Contract-based approach to analyze software components

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

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

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

ICECCS'08/Workshop « UML&AADL'2008 »,
BELFAST
April, 2, 2008

CONTENTS

Introduction (1/3)

- Component-based software development:

  - Building large software systems
  - Software components.

- Component-based approaches:

  - Create, deploy software systems assembled from components.

Introduction (2/3)

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 (3/3)

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.

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

Background

LOTOS

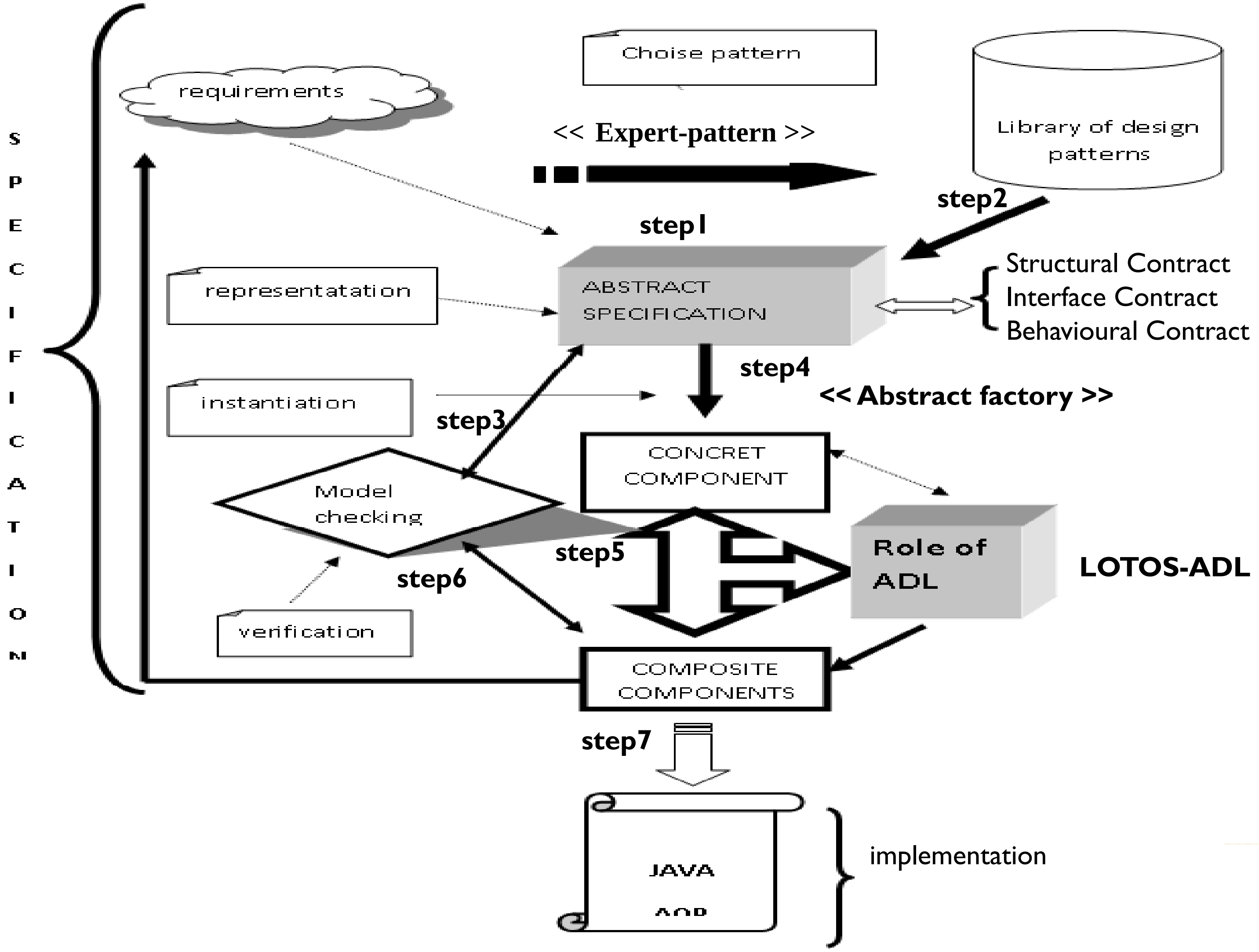
- Language of Temporal Ordering Specifications (LOTOS) is a formal description technique standardized at ISO, based on a combination of CCS [Milner] and CSP [Hoare].
- The basic idea supporting LOTOS is that systems can be specified by expressing the relations among the interactions that constitute their externally observable behaviour.
- In LOTOS, a system is seen as a process, possibly consisting of several sub-processes.

operator	Description	Example
[ ]	either P1[a,b] or P2[c,d] depending on the environment	P[a,b,c,d]=P1[a,b] [ ] P2[c,d]
|||	Parallel composition without synchronization: P1[a,b] is independent from P2[c,d]	P[a,b,c,d]=P1[a,b]|||P2[c,d]
[b]	Parallel composition with synchronization on gate b	P[a,b,c,d]=P1[a,b]|[b]|P2[c,d]
>>	Sequential composition: P1[a,b] is followed, when P1 terminated, by P2[c,d];	P[a,b,c,d]=P1[a,b] >> P2[c,d]
[>	Disrupt: P1 [a, b] may be interrupted at any time before its termination by P2[c, d].	P[a,b,c,d]=P1[a,b] [> P2[c,d]
;	Process prefixing by action a	a;P
Stop	Process which cannot communicate with any other process	Stop
Exit	Process which can terminate and then transforms itself into stop	Exit

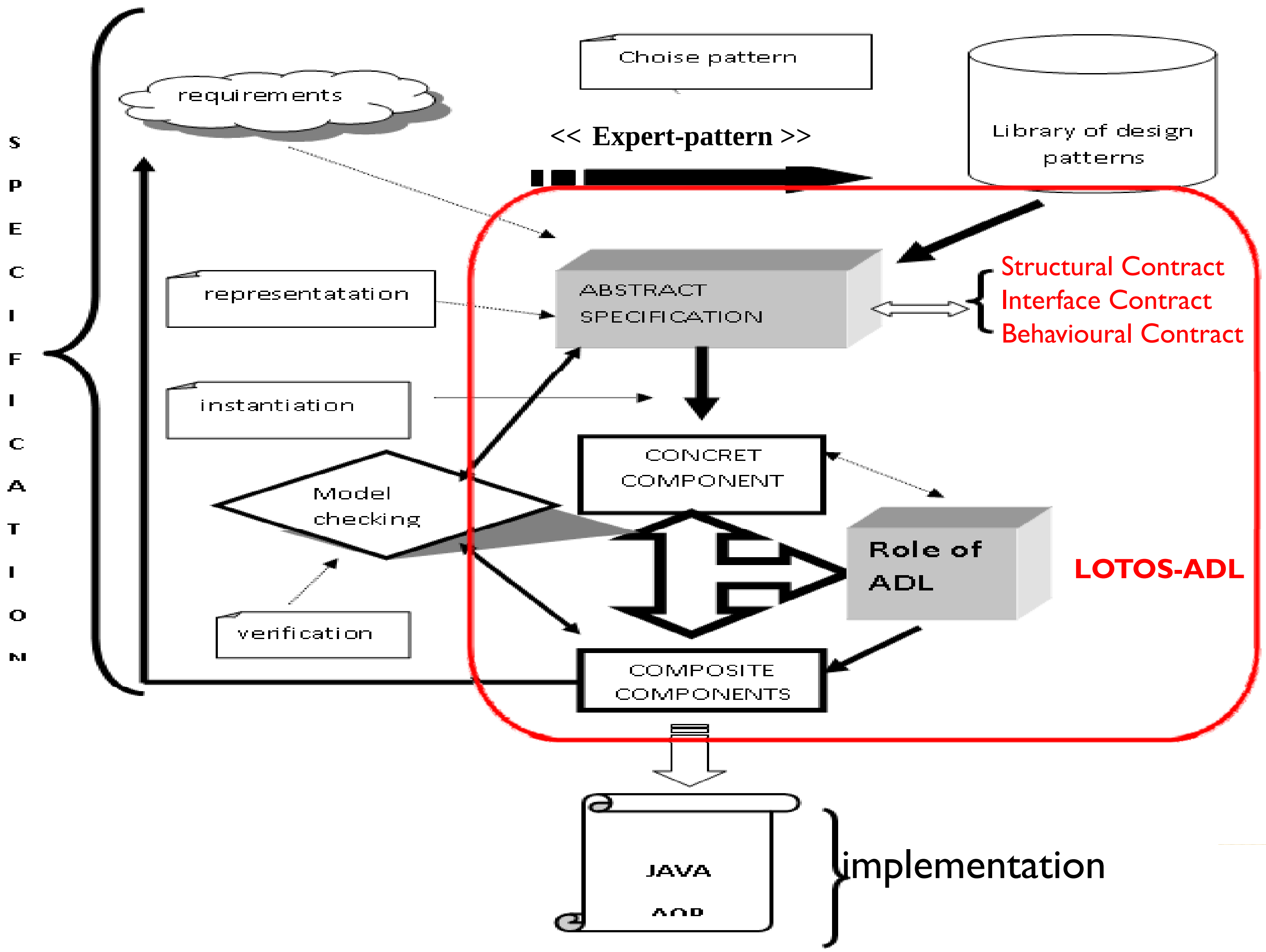
LOTOS operators

CONTENTS

Overview of our approach



Overview of our approach



CONTENTS

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::={<Name>, <SC>,<IC>, <BC>}

For all i, j / i # j $\longrightarrow$ name.cpi # name.cpj

The abstract specification contract is defined by:

- ASC::={<Name>, <SC>,<IC>, <BC>}

describe the relations of the constructs of each design component

The abstract specification contract is defined by:

- ASC::={<Name>, <SC>,<IC>, <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:

- ASC::={<Name>, <SC>,<IC>, <BC>}

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

a. Structural contracts

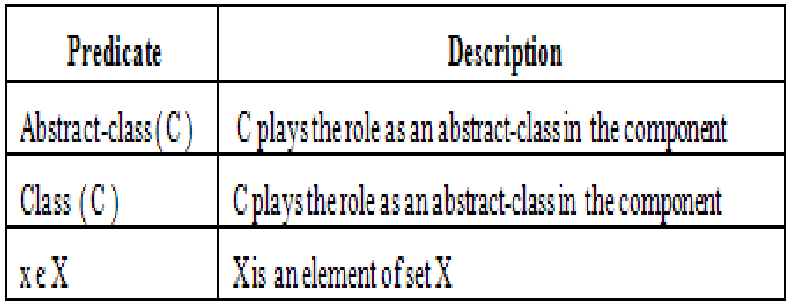
The structural aspect of a design component contract SC is a tuple SC = (***C, A, M, T, Ar, Pc, Pa,***),where

- ***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 = {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…)

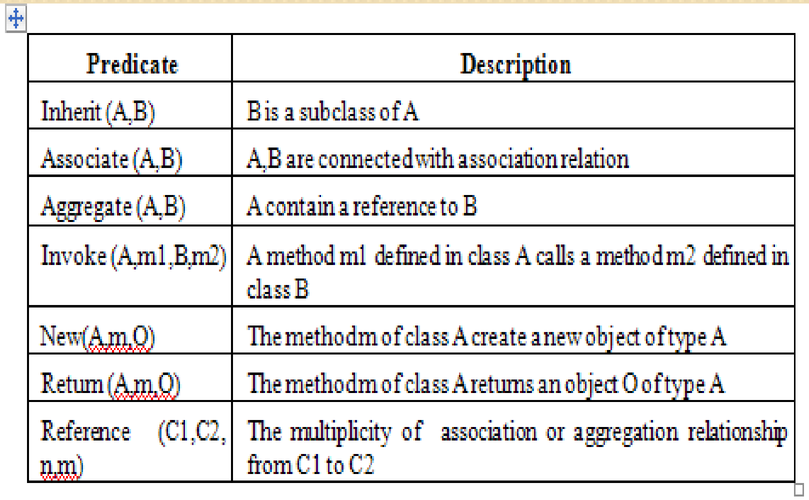
- Can be formalized using a subset of First Order Logic (FOL),
- The subset of FOL used to describe the structural aspect of a design component comprises variable symbols, connectives ('∧'), quantifiers ('∃'), element (ϵ) and predicate symbols acting upon variable symbols.
- The variable symbols represent 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….

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



Predicate	Description
Abstract-class (C)	C plays the role as an abstract-class in the component
Class (C)	C plays the role as an abstract-class in the component
x e X	X is an element of set X



Predicate	Description
Inherit (A,B)	B is a subclass of A
Associate (A,B)	A,B are connected with association relation
Aggregate (A,B)	A contain a reference to B
Invoke (A,m1,B,m2)	A method m1 defined in class A calls a method m2 defined in class B
New(A,m,O)	The method m of class A create a new object of type A
Return (A,m,O)	The method m of class A returns an object O of type A
Reference (C1,C2, n,m)	The multiplicity of association or aggregation relationship from C1 to C2

Entity predicates

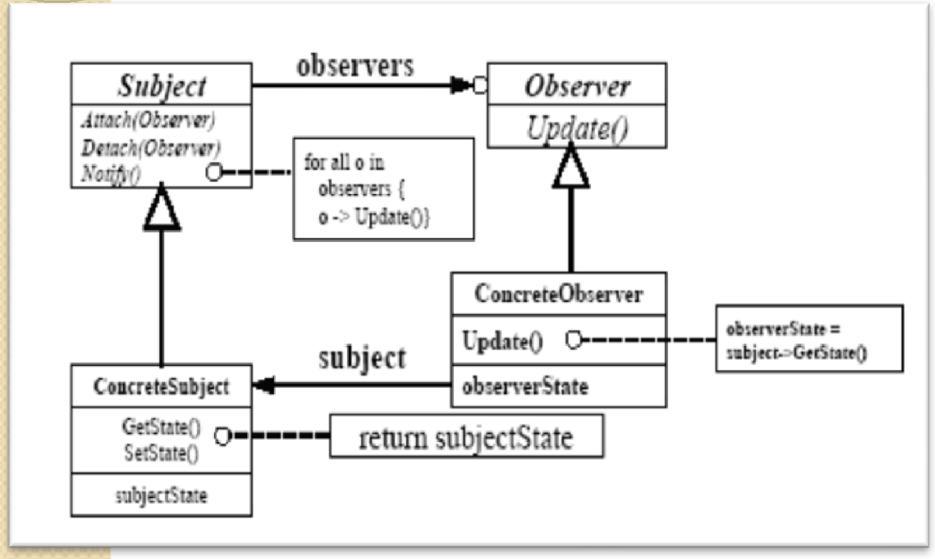
Relationship predicates

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) ∃ **abstract-class**(Subject,Observer) ϵ C;

(2) ∧ ∃ **class**(ConcreteObserver,ConcreteSubject)} ϵ C;

(3) ∧ ∃ (attach, detach, getstate, update, notify) ϵ M;

(4) ∧ ∃ (void, datatyp)} ϵ T;

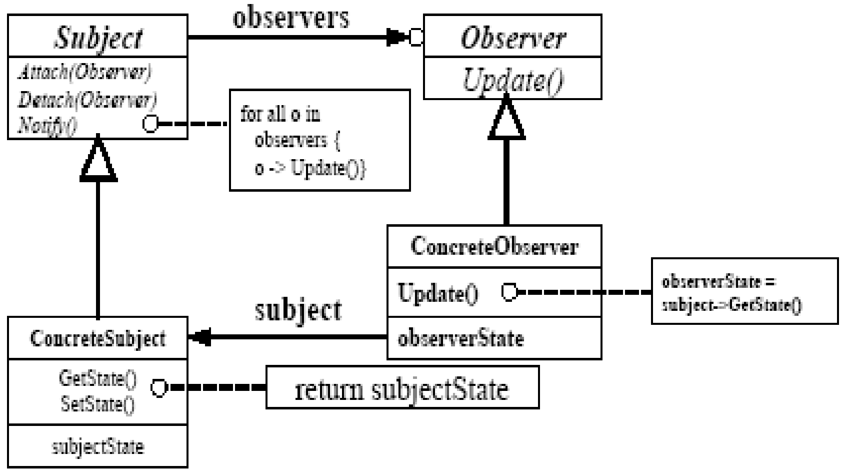
(5)∧ ∃ **Inherit** { (Observer, ConcreteObserver)
∧ (Subject, ConcreteSubject) };

(6) ∧ ∃ **Invoke**{(Invoke(Subject,attach, observer, append) ∧ (Subject, detach, observer,remouve)
∧ (Subject, notify, observer, update)};

(7) ∧ ∃ **Return** (concreteSubject, getstate, subjectstate)

(8) **Where** ∃ Method {(attach, detach, notify) ϵ Subject
∧ (updtate)ϵObservet ∧ (getstate, notify) ϵ ConcreteSubject
∧ (updtate)ϵ ConcreteObservet}

Observer pattern (class diagram)



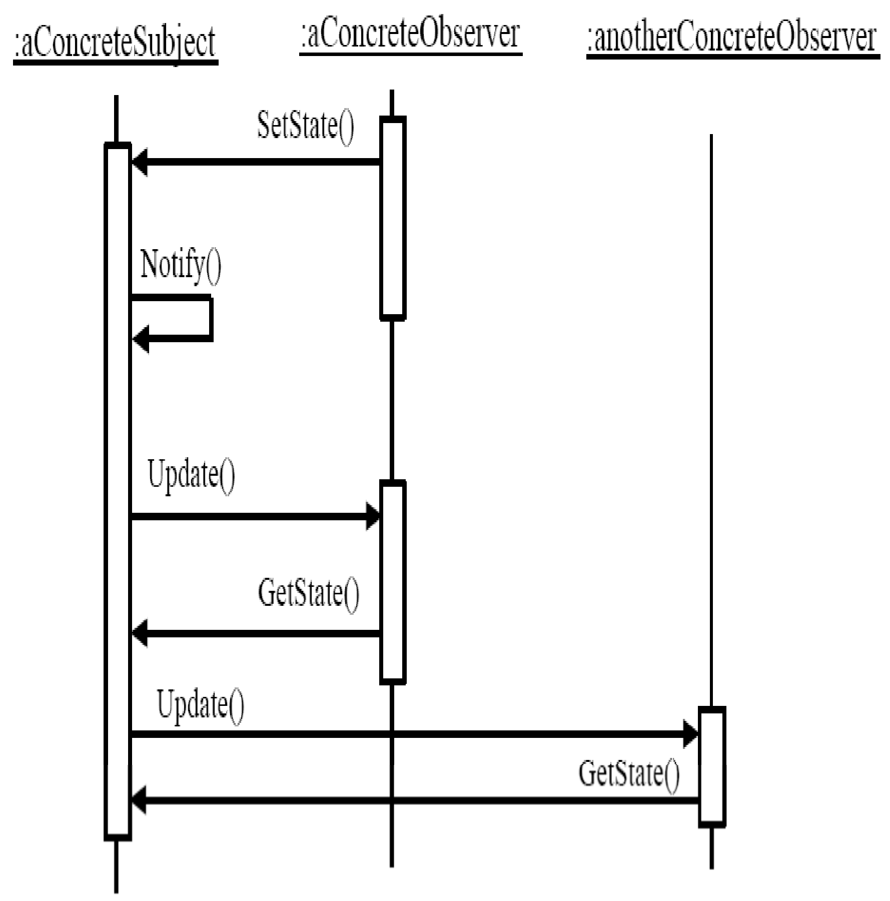
```
Public interface Observer {
        Public void Update (subject s) ;}
Public interface Subject {
        Public void attach (Observer o) ;
        Public void detach (Observer o);
        Public void notify ();    }
Public Class ConcreteSubject implements
    Subject {
        Public void attach (Observer o)
    {...........} ;
        Public void detach (Observer o)
    {...........};
        Public void notify () {...........};
    }
Public Class ConcreteObserver implements
    Observer {
Public void Update (subject s)
    {.................} ;}
```

b. 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)∃ (aConcreteSubject,aConcreteObserver, anotherConcreteObserver) є C

(2) ∧ (inOS, inSO,self, input) є IP

(3) ∧ (outOS, outSO, output) є OP

(4) ∧ (attach, detach, getstate, setstate,update, notify, change) є IM

(5) ∧ ∃ (attach, detach, getstate, setstate, update, notify) є OM

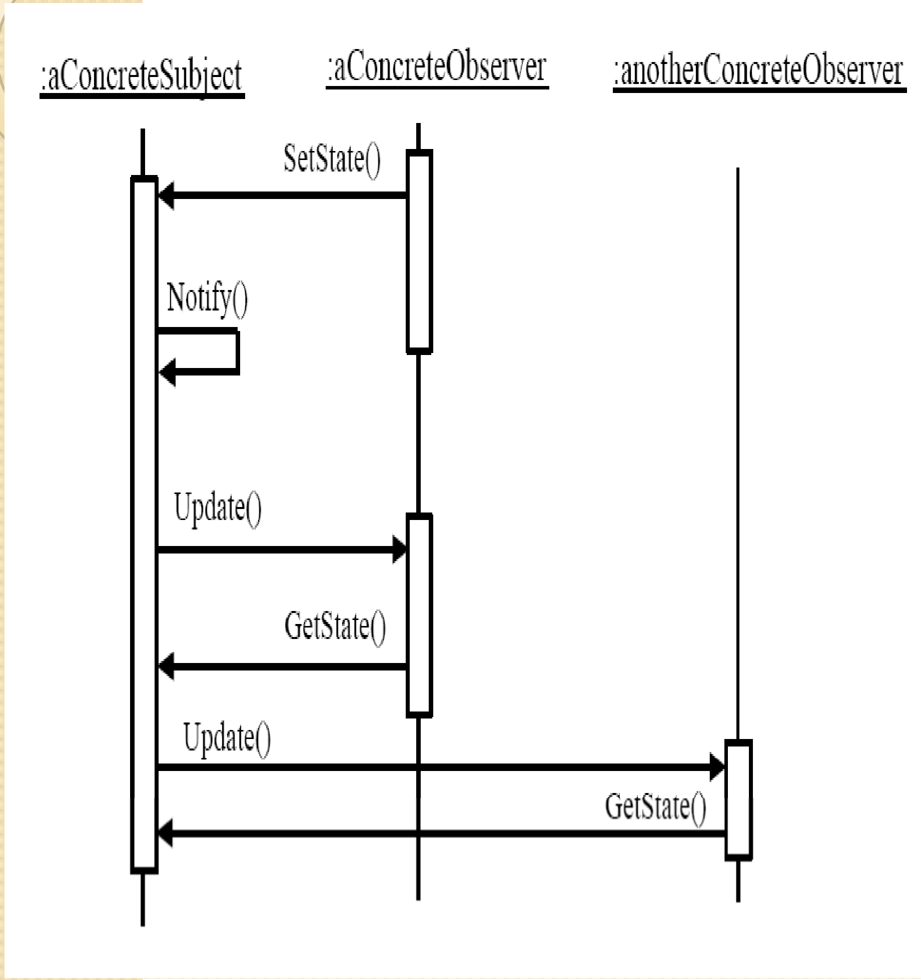
(6) *∧ ∃* (change) *є IMI*

c. Behavioural contracts

- 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 ;?getsate;
            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

Proposal Architecture Description Language (LOTOS-ADL)

- 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 := <ct_1, ct_2, ….., ct_m> m ≥ 1

with constraints:

for all cpi, cpj є component / name.cpi # name.cpj

є connector / name.cti # name.ctj

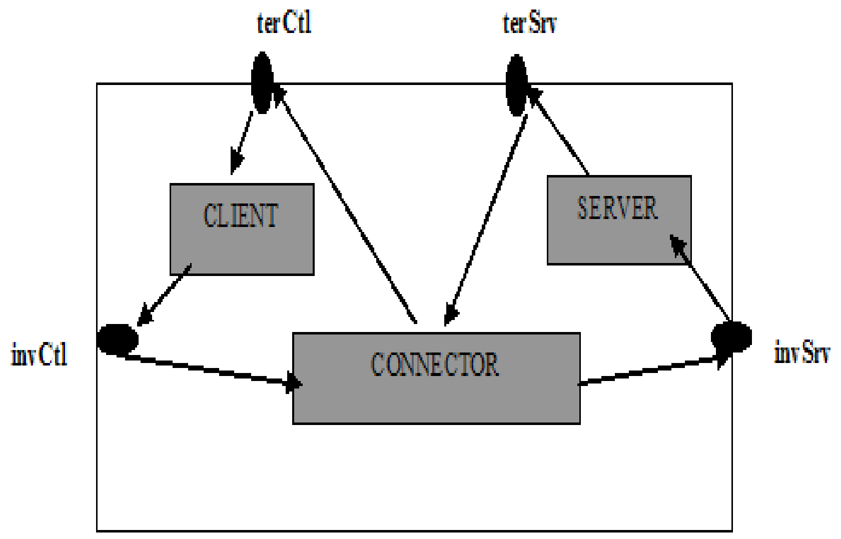
configuration: = < /* LOTOS operators construct*/>

<**behavioural viewpoint**>:= < LOTOS behavior expression >

LOTOS-ADL

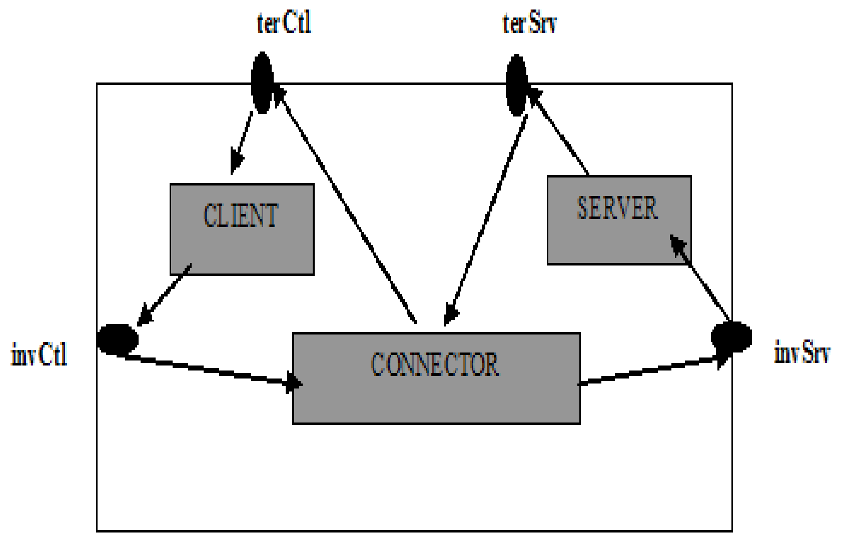
- A LOTOS specification describes a system through a hierarchy of active components, or processes.
- A process is an entity able to realize non-observable internal actions, and also interact with others processes through externally observable actions.

simple client-server



```
specification Client-Server
   [invClt,terClt,invSrv,terSrv] :
   noexit:=
    library RESULT, SERVICES endlib
       behaviour
        Client [invClt, terClt]
             |[invClt, terClt]|
  connector [invClt, terClt, invSrv,
            terSrv]
       |[invSrv, terSrv]|
       Server [invSrv, terSrv]
       where
           .........
             .........
     Endprocess
```

simple client-server



process Connector
[invClt,terClt,invSrv,terSrv] **: noexit: =**
invClt ? s : SERVICE ? op: OPER /*
the client passes the request to
connector* /
invSrv ! s ! op; /* the connector passes
the request to the server*/
terSrv ! s ? r : RESULT; /*the server
passes the reply to the connector*/
terClt ! s ! r; /*the connector passes the
reply to the client*/
Connector [invClt, terClt,invSrv,terSrv]
Endproc

CONTENTS

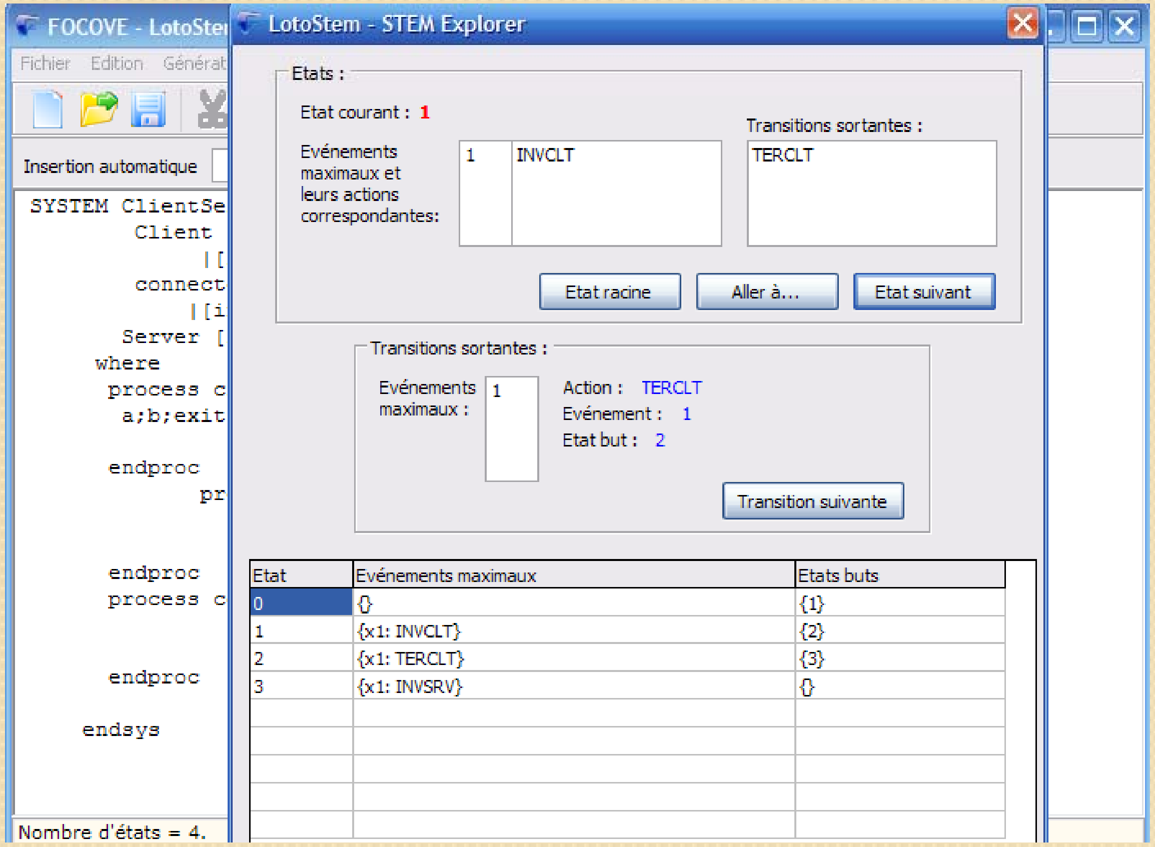
validation

For the verification of our approach, we use the FOCOVE (Formal Concurrency Verification Environment) (available in www.focove.new.fr)

The FOCOVE environment is dedicated to the design and verification for component based software development.

Translates a LOTOS program into a Labelled Transition System (LTS) describing its exhaustive behaviour.

THE FOCOVE ENVIRONMENT AND GENERATION OF THE LABEL TRANSITION SYSTEMS



```
SYSTEM ClientSe
        Client
             |[
        connect
            |[i
       Server [
     where
      process c
       a;b;exit

      endproc
             pr


      endproc
      process c

      endproc

    endsys
```

Nombre d'états = 4.

Etat courant : 1

Evénements maximaux et leurs actions correspondantes: 1 INVCLT

Transitions sortantes : TERCLT

Evénements maximaux : 1

Action : TERCLT

Evénement : 1

Etat but : 2

Etat	Evénements maximaux	Etats buts
0	{}	{1}
1	{x1: INVCLT}	{2}
2	{x1: TERCLT}	{3}
3	{x1: INVSRV}	{}

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

- We propose a solution which is based on extending the abstract structural contract by introducing new primitives and constraints .
- We plan to extend a concept of contract using pre- and post-conditions for describing the semantics of component's services.
- We are investigating to proposing a rules-based transformation enabling the mapping from LOTOS specification to JAVA pseudo code.

Questions…