit distance* is measured between each pair of sub-trees ($> \theta$, a threshold)

Metamodel Migration

- Transformation of OCL expressions defined on the UML metamodel into OCL/CLACS constraints
- We specified declarative mappings between UML and CLACS metamodels
- These mappings are applied automatically by transforming in the order:

  1. navigation patterns
  2. simple navigations and attribute accesses
  3. meta-classes

CLACS Arch. Description Generation

- CLACS query and constraint component descriptors are instantiated and connected together
- These components embed the refactored architecture constraints
- This is defined in a composite constraint component descriptor
- This component can then be instantiated and connected to business components in order to check the initial constraint

Used dataset and Metric

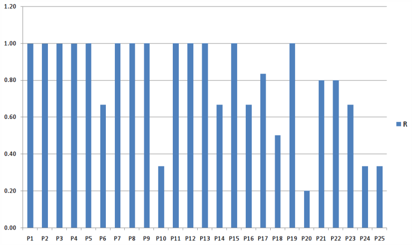
- We collected 25 architecture constraints of arch. patterns
- Average times: parsing 96ms, refactor. 5081ms, gener. 846ms
- We chose a well-known (industry-validated) metric to measure the reusability in the results of the translation process

$$C = \left(b + \left(\sum \frac{E}{n}\right) - 1\right) R + 1 \qquad (1)$$

- C: cost of software development –specification of a constraint-component
- E: the cost of developing a reusable element –a constraint-component
- n: the number of uses of this reusable element
- b: cost of integrating the reused elements into the new artifact –integration of constraint-components in a composite
- R: the proportion of reused elements

Results

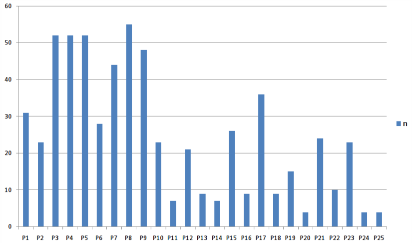
- R represents the proportion of a constraint's structure which is reused from other constraints



- R values are in the range 20-100 %
- 13 (/25) constraints have 100 % of their internal structure reused elsewhere

Results -Ctd

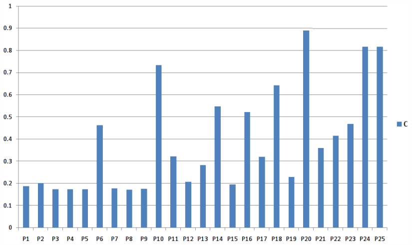
- n: number of times a constraint-component is reused in the whole set of constraints



- 6 constraint-components have a structure reused more than 50 times
- The constraint-component no 8 is reused 55 times

Results -Ctd

- Constants from the literature: b = 0.15 and E = 1.2



- C is in the range of 0.18 to 0.89
- All of the constraint-components have a cost less than 1: effective reuse in the translation results

Complementary Evaluation

- For now, we have evaluated the “quality” of the process output
- Evaluation of the translation process: “Does it produce the same/better output than a manual design with CLACS?”
- We selected 7 constraints (AST sizes: ∼500 to 2500 tokens)
- Compare the translation results with CLACS descriptions manually specified by another person (two years ago)
- Average Precision = 0.84
- The lowest precision is due to the fact that the decomposition produced a lot of small “re-useless” constraint-components
- Recall: not pertinent to measure (no false negatives)

Conclusion

- Architecture constraints are a means to enforce architecture styles, patterns or general design rules
- Catalogs of these architecture constraints have been designed
- In these catalogs, constraints are “gross” specifications, which are subject to reuse, customization and composition
- We have proposed a process for translating them automatically into components
- This enables to improve reusability of this kind of architecture “documentation” without having to manually redefine it

Perspectives

- Make the generated CLACS components checkable:
  - in the implementation stage on component-based programs
  - and at runtime on reifications of architecture descriptions

by translating them into Compo, our component-based reflective programming language

Perspectives

- Make the generated CLACS components checkable:
  - in the implementation stage on component-based programs
  - and at runtime on reifications of architecture descriptions

by translating them into Compo, our component-based reflective programming language

Thanks for your attention