A Contract-Based Language to Specify Design Components

Abdelhafid Zitouni[1], Mahmoud Boufaida[1], Lionel Seinturier[2]

[1] *Laboratory LIRE, Computer Science Department*
Mentouri University of Constantine, Algeria, 25000
{ah_zitouni,@yahoo.fr, mboufaida@umc.edu.dz}

[2] *University of Lille LIFL-INRIA ADAM*
59655 Villeneuve d'Ascq, France
lionel.seinturier@lifl.fr

Abstract

For component-based software development to be successful in organizations, the software developers must give close attention to the design of components as independent abstractions with well-specified behaviors. Without well-specified behaviors the possibility to distribute and acquire software components will be limited. In this article, we present a contract-based approach to analyze and model the properties of design components and their composition in order to detect and correct composition errors. This approach permits to characterize the structural, interface and behavioural aspects of design component. As a matter of fact, we give a specific form of their instantiation, evolution and integration. To enable this, 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. In addition, we propose the use of the LOTOS language as an ADL for formalizing these aspects. We illustrate the approach by applying it to a standard design pattern.

Keywords: *Architecture Description Language, Design by contract, design components, Design Patterns, LOTOS.*

1. Introduction

Component-based approaches have been proposed to create and deploy software systems assembled from components. The use of previously developed components should lead to faster time to market for complex software applications. Therefore, component-based software development is a promising solution to some of the problems that designers, developers and integrators face when building their systems [3]. Software patterns are a design paradigm used to solve problems that arise when developing software within a particular context. Patterns capture the static and dynamic structure and the collaboration among the different components in a software design. A key promise of the pattern-based approach is that it may greatly simplify the construction of software systems, reuse experience and reduce cost. Design patterns [6] have been proposed to reify a good design practice from conceptual design building blocks into a composable form.

Since a design pattern is a recurring piece of software design, it can be seen as a component, called a design component in [7], and used to reify good design practice from conceptual design building blocks into a composable form. Design components focus on component-based problem solving instead of component-based implementation.

The benefits of design patterns are that they serve as guidance to the novice designer, and they provide an extended vocabulary for documenting software design. Unfortunately, the descriptive format popularized by these catalogs is inherently imprecise. As a consequence, it is unclear when a pattern has been applied correctly, or what can be concluded about a system implemented using a particular pattern.

In order to address the ambiguity issues associated with design pattern descriptions, we introduce the concept of a design pattern contract as a formalism for precisely specifying design patterns. The responsibility of a pattern contract precisely characterizes the requirements that must be satisfied by the designer when applying a particular pattern.

Formal specification and verification techniques are useful for design analysis since they are more precise, expressive, and unambiguous than the informal techniques, such as graphical and textual notations. In order to achieve an effective reuse, we argue that it is important to specify both functional and architectural properties of a component in terms of formal specifications. Formal specifications are amenable to automation in analyzing component properties and thus facilitate the determination of reuse. The formal description technique LOTOS (Language of Temporal Ordering Specifications) [2] was originally designed to specify the interactions among communicating processes, thus making it suitable for capturing the architectural (interaction) properties of components. Furthermore, LOTOS also incorporates the algebraic specification language ACT-ONE [5] for specifying data and related operations. In addition, LOTOS is an ISO standardized formal specification language and has the rich support of industrial tools.

A contribution of this article, is to provide a rigorous description of component functionality. This description can be achieved by means of contracts [8], using pre- and post-conditions for describing the semantics of component's services. Another contribution of this article is a proposition of a novel Architecture Description Language (LOTOS-ADL) that has been designed to address specification of structural and dynamic architectures.

The rest of this article is organised as follows. Section 2 presents an overview of our approach. Section3, the main section of this article, focuses on the abstract specification of a component. Section 4 presents the concepts of LOTOS-ADL. Section 5 illustrates a case study and provides an overview of our environment of validation. Finally, the last section concludes the article and gives some directions for a future work.

2. Overview of the Approach

In this section, we present an overview of our approach and outline the general ideas in our formal models. We separate the abstract specification from its implementation.

Our main goal is to provide a systematic approach for a software designer to model and analyze component integration during the design phase, the early planning stage of the software lifecycle.

In [12] we have presented a systematic approach for a software designer to model and analyze component integration during the design phase, the early planning stage of the software lifecycle. This approach includes a process of representing, specifying, instantiating and integrating design components and analyzing their compositions, which are captured as contracts. The process is illustrated in Fig.1.

This approach allows one to design components to be reused by making the components description available in a component library. With this approach, the designer can not only model the design component precisely, unambiguously and expressively, but also detect the interactions between components and correct design errors before implementation [11]. As

shown in figure1, our approach begins by four steps: (Analysis, selection, abstract specification and the instantiation steps). These steps are describes as follows:

-Analysis: the purpose of this step is to analyze the application requirements and to decide on the set of design patterns that will be used in designing the system. In [12] we show that the specification and the description of the system configuration and its components must be put into a form amenable for analysis and design [12].

-Selection: in this step we analyze the responsibilities and the functionalities of each component and identify candidate patterns that could provide a design solution for each component. In doing so, we have considered the design problem that we want to solve and match it to the solution provided by general purpose design patterns (expert pattern [6] is a good candidate for this task) [11].

-The abstract specification: this step describes a formal model of design component with several contracts (section 3).

-Instantiation: in this step, we create instances of the selected patterns and identify the relationships between these instances (This is a role of the Abstract factory pattern). Finally, we use the pattern instances and their relationships to construct the composite component.

We use the LOTOS-ADL for this task (section 5). During the design or design refinement phases we could discover that a selected pattern has limitations or impacts on other design aspects. In this case, the designer would revisit this design level to choose another pattern, replace previous choices, or create a new pattern dependency or a new uses relationship.

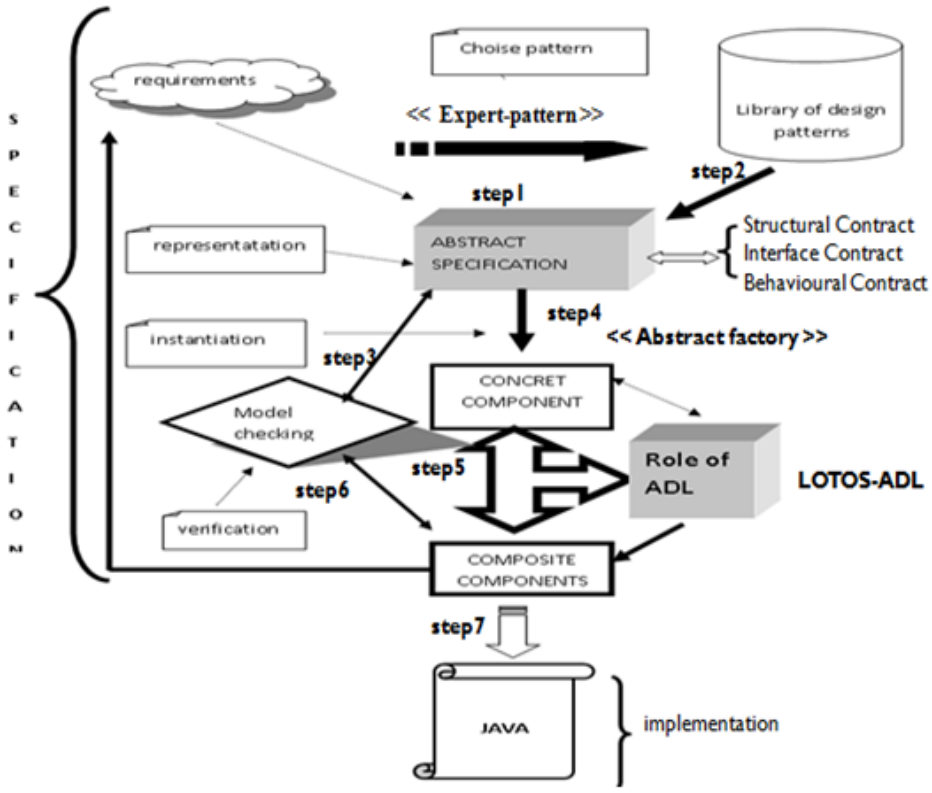

Fig.1. Overview of our Approach [12]

In this article we focus on the abstract specification of the component and the ADL for describing architecture of component-based software, which provide explicit support for specifying components. ADLs are important since they can document component-based architecture early, reason about their properties, and automate their analysis and system generation [9].

3. Abstract Specification of a Component

The abstract specification is inspired from the work of Dong and al. [4] and contains a formal model of design component, called design component contract. A design component contract includes structural contract (SC), behavioural contract (BC) and interface contract (IC).

The structural properties describe the relations of the constructs of each design component. The behavioural properties are constraints such as event ordering, and action sequence of each design component. The interface contract describes 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. We define an abstract specification contract (ASC) as:

ASC::=<Component-Name> **Where**
<assertion>**and** <SC>**and**<IC>**and** <BC> **End**

3.1. A motivating Example

In order to motivate this article, we consider the structure (class and interaction diagrams) of the Observer pattern shown in fig.2 [6]: The Observer pattern regulates how a change in one object can be reflected in an unspecified number of dependant objects).

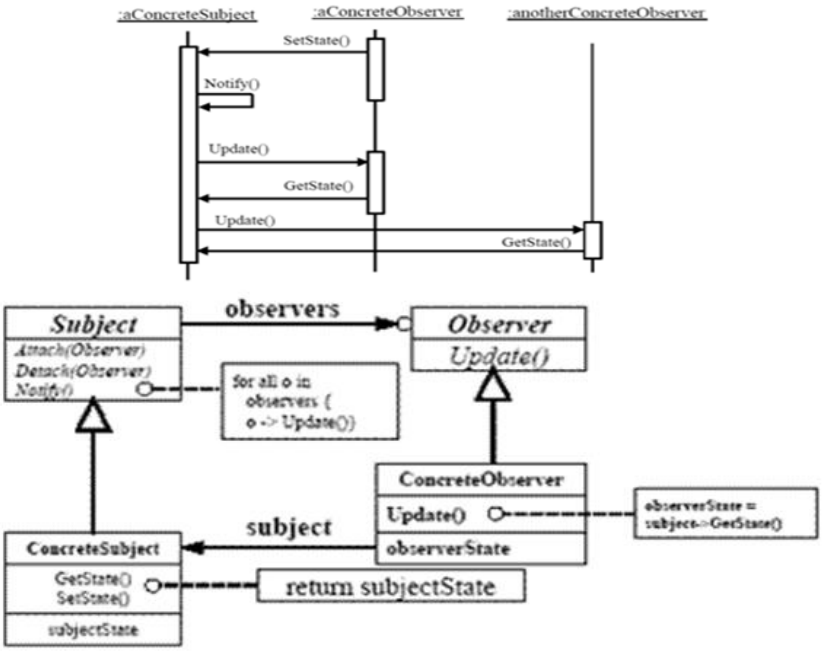

Fig.2. Observer Pattern (interaction diagram, class diagram)

3.2. Structural Contracts

In [12], we have formalized the structural aspect of a design component contract by using a subset of First Order Logic (FOL), because the relations between pattern participants can be easily expressed as predicates. The subset of FOL used to describe the structural aspect of a design component comprises variable symbols, connectives ('∧'), quantifiers ('∃'), element (∈) and predicate symbols acting upon variable symbols. The variable symbols represent class, objects, while the predicate symbols represent permanent relation [12].

We define two groups of predicates, entities (Table 1) and relationships (Table 2).

- 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.

Table1. Entity Predicates

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 ∈ X	X is an element of the set X

Table2. Relationship Predicates

Predicate	Description
Inherit (A,B)	B is a subclass of A
Associate (A,B)	A,B are connected with a 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 creates a new object of type A
Return (A,m,O)	The method m of class A returns an object O of type A

The abstract specification of structural contract is done by[12]:

(0) **Component-name** is Observer **where:**

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

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

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

(4) ∧ ∃ (void, datatype)} ∈T;

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

(6) ∧ ∃ **Invoke**{(Invoke(Subject,attach, observer, append)

∧ (Subject, detach, observer,remove)

∧ (Subject, notify, observer, update)};

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

(8) **Where** ∃ Method {(attach, detach, notify) ∈Subject ∧ (updtate) ∈ Observer

∧ (getstate,notify) ∈ConcreteSubject

∧ (updtate) ∈ConcreteObserver}

Based on this description, and to capture the evolution, instantiation, and integration processes of design patterns, we extend in this article, this formalisation by new operations (structural instantiation, structural evolution, and structural integration).Then a new structured contract (SC1) is defined as the following:

< SC1>**:=** <SC>**.and.** <S-Instantiation> **.and.** <S-Evolution>.**and.** <S- Integration>

3.2.1. Structural Instantiation: When a component is used in a specific application, it needs to be instantiated to include the application domain information. This process can be achieved by unifying the arguments of the description of each design pattern component with terms representing domain information.

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**) [13], 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 a operation:

£: C×A ×M× T × Ar × Pc × Pa → C′×A' ×M′× T' × Ar′ × P'c × P'a Where:
C' = £ (C) is a set of new class names that replace the old class names in the design component;
A' = £ (A) is a set of new attribute names that replace the old attribute names in the design pattern;
M' = £ (M)= M is a set of new method names that replace the old method names in the design pattern;
T' = £ (T) is a set of new type names that replace the old type names in the design pattern;
Ar' = £ (Ar)= Ar ;
Pc' = £ (Pc)= Pc ;
Pa' = £ (Pa)= Pa.

Let us an instance, shown in Figure 2.1, of the Observer pattern described in Figure 2. This instance describes an application for visualizing data elements (a, b, c) with different views. The structural contract of an instance of SC (observer) is:

SC' = (**C', A', M', T', Ar', P'c, P'a)** with the instantiation operation £

(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) ∧ (TablView, 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 ∧ (updtate) ∈ Observer
∧ (getstate,Setstate) ∈ DataABC ∧ (updtate) ∈PieVie ∧ (updtate) ∈ TableView}

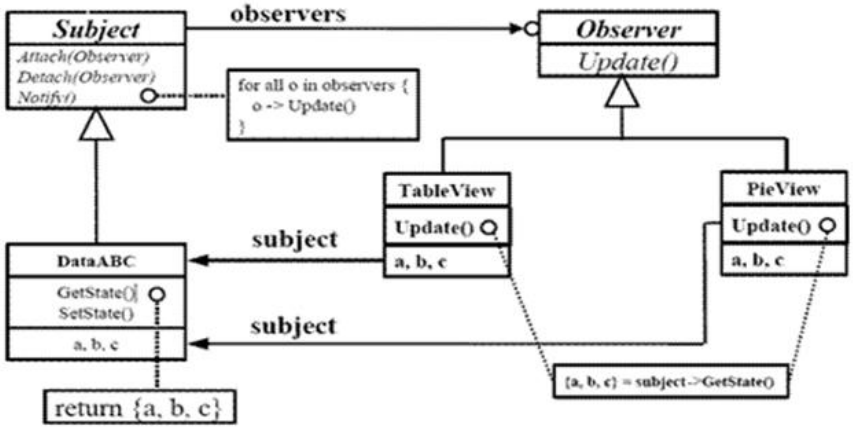

Fig2.1 An Instance of the Observer Pattern

3.2.2. Structural Evolution: One of the important goals of design patterns is design for change. Design patterns capture some expert design experience by partitioning software designs into a stable part and changeable part. By separating and encapsulating both parts, the change impact of a software design can be minimized.

After a design pattern is applied in a software application, the designer may change the application design in the particular ways directed by the design pattern. The evolution information of each design pattern allows the designers to change the system design with a minimum impact of another part of the system [3].

To enable this we have proposed in [13] a solution which is based on extending the abstract structural contract by introducing new primitives and constraints (seeTable3).

The primitives include the addition or removal of a modelling element, such as a class, a method, a field or a composition, a realization or a dependency relation.

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, r1,….,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 the Ci ∈C~, with constraints:

SC =SC~ ∪ SC– and C~ ∩ C- = {∅}

Table3. Evolution Predicates

Primitives	Description
+(R,C1,C2)	Adding relationships R between class C1 and C2
- (R,C1,C2)	Removing relationships R1 between class C1 and C2
C1* C2 (r)	C1 and C2 have an association r
C1 # C2	C1 and C2 have the same structure
C1 as R(C2)	C1 may adopt a role R
Tr (C1, r_i)	List of the relationships of C1

For example: the abstract specification of structural contract is done by:

(0) Component-name **is** Observer **Where**:

(1) ∃ **abstract-class**(Subject,Observer) ∈ SC-;

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

(3) ∧ …….

∧ (ConcreteObserver,ConcreteSubject)}ϵC~.

(..) ∧ (Subject, ConcreteSubject) } ϵ TR~ ;

(..) **Where** TR~= {TR~ (ConcreteObserver, Inherit),
TR~(ConcreteSubject, Inherit) }

The preSC, postSC are constraints on primitives (+(R,C1,C2) and -(R,C1,C2)).

With these constraints we ensure to confirm that the pre-condition of the evolution is true. Then, we may assume that the post-condition is true after the evolution (the addition or removal of the class does not cause any effect on the existing classes of the design).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) := **new** C1
Pre: ∀ C_i ∈SC~ ∧ card (TR(C,r_1,r_2,…., r_i) ≥1
∧ C1 # C_i ∧ C1 as R(C2)
Post: ∀ r_i ∈ TR(C,r_1,r_2,…., r_i) then C1* C2 (r_i)

For example in the Observer pattern the evolution (see Figure.2.2) is done by:
(new(*ConcreteObserve2)* ***∧ +Inherit (****Observer, ConcreteObserver2)).***

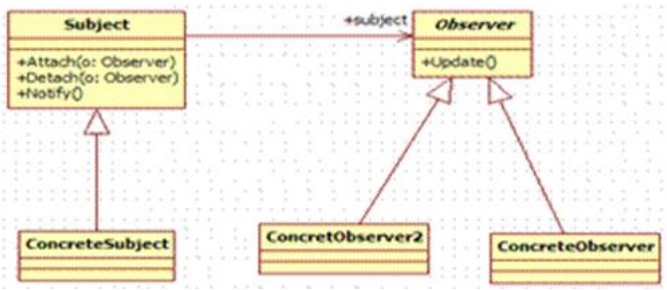

Fig.2.2. Observer Pattern (with two concretes observers)

In this example, the evolution is a simple addition of one independent class (ConcretObserver2) and the corresponding relationships (inherit) between this class and other ones in the original pattern.

3.2.3. Structural Integration: A design pattern may be composed with other patterns to solve multiple design problems in an application. The integration of two design patterns describes the particular ways that the two group of classes are combined, which may include strong chaining (connecting them by some relationships) or overlapping (overlap them at some classes). Such an integration may happen before or after the pattern instantiation process.

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,
then the integration of SC_1 and SC_2 denoted by:
$SC = (C, A, M, T, Ar, Pc, Pa)$ and a function ∇
∇: $C_1 \cup C_2 \rightarrow C$, $A_1 \cup A_2 \rightarrow A$, $T_1 \cup T_2 \rightarrow T$,
$Ar_1 \cup Ar_2 \rightarrow Ar$, $Pc_1 \cup Pc_2 \rightarrow Pc$, $Pa_1 \cup Pa_2 \rightarrow P$
with the constraint:
$C = \nabla C_1 \cup \nabla C_2 \,/\, \forall c \in C_1 \cup C_2$ and $\nabla c \in C_1 \cup C_2$

Let us consider another pattern (Mediator [6]), and its structural contract SC (mediator). The integration of a Mediator instance ($SC_{mediator}$) with the Observer instance ($SC_{observer}$) is shown in Figure 2.3.

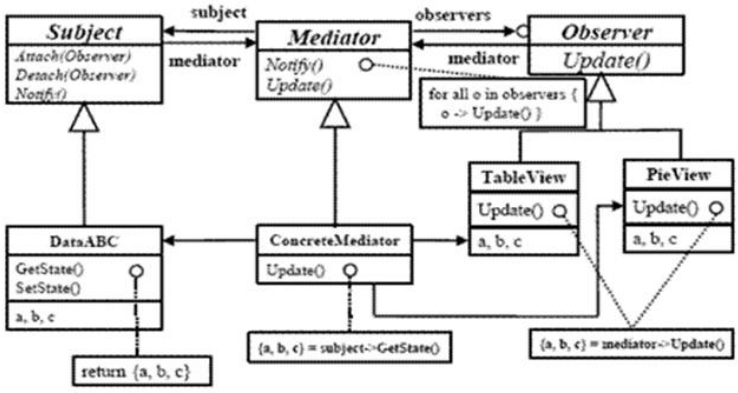

Fig 2.3 Integration of two Patterns (Mediator and Observer) [3]

In this composition, both subjects and observers play the role of Colleague in the Mediator pattern. Therefore, the structural contract for the integration is defined by

SC=(C, A, M, T, Ar, Pc,Pa), and the mapping functions ∇_1: $C_1 \rightarrow C$, ∇_2: $C_2 \rightarrow C$, **where:**

(0) Component-name is <ObserverMediator> **where:**
(1) ∃ **class** (Observer, Subject, TableView, PieView, DataABC, Mediator, ConcreteMediator) ∈C;
(2) ∧ ∃(attach, detach, GetState, update,notify, append, remove) ∈M;
(3) ∧ ∃ (void, datatype) ∈T;
(4) ∧∃ (subject,observers, Mediator, a, b,c) ∈ A;
(5) ∧∃ **Inherit** {(Observer, TableView) ∧(Observer,PicView)∧(Subject, ConcreteSubject) ∧ (Subject, DataABC)};
(6)∧ ∃ **Invoke**{(Subject, attach, observer, append) ∧ (Subject, detach, observer, remove) ∧ (Subject, notify,Mediator, update) ∧ (Mediator, notify, observer, update) ∧ (TablView, update,Mediator, update) ∧ (PieView, update,Mediator, update) ∧ (Mediator, update, subject, GetState) };
(7) ∧∃ **Return** {(DataABC, getstate, a) ∧(DataABC, getstate, b)∧ (DataABC, getstate, c)}
(8) **Where** ∃ Method{(attach, detach, notify) ∈Subject ∧(updtate) ∈Observer∧ (getstate,Setstate) ∈DataABC ∧ (notify, updtate) ∈ Mediator ∧ (updtate) ∈ConcreteMediator ∧ (updtate) ∈ PieView ∧ (updtate) ∈ TableView}

3.3. Interface Contracts

We define the interface aspect of a design component contract as follow:

Let a tuple IC = (P, IP,OP, IM,OM, IMI), where 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.

The abstract specification of the interface contract of Observer is done by:

(0) Component-name is Observer where:
(1) ∃ (aConcreteSubject,aConcreteObserver, anotherConcreteObserver) ϵ C
(2) ∧ ∃ (inOS, inSO,self, input) ∈IP
(3) ∧ ∃ (outOS, outSO, output) ∈OP
(4) ∧ ∃ (attach, detach, getstate, setstate,update, notify, change) ∈ IM
(5) ∧ ∃ (attach, detach, getstate, setstate, update, notify) ϵ OM
(6) ∧ ∃ (change) ϵ IMI

In order to be able to support a dynamic reconfiguration of the service and to provide a precise specification about the relationships of operations calls to each other, we include the constraints on component interfaces.

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:

$p \in IP(p) = \{i \in IP \setminus gate\_Ini = p\} \wedge$

$p \in OP(p) = \{i \in OP \setminus gate\_Outi = p\} \wedge$

$m \in IM(p) = \{i \in IP, m \in IM \setminus gate\_Ini\ ?m\} \wedge$

$m \in OM(p) = \{i \in OP, m \in OM \setminus gate\_Outi\ !m\} \wedge$

a ll-gateIN= { all IP(p)/ $p \in$ Component } $\wedge$

all-gateout={ all OP(p)/ $p \in$ Component } $\wedge$

Where Constraint /*constraints on gates*/:

$i,j \in 1,n \rightarrow$ gate_Ini $\neq$ gate_Inj $\wedge$

$i,j \in 1,n \rightarrow$ gate_Outi $\neq$ gate_Outj $\wedge$

$\forall j \in 1,n \rightarrow \exists\ i \in 1,n/$gate_Inj ?mi ϵ gate_Outi !m i $\wedge$

$\forall j \in 1,n \rightarrow \exists\ i \in 1,n/$ gate_Outj?mi ϵ gate_Ini !m

3.4. Behavioural Contracts

In contrast to the structural aspect of a design component contract, the behavioural contract describes the dynamic information, such as the collaboration among the objects participating in the component and the creation of new objects.

We have chosen a basic LOTOS (Fig 3) for defining a formal semantic model of behavioural contracts because it represents a powerful approach for modeling behaviour and concurrency. The choice of LOTOS is motivated by its powerful ability for describing behaviour and the availability of tools enabling formal verification and automatic generation of distributed programs. Our proposal focuses on formally describing architectures encompassing both the structural and behavioural viewpoints.

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,c,d]**|	Parallel composition with synchronization on several gates (b,c,d)	P[a,b,c,d,e]=P1[a,b,c,d] |[b,c,d]| P2[b,c,d,e]
hide b in |[b]|	Parallel composition with synchronization on gate (b)	P[a,c]= hide b in P1[a,b] |[b]| P2[b,c]
>>	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

Fig.3. Basic LOTOS Operators [1]

The LOTOS specification of the observer is as presented in the following:

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 anotherConcreteObserver[inbCO,outbCO] **:= noexit**
I; !setstate; ?update; !getstate
anotherConcreteObserver [inbCO, outbCO]
Endprocess

Endspec

4. Proposal for an Architecture Description Language

A key aspect of the design of any software system is its architecture. From a runtime perspective, an architecture description should provide a formal specification of the architecture in terms of components and connectors and how they are composed together. Enabling specification of dynamic architectures is a large challenge for an Architecture Description Language (ADL). This section describes LOTOS-ADL, our proposal ADL that

has been designed to address specification of structural and dynamic architectures. While most ADLs focus on defining software architectures from a structural viewpoint, our proposal LOTOS-ADL focuses on formally describing architectures encompassing both the structural and behavioural viewpoints.

From a runtime perspective, two viewpoints are frequently used in software architecture: the structural viewpoint and the behavioural one.

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

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

< structural viewpoint> := <component, connector, configuration>/

component := <cp1, cp2, ….., cpn> $n \geq 2$ and

connector := <ct_1, ct_2, ….., ct_m> $m \geq 1$

with **constraints:**

$\forall$cp1, cp2$\in$ component / name.cp1 $\neq$name.cp2

$\forall$ct1, ct2$\in$ connector / name.ct1 $\neq$name.ct2

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

<behavioural viewpoint>:= < LOTOS behavior expression >

Thereby, from a structural viewpoint, an architecture description should provide a formal specification of the architecture in terms of components and connectors and how they are composed together. Further, in the case of a dynamic architecture, it must provide a specification of how its components and connectors can change at runtime. 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.

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 to interact with others processes through externally observable actions.

We model a component as a black-box with a set of input and output gates (or channels), where visible events occur. Instead of describing the static functionalities that a component provides, we specify the set of (dynamic) behaviors that a component may exhibit in constituting a system. All the gates, together with constraints that may be imposed upon the ports, constitute the interface of a component. The interface of a component specifies the constraints on the way the component is to be used. A component may have overall constraints imposed upon the gate. The set of concepts that are manipulated are presented within our ADL meta-model (Fig. 4).

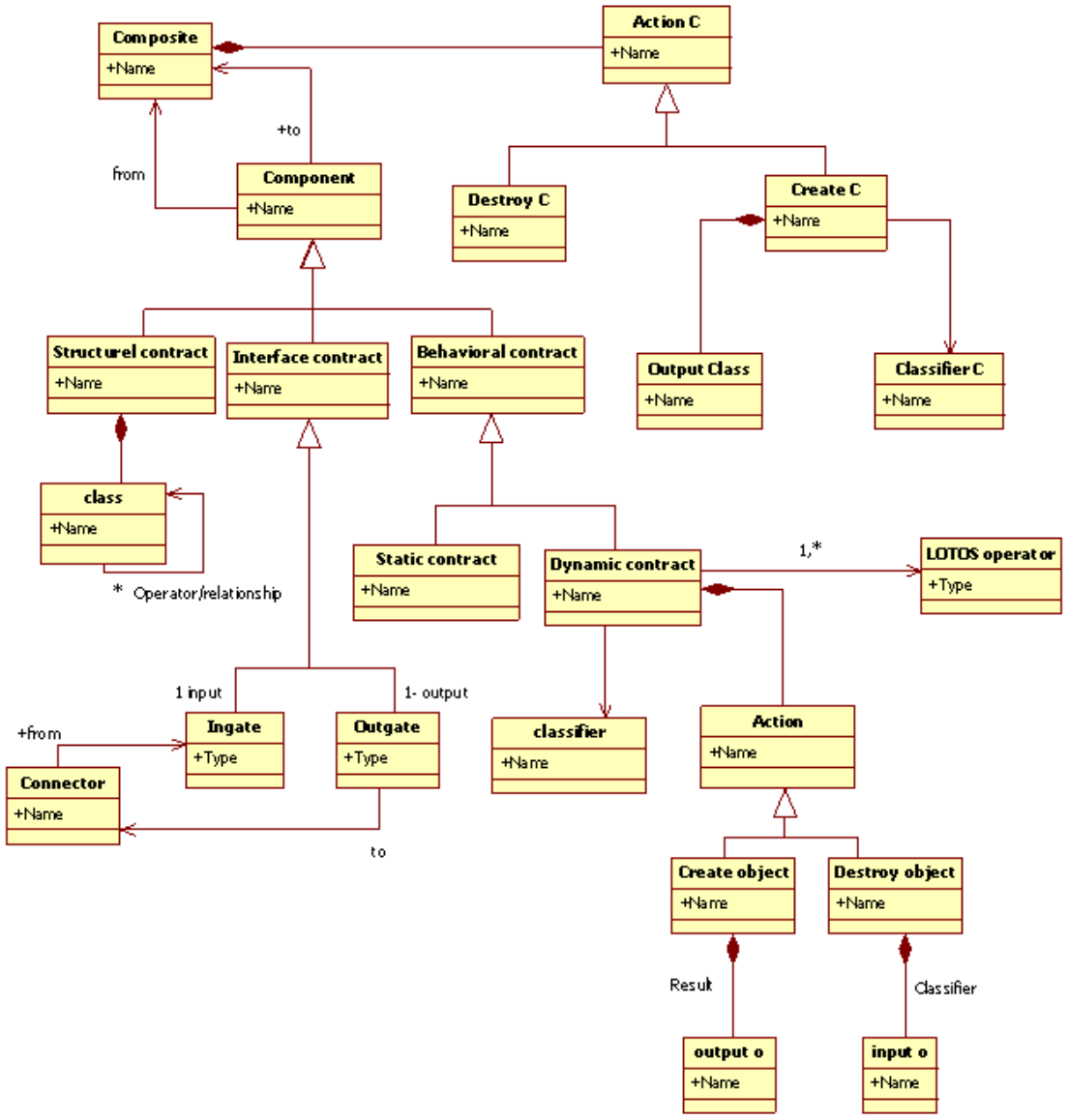

Fig.4. The LOTOS-ADL Meta-model

In our meta-model, we are mainly interested in representing static and dynamic behaviour contract using static and dynamic contract. A major benefit of separate static part from the dynamic part is that reasoning independently from any particular situations. The static contract of a component is a part that does not evolve. The evolution of a dynamic contract may have different purposes.

5. A Case Study: Client/server

Let us, consider the simple client-server system as shown in Fig5. It consists of one client and one server interacting via a link connector. Such a system is easy to describe in LOTOS-ADL. A LOTOS-ADL specification describes a system through a hierarchy of components (process). A process is an entity able to realise non-observable actions, and also interact with others process through externally observable actions.

The LOTOS specification at the top-level is a parallel composition of the process Client (component client), the process Server (component server) and the process connector (connector) (Fig.5). In order to specify this system, we adopt the following guiding [22]:

- The basic architecture elements, namely components and connectors, are modelled through the basic LOTOS abstraction, namely process.
- Any two LOTOS processes that model components must be in parallel composition with a LOTOS process defined as a connector
- The service specification consists of the temporal ordering of events executed at the service interface.
- We call to invocation (inv) those actions to activate the service and termination (ter) to the action of return a result.

5.1. Point to Point Connector

The LOTOS specification at the top-level is a parallel composition of the process Client (component client), the process Server (component server) and the process connector (connector) (Fig.5).

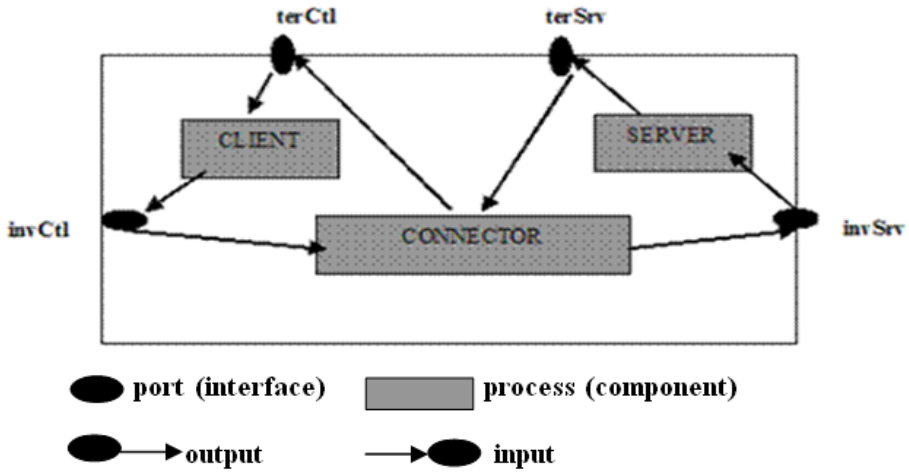

Fig. 5. Illustration of the Client-Server Specification

```
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
```

The connector behaviour is defined through the temporal ordering of invocation operations in the connector interface. The connector interface is made up of four ports: invCtl

to invocations from client, terCtl to returns to client, invSrv to invocations from server and terSrv to return to 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
```

In this case, the connector receives an invocation from the server that contains both the name of the requested service and the operation being requested on the server (invClt?s: SERVICE? Op: OPER). The connector passes both of them to the server and waits for the reply. Finally, the connector passes the reply containing the result to the client.

5.1. Multicast Connector

The connector abstract software architecture is defined as a collection of services. In order to specify the connector abstract software architecture, we assume that is composed by three components (Fig. 6) (service1, service2, service3) and a single connector (communication Service). The LOTOS specification of this software architecture is done by a parallel composition of the set of basic services and the process Communication Service.

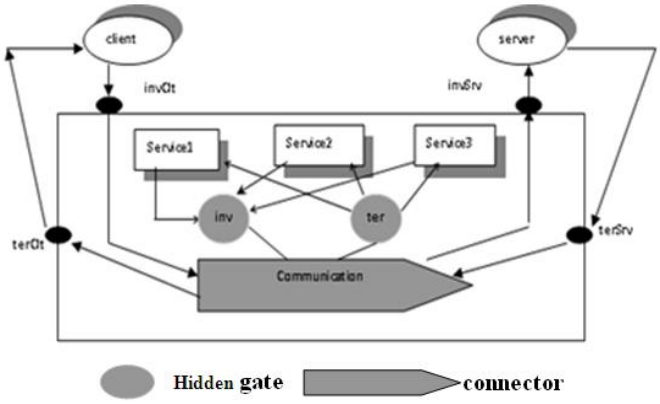

Fig. 6. Illustration of the Abstract Software Architecture

```
process Connector_Abstract [invClt,terClt,invSrv,terSrv] : noexit : =
 hide inv, ter in
  ((Service1 [inv, ter ] |||Service2 [inv, ter ] ||| Service3 [inv, ter ])
                    ||
    ServiceOrdering [inv, ter ])
              |[inv, ter ]|
   CommunicationService  [inv, ter, invClt,terClt,invSrv,terSrv]
 Where
  ……
  ……
Endproc
```

According to the constraints imposed by ServiceOrdering, after the request gets in the connector, it is passed to Service1 followed by Service2 and Service3. The LOTOS specification of the ServiceOrdering is done by:

Process ServiceOrdering [inv,ter] **: noexit : =**
inv ! Service1 ? op: OPER
ter ! Service1 ? r : RESULT
inv ! Service2 ? op: OPER
ter ! Service2 ? r : RESULT
inv ! Service3 ? op: OPER
ter ! Service3 ? r : RESULT
ServiceOrdering [invClt, terClt, invSrv,terSrv]

Endproc

5.3. Verification

For the verification of our approach, we use the FOCOVE (Formal Concurrency Verification Environment) [12] (Fig. 7). The FOCOVE environment is an integrated environment designed to edit Basic LOTOS behavior expressions which describe reactive systems and to generate and analyze Maximality based Labelled Transitions Systems structures (MLTS).

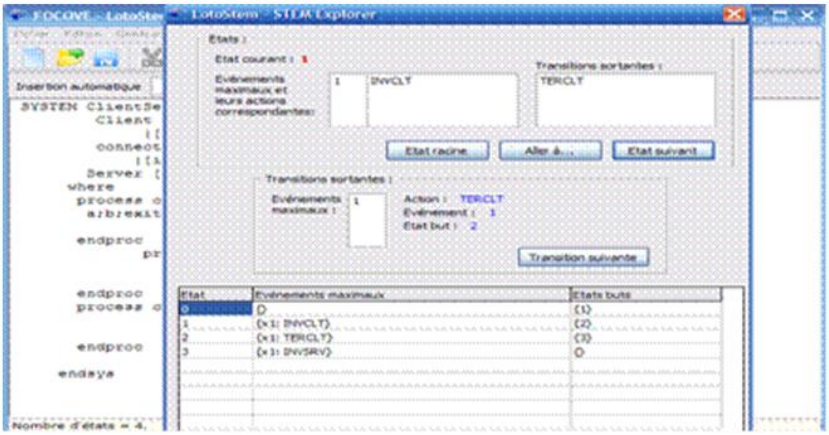

Fig. 7. The Environment of Verification

The FOCOVE environment is dedicated to the design and verification for component based software development. FOCOVE translates a LOTOS program into a Labelled Transition System (LTS for short) describing its exhaustive behaviour. This LTS can be represented either explicitly as a set of states and transitions or implicitly as a library of C functions allowing us to execute the program behaviour in a controlled way. By verification, we mean comparison of a complex system against a set of properties characterizing the intended functioning of the system (for instance, deadlock freedom, mutual exclusion, and so one.).

6. Conclusion

In this article, we have proposed of formal model of design component based on contract and a rigorous analysis approach to software design composition based on automated

verification techniques. Our approach allows us to find errors in the design composition early in the development process. This article has illustrated 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. It supports various levels of abstraction and provides several specification styles. These positive features encouraged us to adopt LOTOS as an ADL for describing both component and connector enables us to check behaviours properties. Finally, LOTOS specifications can also be used to express and verify concurrency models and real-time properties of systems.

The presented LOTOS specifications serve as a basis for very interesting future work. We are currently interested in the refinement of specifications in which the refinement process follows the rules of the software architecture refinement.

Also, we are investigating to propose a rule-based transformation enabling the mapping from LOTOS specification to JAVA pseudo code.

References

[1] L. Aprille, P. Saqui-sannes , C.Lohr ,"A new UML profile for reel-time system formal design and validation", in LNCS 2185, 2001
[2] T. Bolognesi, E. Brinksma. "Introduction to the ISO specification language LOTOS', In Van EIJK, 1989, pp 23-73,
[3] J. Dong. "Design component contracts", Phd thesis. Computer Science department, university of Waterloo, June 2002.
[4] J .Dong, S C. Paulo Alencar, D. Cowan., "Automating the analyse of design component contracts", In software Practice and Experience, 2005.
[5] H. Ehrig, B. Mahr,. "Fundamentals of Algebraic specification", volume1, Springer-verlag, Berlin., 1985
[6]. E. Gamma, R. Helm, R. Johnson, J. Vlissides, "Design Patterns, Elements of Reusable Object-Oriented Software", Addison-Wesley Longman. 1995
[7] R. K. Keller, R. Schauer, "Design Components: Towards Software Composition at the Design Level". *Proceedings of the 20th International Conference on Software Engineering*, 1998, pages 302–311.
[8] B. Meyer, "Applying design by contract", IEEE Computer, October 1992, pp 40-51
[9] N S Rosa, P R Cunha, 'A software architecture-based approach for formalising middleware behaviour", in ENTCS 2004.
[10] T. Taibi, D.C.L Ngo, "Modeling of distributed objects computing design patterns combination". Journal AMCS vol 13 N°2, 2004, pp 239-253.
[11] A. Zitouni., "Un framework pour l'utilisation des design patterns par intégration du langage de spécification LOTOS", Congré International en Informatique Appliquée CIIA05, Novembre 2005, BBA, Algérie, ISBN: 9947-0-1042-2
[12] Zitouni, A., Seinturier, L., Boufaida.,M., 2008, "Contract-based approach to analyze software components", International Conference on *Engineering of Complex Computer Systems* (ICECCS 2008/UML&AADL) workshop, Belfast, April 2008, pp 237, 242.
[13] A. Zitouni, M. Boufaïda, L. Seinturier, "Specifying Components With Compositional Patterns, LOTOS and design by contract". ISCA, 19th International Conference on Software Engineering and Data Engineering (SEDE-2010), San Francisco, California, USA, June 16 - 18, 2010, pp 190-195.

Authors

Dr. Abdelhafid Zitouni received his PhD in computer science in 2008 from the University Mentouri of Constantine, Algeria. Currently working as Assistant professor in Mentouri University of Constantine. His research interests include Software Engineering, Software Design, Software Reuse and Design pattern Detection.