Specifying Components With Compositional Patterns, LOTOS And Design By Contract

Abdelhafid ZITOUNI
Laboratory LIRE
University of Constantine
Algeria
ah_zitouni@yahoo.fr

Mahmoud BOUFAIDA
Laboratory LIRE
University of Constantine
Algeria
boufaida@hotmail.com

Lionel SEINTURIER
LIFL-INRIA ADAM
University of Lille
France
Lionel.Seinturier@lifl.fr

SEDE 2010,
JUNE, 16-18, 2010
CONTENTS

Introduction

Abstract specification of a component

- Structural contracts

Interface contracts

Behavioural contracts

Proposal  Architecture Description Language

Conclusion & future work
Introduction (1/4)

- Component-based software development:
  - Building large software systems
  - Software components.

- Component-based approaches:
  - Create, deploy software systems assembled from components.
Introduction (2/4)

Motivations

- Previously developed components.

- Behavioural and compositional conflicts among components constitute a crucial barrier.

- Contract through a formal model analyze:
  - Analyze pattern-based designs.
  - Precise criteria of comparison.
Introduction (3/4)

Contribution

- Contract-based approach:
  - Representing, instantiating and integrating design patterns, and analyzing their compositions
  - Using LOTOS as an Architecture Description Language (ADL) for formalising these aspects.
Introduction (4/4)

Contribution

✓ How to represent the instantiation, the evolution and the integration of design components, and how to find design composition errors.

✓ we present a pattern contract language that captures the structural and behavioral requirements associated with a range of patterns, as well as the system properties that are guaranteed as a result.
Background

Design Patterns

- Design patterns are a design paradigm used to solve problems that arise when developing software within a particular context.

- Capture the static and dynamic structure and collaboration among the components in a software design.

- To build software systems, a designer needs to solve many problems. Applying known design patterns to address these problems allows the designer to take advantage of expert design experience.
LOTOS

- Language of Temporal Ordering Specifications (LOTOS) is a formal description technique standardized at ISO, based on a combination of CCS [Milner] and CSP [Hoare].

- The basic idea supporting LOTOS is that systems can be specified by expressing the relations among the interactions that constitute their externally observable behaviour.

- In LOTOS, a system is seen as a process, possibly consisting of several sub-processes.
<table>
<thead>
<tr>
<th>operator</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ ]</td>
<td>either $P_1[a,b]$ or $P_2[c,d]$ depending on the environment</td>
<td>$P[a,b,c,d]=P_1[a,b] [ ] P_2[c,d]$</td>
</tr>
<tr>
<td>[ ] [ ]</td>
<td>Parallel composition without synchronization: $P_1[a,b]$ is independent from $P_2[c,d]$</td>
<td>$P[a,b,c,d]=P_1[a,b] [ ] P_2[c,d]$</td>
</tr>
<tr>
<td>[b]</td>
<td>Parallel composition with synchronization on gate b</td>
<td>$P[a,b,c,d]=P_1[a,b] [b] P_2[c,d]$</td>
</tr>
<tr>
<td>&gt;&gt;</td>
<td>Sequential composition: $P_1[a,b]$ is followed, when $P_1$ terminated, by $P_2[c,d]$;</td>
<td>$P[a,b,c,d]=P_1[a,b] &gt;&gt; P_2[c,d]$</td>
</tr>
<tr>
<td>[&gt;]</td>
<td>Disrupt: $P_1 [a, b]$ may be interrupted at any time before its termination by $P_2[c, d]$.</td>
<td>$P[a,b,c,d]=P_1[a,b] [&gt; P_2[c,d]$</td>
</tr>
<tr>
<td>;</td>
<td>Process prefixing by action a</td>
<td>a;P</td>
</tr>
</tbody>
</table>

**LOTOS operators**
CONTENTS

Introduction

Abstract specification of a component

- Structural contracts

Interface contracts

Behavioural contracts

Proposal Architecture Description Language

Conclusion & future work
Abstract specification of a component

- The abstract specification contains a formal model of design component, called design component contract.

- A design component contract includes structural contract, behavioural contract and interface contract.
The abstract specification contract is defined by:

- \( ASC ::= \{<\text{Name}>, <\text{SC}>, <\text{IC}>, <\text{BC}>\} \)

For all \( i, j / i \# j \rightarrow \text{name.cpi} \# \text{name.cpj} \)
The abstract specification contract is defined by:

- \( \text{ASC} ::= \{\text{Name}, \text{SC}, \text{IC}, \text{BC}\} \)

describe the relations of the constructs of each design component
The abstract specification contract is defined by:

- \( \text{ASC} := \{ <\text{Name}>, <\text{SC}>, <\text{IC}>, <\text{BC}> \} \)

The finite set of input or output ports attached to a design component and the set of messages sent to or received by a component.
The abstract specification contract is defined by:

- \( \text{ASC} ::= \{ \langle \text{Name} \rangle, \langle \text{SC} \rangle, \langle \text{IC} \rangle, \langle \text{BC} \rangle \} \)

The behavioural properties are constraints such as event ordering, and action sequence of each design component.
Structural contracts

The structural aspect of a design component contract $SC$ is a tuple $SC = (C, A, M, T, Ar, Pc, Pa)$, where

- $C$ is a set of classes in the design component,
- $A$ is a set of attributes defined in classes $C$,
- $M$ is a set of methods defined in classes $C$,
- $T$ is a set of types,
- $Ar$ is a set of access rights $= \{\text{public, protected, private}\}$,
- $Pc$ is a set of connection predicates symbols that capture the relationships For example (Inherit, association, aggregation,…), and
- $Pa$ is a set of action predicates symbols that can perform in a design component For example (invoke, new, return…)
Can be formalized using a subset of First Order Logic (FOL),

The subset of FOL used to describe the structural aspect of a design component comprises variable symbols, connectives (\(\land\)), quantifiers (\(\exists\)), element (\(\in\)) and predicate symbols acting upon variable symbols.

The variable symbols represent class, objects, while the predicate symbols represent permanent relations.
Entity predicates define whether a design component has a specific class (abstract or concrete), what a method (or attribute) is defined in a class....

<table>
<thead>
<tr>
<th>Predicate</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract-class(C)</td>
<td>C plays the role as an abstract-class in the component</td>
</tr>
<tr>
<td>Class(C)</td>
<td>C plays the role as an abstract-class in the component</td>
</tr>
<tr>
<td>x ∈ X</td>
<td>X is an element of set X</td>
</tr>
</tbody>
</table>

Relationship predicates define the relations between classes, attributes, and operations and the actions that a role can perform in a component.

<table>
<thead>
<tr>
<th>Predicate</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inherit(A,B)</td>
<td>B is a subclass of A</td>
</tr>
<tr>
<td>Associate(A,B)</td>
<td>A,B are connected with association relation</td>
</tr>
<tr>
<td>Aggregate(A,B)</td>
<td>A contain a reference to B</td>
</tr>
<tr>
<td>Invoke(A,m1,B,m2)</td>
<td>A method m1 defined in class A calls a method m2 defined in class B</td>
</tr>
<tr>
<td>New(A,m,C)</td>
<td>The method m of class A create anew object of type A</td>
</tr>
<tr>
<td>Return(A,m,O)</td>
<td>The method m of class A returns an object O of type A</td>
</tr>
<tr>
<td>Reference(C1,C2,m)</td>
<td>The multiplicity of association or aggregation relationship from C1 to C2</td>
</tr>
</tbody>
</table>
Example

We consider the structure (class and interaction diagram) of the Observer pattern (Gamma, 1995): (The Observer pattern (also called Publisher-Subscriber).

Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.
Observer pattern (class diagram)

(0) Component-name is Observer where:

1. $\exists$ abstract-class$(Subject, Observer) \epsilon C$;
2. $\land \exists$ class$(ConcreteObserver, ConcreteSubject)} \epsilon C$;
3. $\land \exists$ (attach, detach, getstate, update, notify) $\epsilon M$;
4. $\land \exists$ (void, datatyp) $\epsilon T$;
5. $\land \exists$ Inherit { (Observer, ConcreteObserver) $\land$ (Subject, ConcreteSubject) };
6. $\land \exists$ Invoke{(Invoke(Subject,attach, observer, append) $\land$ (Subject, detach, observer, remove)) $\land$ (Subject, notify, observer, update)};
7. $\land \exists$ Return (concreteSubject, getstate, subjectstate)
8. Where $\exists$ Method {( attach, detach, notify) $\epsilon$ Subject $\land$ (update)$\epsilon$ Observet $\land$ (getstate, notify) $\epsilon$ ConcreteSubject $\land$ (update)$\epsilon$ ConcreteObservet}
The abstract specification contract (ASC) is defined by:

\[
\text{ASC} ::= \text{<Component-Name>} \text{ Where }
\qquad \text{<assertion> and <SC> and <IC> and <BC> End}
\]

In order to capture the evolution, instantiation, and integration processes of design patterns, we extend this formalisation by new operations (structural instantiation, structural evolution, and structural integration).

\[
\text{<SC1> ::= <SC> .and. <S-Instantiation> .and. <S-Evolution> .and. <S-Integration> End.}
\]
CONTENTS

Introduction

Abstract specification of a component

Structural contracts

Interface contracts

Behavioural contracts

Proposal Architecture Description Language

Conclusion & future work
a. Structural Instantiation

To use a design pattern in a particular application, one needs to instantiate it with the application domain information.

This instantiation process may change the generic names of the group of classes into those reflecting the application. It may also change the number of classes in some prescribed way.

Nevertheless, such changes are not arbitrary and have to respect the constraints of the design pattern.

The structural aspect of a design component contract $SC$ is a tuple $SC = (C, A, M, T, Ar, Pc, Pa)$. 

Then, an instance, denoted by:

\[ SC' = (C', A', M', T', Ar', P'c, P'a) \]

of the structure contract \( SC \) is defined by the instantiation of an operation:

\[ \xi: C \times A \times M \times T \times Pc \times Pa \rightarrow C' \times A' \times M' \times T' \times Ar' \times P'c \times P'a \]

Where:

\[ C' = \xi(C) \] is a set of new class names that replace the old class names in the design component;

\[ A' = \xi(A) \] is a set of new attribute names that replace the old attribute names in the design pattern;

\[ M' = \xi(M) = M \] is a set of new method names that replace the old method names in the design pattern;

\[ T' = \xi(T) \] is a set of new type names that replace the old type names in the design pattern;

\[ Ar' = \xi(Ar) = Ar ; \]

\[ Pc' = \xi(Pc) = Pc ; \]

\[ Pa' = \xi(Pa) = Pa. \]
An instance of Observer pattern (class diagram)

(0) Component-name is <Instance-of-Observer>
where:

(1) ∃ class (Observer, Subject, TableView, PieView,
    DataABC) ∈ C';

(2) ∧ ∃ (attach, detach, GetState, update, notify,
    append, remove) ∈ M';

(3) ∧ ∃ (void, datatype) ∈ T';

(4) ∧ ∃ (subject, observers a, b, c) ∈ A';

(5) ∧ ∃ Inherit {{Observer, TableView}
    ∧ (Observer, PicView)
    ∧ (Subject, DataABC)};

(6) ∧ ∃ Invoke{{Subject, attach, observer, append}
    ∧ (Subject, detach, observer, remove)
    ∧ (Subject, notify, observer, update)
    ∧ (TableView, update, Subject, getstate)
    ∧ (Pieview, update, subject, getState) });

(7) ∧ ∃ Return {{DataABC, getstate, a}
    ∧ (DataABC, getstate, b)
    ∧ (DataABC, getstate, c)}

(8) Where ∃ Method {{attach, detach, notify) ∈ Subject
    ∧ (update) ∈ Observer ∧ (getstate, Setstate) ∈ DataABC
    ∧ (update) ∈ PieView ∧ (update) ∈ TableView}
b. Structural evolution

The evolution information of each design pattern allows the designers to change the system design with a minimum impact on other parts of the system.

We propose a solution that is based on extending the abstract structural contract by introducing new primitives and constraints.

<table>
<thead>
<tr>
<th>Primitives</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>+( R,C1,C2)</td>
<td>adding relationships R between class C1 and C2</td>
</tr>
<tr>
<td>- ( R,C1,C2)</td>
<td>removing relationships R1 between class C1 and C2</td>
</tr>
<tr>
<td>C1* C2 (r )</td>
<td>C1 and C2 have an association r</td>
</tr>
<tr>
<td>C1 # C2</td>
<td>C1 and C2 have the same structure</td>
</tr>
<tr>
<td>C1 as R(C2)</td>
<td>C1 may adopt a role R</td>
</tr>
<tr>
<td>Tr (C1, r_i)</td>
<td>List of the relationships of C1</td>
</tr>
</tbody>
</table>
b. Structural evolution

A structural aspect of a design component contract $SC$ is a tuple $SC = (SC-, SC~, preSC, postSC)$, with:

- $SC-$ is a stable structure,
- $SC~$ is the evolves (changeable) structure and,
- $preSC$ and $postSC$ are the constraints (invariant, post-condition and pre-condition).

We define $SC~ = (C~, R~, TR(C, r_1, ..., r_i))$ where

$C~$ are the classes which we can instantiate, add and remove in the design component,
$R~$ are the relationships attached to $C~$, and

$TR= U_i(Tr_i, i=1,..)$ are the list of the relationships of each $Ci \in C~$, with constraints:

$SC = SC~ U SC-$  and  $C~ \cap C- = \{\emptyset\}$
b. Structural evolution

- (0) Component-name is Observer Where:
- (1) \( \exists \text{abstract-class}(\text{Subject, Observer}) \in SC_-; \)
- (2) \( \land \exists \text{class} (\text{ConcreteObserver, ConcreteSubject}) \in SC_-; \)
- (3) \( \land \) 
- (ConcreteObserver, ConcreteSubject)) \in C_-.
- (..) \( \land (\text{Subject, ConcreteSubject}) \in TR_-; \)
- (..) Where \( TR_- = \{TR_- (\text{ConcreteObserver, Inherit}), \)
- \( TR_- (\text{ConcreteSubject, Inherit}) \} \)
- The preSC, postSC are constraints on primitives (+( R,C1,C2) and -( R,C1,C2)).
b. Structural evolution

- The preSC, postSC are constraints on primitives
- \((+ (R,C1,C2) \text{ and } -(R,C1,C2)).\) With these constraints we ensure to confirm that the pre-condition of the evolution is true.
- The addition of one class in a design component may be required if:
  - the new class has a copy (play the same role) in the design component and,
  - this copy (old class) is a member of SC~ (ie: Cardinality of TR > 1).

- The primitive evolution \(+ (R,C1,C2)\) (adding design elements in existing design pattern) is defined as follow:
  - \(+ (R, C1, C2) := \text{new } C1\)
  - \textbf{Pre: } \(C_i \text{ SC~ } \land \text{card (TR}(C,r_1,r_2,\ldots, r_i) \geq 1\)
    \(\land C1 \not\# C_i \land C1 \text{ as R}(C2)\)
  - \textbf{Post: } \(r_i \text{ TR}(C,r_1,r_2,\ldots, r_i) \text{ then } C1* C2 (r_i)\)
b. Structural evolution

\( \textit{new(ConcreteObserver2)} \land \textit{+Inherit (Observer, ConcreteObserver2 )}. \)
c. Structural integration

- We describe the integration of two structural contracts by the following definition:
  
- Let \( SC_1 = (C_1, A_1, M_1, T_1, Ar_1, Pc_1, Pa_1) \) and
- \( SC_2 = (C_2, A_2, M_2, T_2, Ar_2, Pc_2, Pa_2) \)
- be two structural contracts, the integration of \( SC_1 \) and \( SC_2 \)
- denoted by:
  
- \( SC = (C, A, M, T, Ar, Pc, Pa) \) and a function \( \nabla \)

\( \nabla: C_1 \cup C_2 \rightarrow C, \ A_1 \cup A_2 \rightarrow A, \ T_1 \cup T_2 \rightarrow T, \)

\( Ar_1 \cup Ar_2 \rightarrow Ar, \ Pc_1 \cup Pc_2 \rightarrow Pc, \ Pa_1 \cup Pa_2 \rightarrow P \)

with the constraint:

\[ C = \nabla C_1 \cup \nabla C_2 / \forall c \in C_1 \cup C_2 \text{ and } \nabla c \in C_1 \cup C_2 \]
c. Structural integration

• (0) Component-name is Observer Where:
• (1) \( \exists \) abstract-class (Subject, Observer) \( \in \) SC~;
• (2) \( \land \exists \) class (ConcreteObserver, ConcreteSubject) \( \in \) SC~;
• (3) \( \land \) ..............................................
  \( \land \) (ConcreteObserver, ConcreteSubject) \( \in \) C~.
  (..) \( \land \) (Subject, ConcreteSubject) \( \in \) TR~ ;
• (..) Where TR~ = {TR~ (ConcreteObserver, Inherit),
  TR~ (ConcreteSubject, Inherit)}
• The preSC, postSC are constraints on primitives (+ (R,C1,C2) and -( R,C1,C2)).
Integration of two Patterns (Mediator and Observer)

- Let us consider another pattern (Mediator), and its structural contract SC (mediator). The integration of a Mediator instance (SC\textsubscript{mediator}) with the Observer instance (SC\textsubscript{observer}) is shown in Figure.

In this composition, both subjects and observers play the role of Colleague in the Mediator pattern.
Integration of two Patterns (Mediator and Observer)

- (0) Component-name is <ObserverMediator>
- (1) \( \exists \) class (Observer, Subject, TableView, PieView, DataABC, Mediator, ConcreteMediator) \( \in C \);
- (2) \( \land \exists \) (attach, detach, getState, update, notify, append, remove) \( \in M \);
- (3) \( \land \exists \) (void, datatype) \( \in T \);
- (4) \( \land \exists \) (subject, observers, mediator, a, b, c) \( \in A \);
- (5) \( \land \exists \) Inherit \((Observer, TableView)(Observer, PicView)(Observer, ConcreteSubject)
- \( \land (Subject, DataABC) \);
Integration of two Patterns (Mediator and Observer)

\[
\begin{align*}
(6) \land \exists \text{Invoke}\{(\text{Subject, attach, observer, append})
\land \ (\text{Subject, detach, observer, remove})
\land \ (\text{Subject, notify, Mediator, update})
\land \ (\text{Mediator, notify, observer, update})
\land \ (\text{TablaView, update, Mediator, update})
\land \ (\text{PieView, update, Mediator, update})
\land \ (\text{Mediator, update, subject, GetState}) \}\;
\end{align*}
\]
Integration of two Patterns (Mediator and Observer)

(7) \(\land \exists \text{Return} \{(\text{DataABC, getstate, a})\land(\text{DataABC, getstate, b})\land(\text{DataABC, getstate, c})\}\)

(8) Where \(\exists \) Method \{(attach, detach, notify) \in Subject\)
- \(\land (updtate) \in Observer\)
- \(\land (getstate, Setstate) \in DataABC\)
- \(\land (notify, updtate) \in Mediator\)
- \(\land (updtate) \in ConcreteMediator\)
- \(\land (updtate) \in PieView\)
- \(\land (updtate) \in TableView\)
Contents

Introduction

Abstract specification of a component

- Structural contracts

Interface contracts

Behavioural contracts

Proposal Architecture Description Language

Conclusion & future work
Interface contracts

- Let a tuple $IC = (P, IP, OP, IM, OM, IMI)$

  - $P$ is a finite set of process names,
  - $IP$ is a finite set of input ports attached to a process,
  - $OP$ is a finite set of output ports attached to a process,
  - $IM$ is a finite set of input messages sent to a process and
  - $OM$ is a finite set of output messages sent from a process,
  - $IMI$ is the finite set of input messages sent from outside the design component to a process.
Observer pattern (interaction diagram)

(0) Component-name is Observer where:

(1) \( \exists (\text{aConcreteSubject, aConcreteObserver, anotherConcreteObserver}) \in C \)

(2) \( \land (\text{inOS, inSO, self, input}) \in IP \)

(3) \( \land (\text{outOS, outSO, output}) \in OP \)

(4) \( \land (\text{attach, detach, getstate, setstate, update, notify, change}) \in IM \)

(5) \( \land \exists (\text{attach, detach, getstate, setstate, update, notify}) \in OM \)

(6) \( \land \exists (\text{change}) \in IMI \)
## Interface contracts

- This allows assertions about the gates (set of input or output ports attached to a process) to appear in pre-conditions, and post-conditions.

- Let \( IC1 = (IC, Constraint) \) we denote:
  
  \[
  p \in IP(p) = \{ i \in IP \mid gate_{Ini} = p \} \land \\
  p \in OP(p) = \{ i \in OP \mid gate_{Outi} = p \} \land \\
  m \in IM(p) = \{ i \in IP, m \in IM \mid gate_{Ini} ?m \} \land \\
  m \in OM(p) = \{ i \in OP, m \in OM \mid gate_{Outi} !m \} \land \\
  \text{all-gateIN=} \{ \text{all IP(p)/ } p \in \text{ Component } \} \land \\
  \text{all-gateout=} \{ \text{all OP(p)/ } p \in \text{ Component } \} \land \\
  \]

- **Where** Constraint /*constraints on gates*/:
  
  \[
  \forall \rightarrow gate_{Ini} \neq gate_{Inj} \land \\
  gate_{Outi} \neq gate_{Outj} \land \\
  gate_{Inj} ?mi \in gate_{Outi} !m \ i \land \\
  \text{gate}_{Outj}?mi \in gate_{ini} !m \\
  \]

- 11/06/2013 SEDE 2010, San Francisco
CONTENTS

Introduction

Abstract specification of a component

° Structural contracts

Interface contracts

Behavioural contracts

Proposal Architecture Description Language

Conclusion & future work
We have chosen a basic LOTOS for defining a formal semantic model of behavioural contracts because it represents a powerful approach to modeling of behaviour and concurrency.

Powerful ability for describing behaviour and the availability of tools enabling formal verification.
Observer pattern (interaction diagram)

(Specification Observer [input, output] : noexit := 
/*.... Signature......*/

behaviour

aConcreteSubject [input, output]

| [input, output] |

aConcreteObserver [input, output]

[]

anotherConcreteObserver [input, output]

where

Process aConcreteSubject [inCS, outCS] := noexit

?setstate; !notify; !update ; ?getstate;

aConcreteSubject [inCS, outCS]

Endprocess

Process aConcreteObserver [inaCO, outaCO] := noexit

I; !setstate; ?update; !getstate

aConcreteObserver [inaCO, outaCO] Endprocess

Process bConcreteObserver [inbCO, outbCO] := noexit

I; !setstate; ?update; !getstate

aConcreteObserver [inbCO, outbCO]

Endprocess

Endspec
CONTENTS

Introduction

Abstract specification of a component
  Structural contracts

Interface contracts

Behavioural contracts

Proposal  Architecture Description Language

Conclusion & future work
LOTOS-ADL has been designed to address specification of structural and dynamic architectures.

- The structural viewpoint may be specified in terms of: components, connectors, and configurations of components and connectors.

- The behavioural viewpoint may be specified in terms of: actions a system executes or participates in, relations among actions to specify behaviours, and behaviours of components and connectors, and how they interact.
LOTOS-ADL

<LOTOS-ADL> := < structural viewpoint, behavioural viewpoint >;

< structural viewpoint > := < component, connector, configuration >/
  component := < cp1, cp2, …., cpn >  n ≥ 2 and
  connector := < ct1, ct2, …., ctm >  m ≥ 1
  with constraints:
      for all cpis, cj ∈ component / name.cp1 # name.cpj
      ∈ connector / name.ct1 # name.ctj
  configuration := < /* LOTOS operators construct */ >

< behavioural viewpoint > := < LOTOS behavior expression >
LOTOS-ADL

- A LOTOS specification describes a system through a hierarchy of active components, or processes.

- A process is an entity able to realize non-observable internal actions, and also interact with others processes through externally observable actions.
Conclusion & future work

- Use of formal specifications to assist and automate a system design process based on reusable components and architectures.

- How to adopt LOTOS as ADL to describe the behaviour of software architecture.

- This language is mathematically well-defined and expressive: it allows the description of concurrency, non-determinism, synchronous and asynchronous communications.
Conclusion & future work

- These positive features encourage us to adopt LOTOS as an ADL for describing both component and connector allowing us to check behaviours properties.

- We are investigating to proposing a rules-based transformation enabling the mapping from LOTOS specification to JAVA pseudo code.
Questions...