Contract-based approach to analyze software components

Abdelhafid Zitouni, Lionel Seinturier, Mahmoud Boufaida

Contract-based approach to analyze software components

A. Zitouni,
Laboratory LIRE
Computer Science Department
Mentouri University of Constantine
Algeria,
ah_zitouni@yahoo.fr

L.Seinturier
LIFL-INRIA ADAM
University of Lille
59655 Villeneuve d'Ascq
France
Lionel.Seinturier@lifl.fr

M. Boufaida
Laboratory LIRE
Computer Science Department
Mentouri University of Constantine
Algeria,
boufaida@hotmail.com

Abstract

Component-based software development focuses on building large software systems by integrating existing software components to reduce cost, risk and time. However, behavioural and compositional conflicts among components constitute a crucial barrier to successful software composition. In this paper, we present a contract-based approach to analyze and model the properties of components and their composition in order to detect and correct composition errors. With this approach we characterize the structural, interface and behavioural aspects, and a specific form of evolution of these components. Enabling this, we propose the use of the LOTOS language as an Architecture Description Language (ADL) for formalising these aspects.

1. Introduction

Component-based approaches have been proposed to create and deploy software systems assembled from components. The use of previously developed components should leads to faster time 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 collaboration among the 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 [5] have been proposed to reify good design practice from conceptual design building blocks into a composable form.

Formal specification and verification techniques are useful for design analysis in that the formal representations are more precise, expressive, and unambiguous than the informal ones, such as graphical and textual notations. Formal notations can be a basis for verification techniques, such as model checking [2], which can be used to detect errors.

A contribution of this paper 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.

The approach includes a process of representing, instantiating and integrating design patterns which can seen as a components (called design components), and analyzing their compositions, which are captured as contracts. Compositional patterns can be formally described as connections between process components. The approach involves the modeling of design components and their composition, and a framework in which the design compositions can be analyzed. Another contribution of this paper is a proposition of a novel ADL (LOTOS-ADL) that has been designed to address specification of structural and dynamic architectures.

The rest of this paper is organised as follows. Section 2 introduces an overview of our approach. In Section 3 we focus on the abstract specification of the component. Section 4 presents the concepts of LOTOS-ADL. Section 5 gives an overview of our environment of validation. Section7 illustrate a case study. Finally the last section concludes the paper and gives directions for future work.

2. Overview of the Approach

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. The approach includes a process of representing, instantiating and integrating design components and analyzing their compositions, which are captured as contracts (figure1).

This approach allows 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. As shown in figure1, our approach begins by four steps: (The analysis, the selection, the 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 [11] we are shown the specification and the description of the system configuration and its components must be put into a form amenable for analysis and design [11].

-***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 consider the design problem that we want to solve and match it to the solution provided by general purpose design patterns (expert pattern [4] is a good candidate for this task) [11].

-***Abstract specification:*** this step (inspired from the work of Dong and al. [3]) contains a formal model of design component, called design component contract. A design component contract includes structural contract, behavioural contract and interface contract.

-***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. 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.

In this article we focus on the abstract specification of the component and the ADL for describing the 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 [5].

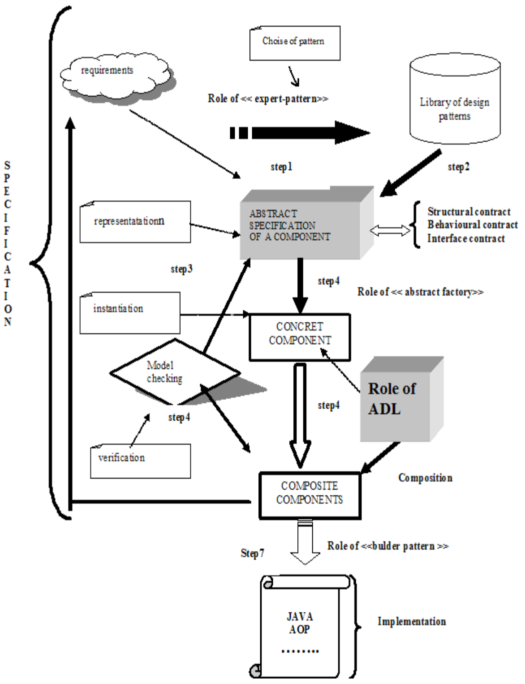


Figure 1. Overview of our approach

3. 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. Each contract defines the generic information about a design component. The instantiation operation can be used to apply a generic contract in a particular application. The integration operation formally defines how to compose two or more contracts to form a new contract.

The structural properties describe the relations of the constructs of each design component, such as connectivity of classes by inheritance or association relations in object-oriented systems. 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.

3.1. Structural contracts

We define the structural aspect of a design component contract as follows: 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 that define the participants within each 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 that are used to define the attributes and methods in classes ***C***, and ***Ar*** is a set of access rights that the attributes and methods can have in a class of ***C***. For example, ***Ar*** = {public, protected, private}, ***Pc*** is a set of connection predicates symbols that capture the relationships between the role each a design component. 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…)

The structural aspect of a design component can be formalized using a subset of First Order Logic (FOL), because the relations between pattern participants can be easily expressed as predicates [8].

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 [10].

We define two groups of predicates, entities (Table1) and relationships (Table2).

- 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.

Table 1. 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 set X

Table 2. Relationship predicates

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

3.2. 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. The behavioural contract is essential because the structural contract only captures the static information. But components are also characterized by the interactions among the objects and operations.

We have chosen a basic LOTOS [1] (figure 2) for defining a formal semantic model of behavioural contracts because it represents a powerful approach to modeling of 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. We illustrate our approach with the example of a client server application

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

Figure 2. LOTOS operators

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.

3.4. Interface contracts

Consider the structure (class and interaction diagram) of the observer pattern shown in figure 3 [10]: (The Observer (also called Publisher-Subscriber) regulates how a change in one object can be reflected in an unspecified number of dependant objects).

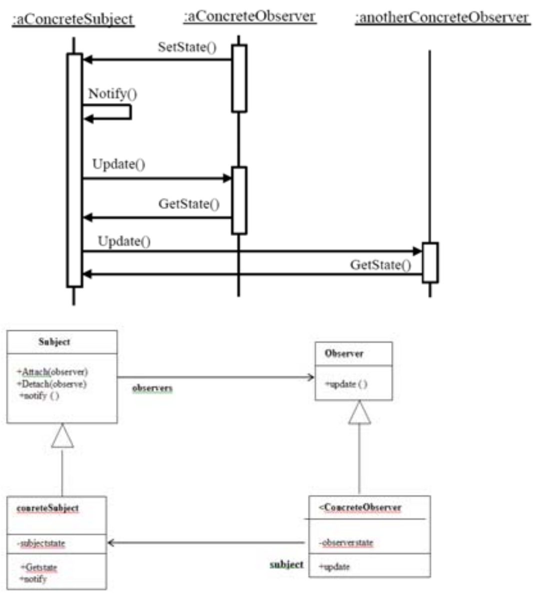


Figure 3. Observer pattern (class diagram ,interaction diagram)

The *abstract* specification of structural contract is done by:

(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)ϵSubject ∧ (updtate)ϵObservet ∧ (getstate, notify)ϵ ConcreteSubject ∧ (updtate)ϵ ConcreteObservet}*

The JAVA pseudo-code of this description is done by:

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) {………………} ; }

The abstract specification of interface contract is done by:

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

(1) *∃ (* aConcreteSubject,aConcreteObserver, anotherConcreteObserver) *ϵ C*

(2) *∧ ∃ (* inOS, inSO,self, input) *ϵ IP*

(3) *∧ ∃ (*outOS, outSO, output *)ϵ OP*

(4) *∧ ∃ (*attach, detach, getstate, setstate,update, notify, change *)ϵ IM*

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

(6) *∧ ∃ (*change*)ϵ IMI*

The behavioural aspect of a design component is expressed by the LOTOS specification. This specification describe the sequences of observable event:{attach,detach,getstate,setstate,update,notify,change}exchanged by elements of components: {ConcreteSubject, ConcreteObserver} by means of a restricted set of operators (figure 2).

4. Architecture Description Language

A key aspect of the design of any software system is its architecture. An architecture description, from a runtime perspective, 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) [9]. This section describes LOTOS-ADL, a novel ADL that has been designed to address specification of structural and dynamic architectures.

It is a formal language based on temporal ordering of observational behaviour. While most ADLs focus on describing software architectures from a structural viewpoint, LOTOS-ADL focuses on formally describing architectures encompassing both the structural and behavioural viewpoints. The LOTOS-ADL design principles, concepts and notation are presented. An architecture description specifies architecture. Architecture can be described according to different viewpoints. From a runtime perspective, two viewpoints are frequently used in software architecture [7], [9]: the structural viewpoint and the behavioural viewpoint.

The structural viewpoint may be specified in terms of:

- components (units of computation of a system),
- connectors (interconnections among components for supporting their interactions),
- Configurations of components and connectors.

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 or mobile architecture, it must provide a specification of how its components and connectors can change or move at runtime. The behavioural viewpoint may be specified in terms of:

- actions a system executes or participates in,
- relations among actions to specify behaviours,
- behaviours of components and connectors, and how they interact.

A large challenge for an Architecture Description Language (ADL) is the ability to describe static but also dynamic software architectures from structural and behavioural viewpoints.

Indeed, for describing dynamic architectures, an ADL must be able to describe changing structures and behaviours of components and connectors (including creation/ deletion/reconfiguration/ moving) at runtime.

The set of concepts that manipulated within our ADL meta model. (We are defined a meta-model for these tasks).

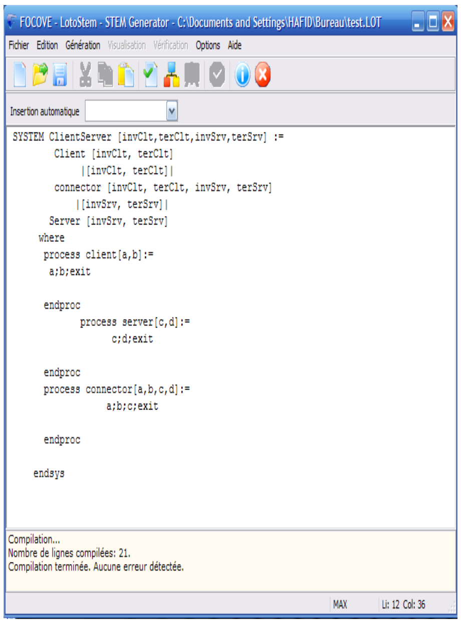
In our meta-model, we are mainly interested in representing static and dynamic behaviour contract using static and dynamic contract. A major benefit of separate static part from the dynamic part is that reasoning independently from any particular situations. The static contract of a component is a part that does

5. Validation

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, etc.). Most of the verification algorithms are based on the labelled transition systems model, which consists of a set of states, an initial state, and a transition relation between states. This model is often generated automatically from high level descriptions of the system under study, and then compared against the system properties using various decision procedures. For the verification of our approach, we use the FOCOVE (Formal Concurrency Verification Environment) (available in www.focove.new.fr) (fig 4)

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

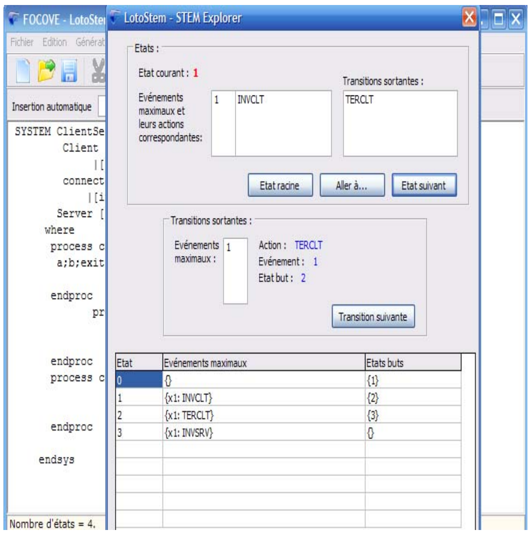
For the verification of concurrent systems, FOCOVE allows errors the use interleaving semantics or the maximality based semantic.


FOCOVE - LotoStem - STEM Generator - C:\Documents and Settings\HAFID\Bureau\test.LOT
Fichier Edition Génération Visualisation Vérification Options Aide
Insertion automatique

```
SYSTEM ClientServer [invClt,terClt,invSrv,terSrv] :=
        Client [invClt, terClt]
             |[invClt, terClt]|
        connector [invClt, terClt, invSrv, terSrv]
            |[invSrv, terSrv]|
       Server [invSrv, terSrv]
     where
      process client[a,b]:=
       a;b;exit

      endproc
             process server[c,d]:=
                   c;d;exit

      endproc
      process connector[a,b,c,d]:=
                  a;b;c;exit

      endproc

    endsys
```

```
Compilation...
Nombre de lignes compilées: 21.
Compilation terminée. Aucune erreur détectée.
```

Figure 4.1. The FOCOVE environment



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}	{}

Nombre d'états = 4.

Figure 4. 2. Generation of the label transition systems

6. Case study

Consider the simple client-server system shown in Figure5. It consists of one client and one server interacting via 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) (figure 5).

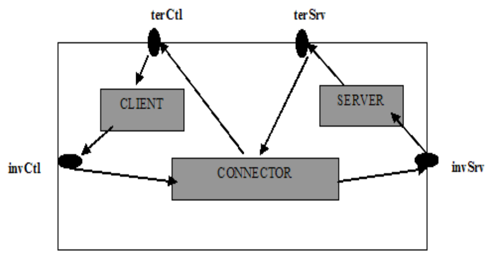


Figure 5. Illustration of the Client-Server specification

```
 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
```

7. Conclusion

In this paper, we have introduced a proposition 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 paper 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.
Currently, we are investigating to proposing a rules-based transformation enabling the mapping from LOTOS specification to JAVA pseudo code.

References

[1] T.Bolognesi, E.Brinksma. Introduction to the ISO specification language LOTOS . In [Van EIJK89] 23-73
[2] Edmund Clarke, Orna Grumberg, and Doron A. Peled.: Model Checking. MIT Press, 1999.
[3] Jing Dong.: Design component contracts, Phd thesis. Computer Science department, university of Waterloo, June 2002.
[4] Dong J, Paulo S C Alencar, Donald D Cowan.: Automating the analyse of design component contracts, In software Practice and Experience, 2005.
[5] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides.: Design Patterns, Elements of Reusable Object-Oriented Software, 1995, Addison-Wesley Longman.
[6] Darlan Garlan, Robert T. Monroe, and David Wile.: Acme: Architectural Description of Component-Based Systems. Foundations of Component-Based Systems, (Leavens G. and Sitaraman M., eds.) Cambridge University Press, pages 47–67, 2000.
[7] IEEE recommended Practice for Architectural description of Software-intensive System, October 2000.
[8]: L. Lamport The temporal logic of actions. ACM Trans. Programm. Lang. Syst., Vol. 16, No. 3, pp. 872-923.
[9]. F. Oquendo.: PI-ADL: An architecture description language based on the higher-order typed pi-calculus for specifying dynamic and mobile software architectures. ACM Software Engineering Notes, volume29, Number04, May 2004
[10]: Taibi T. and Ngo D.C.L (2001).: Modeling of distributed objects computing design patterns combination. Journal AMCS vol 13 N° 2 pp 239-253, 2004
[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