Dynamic Maintenance and Evolution of Critical Components-Based Software Using Multi Agent Systems

Chouarfia Abdallah & Hafida Bouziane
University of Sciences and Technology Oran Mohamed Boudiaf
Computer Science Department
Oran, Algeria
E-mail: chouarfia@univ-usto.dz, h_bouziane@univ-usto.dz

Received: May 10, 2011 Accepted: June 26, 2011 doi:10.5539/cis.v4n5p78

Abstract

Component-based development has become a commonly used technique for building complex software systems by composing a set of existing components. In general adapting an application means stopping the application and restarting it after the adaptation. This approach is not suitable for a large classes of software systems in which continuous availability is a critical requirement, hence the need of adapting dynamically the application at runtime. This paper presents an architecture based approach for dynamic adaptation in critical components based software using multi agent system.To achieve this, we use an agent based system to perform the adaptation. The agent system is guided by an architectural description. The adaptation mechanism is implemented within the connectors using the flexibility offered by the Java script language techniques. The script language Groovy is used. The evaluation is made by comparing the execution time before and after the adaptation mechanism.

The paper is structured as follows: section 2 presents related works to dynamic adaptation. Section 3 describes the proposed solution to achieve a dynamic update of components-based software applications. The implementation details and some measurements relative to our solution are given in section 4. Section 5 concludes and presents some perspectives.

Keywords: Components, Dynamic adaptation, Multi agent system, Architecture description, Maintenance and evolution, Critical components based software

1. Introduction

A components-based development [Szyperski, C., 1998] aims to the development of large software by assembling existing components. The continuous evolution of user's needs and the fast changing in the execution environments make the software adaptation a primordial task. In components-based development paradigm [Aksit, M. and Choukair, Z., 2003], an adaptation can address: *Architectural changes, geographical changes*, *Interface modification*, or *implementation adaptation*. The architectural changes consist of adding or removing components or modifying connections among them. The geographical changes correspond to the migration of components from one site to another. The interface modification consists of changing the interface of a component to make it more compliant to the caller's expectations, while the implementation adaptation affects the internal implementation of components without changing its interfaces.

Traditionally, an application is stopped to be adapted. This approach is not suitable for critical systems that have a high level of availability like banks, Internet or telecommunication services. In these systems the adaptation must take place at runtime and the application should not be entirely stopped. Unfortunately, such adaptation is not trivial; there are several conditions and constraints to be satisfied, and this leads to many problems to overcome. Some important issues to be, in order, made to the dynamic update are [Aksit, M. and Choukair, Z., 2003]:

- *Maintaining application consistency*: Components states must not be affected by changes the application architecture.
- *Preserving components bindings*: Bindings have to be preserved by redirecting the calls to new components and managing transient states.
- *Initializing new components*: New components must be initialized with adequate internal state according to the former component.
- *Preserving communication channels* by avoiding message loss, duplication or excessive delays.

In this paper, we describe our approach to achieve dynamic adaptation of components-based software applications. The idea is to introduce between two components a connector (unit of interaction) [Kell, S., 2007] that intercept and redirect inter components communications. The adaptation is made through a so called adaptation system by acting on connectors. The adaptation system is part from the application, and it consists of agents system, whose role is to perform and supervise the adaptation operations. The agents system is guided by a knowledge base, with:

i) the architectural description of the running application,
ii) a set of reference rules defining the adaptation policy,
iii) a set of coherence rules ensuring the validity and the coherence of adaptation process.

The evaluation is made by comparing the execution time of the initial application (prior to the adaptation) with the execution time after the adaptation mechanism.

We have tested the adaptation mechanism influence on the application response time and the adaptation time. According to the results, we noticed that the influence on the response time is stable and that overhead time is about 16 %. The adaptation time average is greater than three (03) seconds. Of course, this figure is very large compared to the response time of one request which is approximately 14ms. However, this is only true while the adaptation process is taking place. A second experimentation aims to evaluate the ability of the system to preserve the communication channels during adaptation. The obtained results show that the rate of altered messages, during an adaptation, is null. According to the previous results, we can state that the proposed approach is well suited for applications, with a low adaptation rate, and in which loss of performance is preferred than the loss of data.

2. Related works

Several works dealt with dynamic adaptation problems, hence the emergence of several approaches.

In *the model driven approach*, the dynamic adaptation is based on a components model that designed to support this kind of adaptation. DCUP (*Dynamic Component Updating)* [Plasil, F. et al., 1997] is an example of this approach. In DCUP the component is divided into two parts: *permanent part* and *replaceable part.* Adapting a component means replacing its replaceable part by a new version at run-time.

In *the reflexive approach*, an application has an abstract level (meta-level) that reify the real system. The adaptation is made first on the meta-level, after that, the changes are reflected on the executed applications thanks to the causal connection between the meta-level and the real system. DYVA [Ketfi, A., 2004]: is an example of reflexive framework for dynamic reconfiguration of components-based applications. The framework is decomposed in two main parts: The base-level represents the concrete application that provides the expected functionalities and its execution environment and the reconfiguration machine that contains

i) the different operational modules responsible for achieving the reconfiguration,
ii) the meta-level which represents the materialization of the concrete application.

In *the architectural approach,* it uses explicit description of the executed application through specific languages: the ADLs [Medidovic, N. and Taylor, R.N., 2000] (Architecture Description Languages). An ADL describes an application in term of components, connectors and connection among them. The adaptation in this approach is verified and validated in the architectural level before to be applied on the application. This approach is used by [Qun Yang et al., 2006]; the adaptation takes advantage of both meta-architecture and the mobile agents. It uses an architectural model to guarantee the safety of the reconfiguration, while using mobile agents to automate the adaptation process in a flexible way.

In *the flexible middleware approach*, the adaptation is delegated to the execution platform. In such system the dynamic adaptation is looked like a non functional properties offered by the middleware, like security and transactions management. The adaptation in [Brinkschulte, U., Schneider, E. and Picioroag, F., 2005] is based on the real time middleware OSA+ [Brinkschulte, U. et al., 2002]. The objective is to be able to reconfigure services during run-time, with a predictable and predefined blackout time (the time where the system does not react due to the reconfiguration).

The aspect oriented approach, which adaptation is based on the aspect oriented programming techniques [Kiczales, G. et al., 1997]; in particular the dynamic aspect involves plugging and unplugging aspects without stopping, and restarting a running system. DAOP [Pinto, M. Et al., 2003] (Dynamic Aspect-Oriented Platform) is an example of such system. DAOP provides a composition mechanism that plugs aspects into components dynamically at runtime. The composition between components and aspects is established during runtime interaction and is governed by a set of plug-compatibility rules in order to guarantee their correct interoperation.

Authors in [Oreizy, P. et al., 1998] describe Arch Studio, a tool that implements an architecture-based approach to runtime software evolution. The approach is based on an explicit architectural model, which is deployed with the system and used as a basis for change. The connectors in the system are first class elements that have an important role to support run-time changes. An imperative language is used for modifying architectures. The tool supports adding, removing and replacing components and connectors, and changing the architectural topology. On the same axis [John, C., et al., 2004] presents an architecture-centric approach to self-adaptive software applied to systems constructed using independent components interconnected through first-class connectors, both explicitly modelled using architectural descriptions. The architectural models are used as the basis for the decomposition. The architectural models representation is make using xADL 2.0 language [Dashofy, E.M., et al., 2001]; a highly extensible, XML-based ADL. An extension of this language is also used to define the structure of observations, responses, and adaptation policies.

In [Qun Yang et al., 2006], the adaptation takes advantage of both meta-architecture and the mobile agents. It uses an architectural model to guarantee the safety of the reconfiguration, while uses mobile agents to automate the adaptation process in a flexible way.

[Huang, G., et al., 2006] presents an approach to recover software architecture from component based systems at runtime and changing the runtime systems via manipulating the recovered software architecture. As soon as software architecture is recovered, the runtime system can be observed, reasoned and adapted through its architecture views. The approach supports the adding, deletion and replacement of the components and connectors.

As a conclusion each one of the cited approaches has its advantages and drawbacks. Our approach is based on a Multi Agent System (MAS) which is justified by the facts that:

- The adaptation of s distributed application needs distributed tools; this is the case of MAS,

- MAS is mature with stable and efficient platforms, and characteristics like communication, scalability, mobility, and so on…

3. Proposed System

The architecture of the proposed system [Belabed, A. and Chouarfia. A., 2008] consists of two main parts: the *Knowledge Base* and the *Multi Agent System: MAS* (Figure 1):

3.1 Knowledge Base: KB

The KB consists of two parts: the adaptation policy and the architecture description. The formalism used in this base is based on the logic of predicates; we used Prolog. The choice of this formalism is justified by:

- The important role played by the architecture description in the adaptation mechanisms; as it involves a large number of inferences made on the architecture description to guide the adaptation. Such mechanism is provided in Prolog.
- Prolog¦ can be easily used as descriptive language. A predicate which presents a fact is similar to an ¦XML tag¦, for example: < tag > value < tag >, can be written in P¦PPProlog as¦ tag (value). More Moreover, ¦Prolog¦ is used in several projects to describe more complex structures such as ontology [Samuel, K., et al., 2006]; this motivated us to use it as an architecture description language.
- The existing tools facilitate the use and the integration of Prolog formalism with other languages (such as java/Prolog interface), hence eliminating the reprogramming of the necessary inferences mechanisms needed in our approach.

3.1.1 Adaptation Policy

The adaptation policy is a set of rules that guide the adaptation trigger according to the values of particular environment variables. The rules are in the following form:

If <event> Then **<**action>

An event can be a significant change in one of the environment variable like memory usage, network bandwidth saturation, and so on. The following example shows the form of a rule which specifies the trigger of an adaptation if the memory exceeds a given value (**val**).

*Event ('Context', Value)***:-** *Value >* ***val,*** *Replace (compo1, compo2).*

The predicate Event has two parameters: the first specifies the type of context concerned by the event; and the second the value of the event. The action "Replace (*compo1*, *compo2*)" will be triggered only if the condition "Value > val" is verified.

If a new significant event occurred while a component is running, the adaptation is postponed until the execution of the component terminates. We did not investigate yet the interrupt of a running component, and then make the adaptation and resuming the component adaptation.

3.1.2 Architecture Description

The description contains:

- The detailed specification of each component in the component's database in terms of provided and required interfaces along with their operations.
- The architecture description of the running application (components/connectors and interactions).
- The inference rules used to extract the correspondence and compatibility between components.
- The set of rules used to ensure the adaptation coherence and finally validate it.

a- Component description

Component ('name', 'id', 'state') **(1)**

State: passive or active

A component is in passive state (cannot accept incoming calls) while it's directly implied in a task of adaptation.

Required-interface ('component_id', 'interface_id') **(2)**
Provided-interface ('component_id', 'interface_id') **(3)**
Interface ('name', 'id') **(4)**
Include-operation ('interface_id','operation_id') **(5)**
Operation ('name','id','return_type', param_number,'abstract') **(6)**
Param ('operation_id', 'type', rank) **(7)**
Source_component ('source_file','address_file') **(8)**
State_component ('component_id', 'state') **(9)**

b- Connector description

Each connector depends on a component for which it implements the required interfaces. The description is, herein, limited to the dependency component/connector and deployment server address.

Connector ('name', 'id') **(10)**
Dependence ('connector_id','component_id','interface_id') **(11)**
Deploy_address ('connector_id','server_address') **(12)**

c- Interaction between components

Interact ('component_id1, 'component_id2', 'operation_id1', 'operation_id2','connector_id') **(13)**

The predicate **Interact** specifies that the component_id1 and component_id2 are interacting trough the connector_id. The operation_id1 is required by component_id1 and the operation_id2 is provided by component_id2.

d- Inference rules

The knowledge database is enhanced by a set of inference rules used to achieve the adaptation task.

Example of inference rule:

Relation ('component_name', 'connector_name', 'operation_name') **(14)**
Component ('name', 'id', 'state')
Connector ('name', 'id')
Interact ('component_id1, 'component_id', 'operation_id1', 'operation_id2', 'connector_id')

This relation will provide all the couples (connector, operation) in connection with the component_id1. Such relation is used to localize all the connectors which call component_id1 in order to block their messages towards it during an adaptation task.

e- Coherence rules

Like inference rules, the knowledge database is enhanced by two sets of coherence rules. The first set is used to ensure the knowledge database coherence, while the second is used to ensure the application coherence before to validation of the adaptation task.

Knowledge database coherence rules:

Example: 'Two components don't have the same component_id' is expressed as

Coherence_id (component_name1, component_name2):-
Component_name1 != component_name2,
Component (component_name1, component_id1, 'state')
Component (component_name2, component_id2, 'state')
Component_id1=component_id2 **(15)**

Incoherence is detected if the execution of this predicate ends succeeded. The rule (1) is used to make the link between the component_name and its component_id.

Application coherence rules:

Example: "Detection of cycle in the call of operations" is expressed as

Connect (X, Y):- **Interact** (_, _, X, Y, _) **(16)**

Connect (X, Y):- **Interact** (_, _, X, Z, _), **Interact** (_, _, Z, Y, _) **(17)**

Cycle (Op)**:- connect** (Op, Op) **(18)**

The connect predicate defines a connexion between two operations X and Y (X calls Y). This relation is established if the rule (13) exists and it includes the operations X and Y. The rule (17) specifies that the relation is verified even if the call is not direct. A cycle is detected if an operation calls itself.

3.2 Multi Agent System MAS

- As already stated, the use of MAS is justified by the facts that it is a distributed tool in a mature phase along with stable and efficient platform.

MAS (Figure 2) contains two agents: an *adaptation agent A-A* and an *environment agents E-A*.

3.2.1 Adaptation Agent A-A

Its role is to:

- Take decisions about the adaptation triggering, according to the notifications provided by the E-A. The A-A uses the adaptation policy rules to accomplish this task.
- Perform the adaptation according to the actions deduced from the adaptation policy rules. The adaptation operations are guided by the architecture description base.
- Modify the architecture description according to the changes made on the application level.

The A-A makes the above operations through three managers: *the rules manager*, *the adaptation manager* and *the architecture manager*.

The rules manager is responsible for the manipulation of the adaptation policy rules. If a decision to adapt is taken, the rules manager informs the adaptation manager to do so. This operation is guided by the architecture manager which makes the necessary inferences from the architecture description base. Subsequently, after each adaptation the architecture manager reflects the changes upon the architecture description in order to ensure a matching between the running application and its description.

3.2.2: Environment Agent E-A:

Its role is to control the execution environment and to notify the A-A, in case of any significant changes in the environment variables [Belabed, A. and Chouarfia. A., 2008].

3.3 Adaptation Principle

The idea behind the approach is to associate a connector to each component, which implements its required interfaces. The inter components calls are done through these connectors.

The adaptation mechanism acts directly on connectors to achieve the adaptation (Figure 3). Each connector implements the necessary mechanisms (calls and redirection) that allow the adaptation agent to perform the update.

Using Figure 3, we explain the structure of connectors in our approach. In the following example, we assume that the C1 component interacts with the C2 component through a connector in a synchronous communication mode.

The C1 component requires interface I1, the specification of which is:

```
Interface I1 {
                    Result1 Op1 (param1.1, param1.2);
            }
```

Where "Result 1" is the return of operation Op1, with param1.1 and param1.2 as input parameters.

The C2 component provides the interface I2 as:

```
Interface I2 {
                    Result2 Op2 (param2.1, param2.2);
}
```

The basic structure for the connector is as follows:

```
Connector implements I1 {
        Result1 Op1 (param1.1, param1.2)
{
Return execute ('Script');
}
}
```

The script code

```
// type casting of parameters call
Param31= castingToPramr2.1(param1.1) ;
Param32= castingToPramr2.2 (param1.2) ;
// call of C2 component's Op2 method
Result3 = C2.Op2 (Param3.1, Param3.2);
// Result type casting
Return cast_to_Result (Result3);
```

The body of Operation Op1 of the connector is implemented with a script that makes the call to Operation Op2 of C2 component, with the necessary parameters and return types casting. The replacement of the C2 component is made by changing the executed script in the body of connector's operation (Op1 in the example).

The system is designed to support the adding, removing, replacement and migration (change of the deployment server) of components. The A-A operates according to an adaptation scheme, this is a set of basic algorithms specific to each operation (ex: adding or removing operation).

Before any operation, the components directly implied in the adaptation operation must be in a passive state, this is possible thanks to the connectors that can queue calls to a component C during adaptation.

A dynamic connection between two components C1 and C2 is to put in interaction these two components through two operations, usually called methods, M1 and M2. M1 is the required method for C1 and M2 is the provided method by C2. The connection is done by the A-A while indicating to the connector associated with C1 the name of C2 component and the method to be called M2. A mapping phase between the two methods is necessary before establishing the connection. The correspondence concerns the call parameters, their order and the return type of each method. The mapping is done manually by the administrator of the application using the architecture description. The administrator must provide the methods of calculation (script code) for conversion between the types of the call parameters and the return type of each method. The methods are then transmitted to the connector responsible for the connection between the two components. To make conversions, in an automatic way, after a connection, the administrator must also manage the semantic correspondence between the methods to be connected to avoid erroneous semantic uses.

The dynamic disconnection of a component consists of putting it in a passive state.

3.3.1 Adding component

Adding a new component to an application means connecting it with other components of the application. So the addition of component is a succession of connections. The addition is done as follow:

```
Add (component C) {
For each port m of c {
- Identify all component Ci and ports mi to be connected with the port m of component C;
- For each couple (Ci, mi) {
    make the correspondences with the couple (C,m) ;
If not correspondence Then cancel adding; }}
- establish the add at the architectural level;
- verify the adaptation coherence after adding (in architectural level)
If not coherence
      Then {cancel adding;
            Cancel the modifications on architectural level;  }
- make adding in the application level;
                  (make connection (Ci,mi)->(Cj,mj) after deploy C)
 }
```

Listing 1: Component adding code

3.3.2 Removing component

A component is removed only if it does not refer to any component and no component refer to it. This condition can be verified at the architecture level when the component is not implied in any interaction rule. The removing algorithm is defined as follow:

```
remove (component C) {
Check the absence of reference to or from C;
      If  Not reference Then remove C;
      else impossible to remove C;
                                    }
```

Listing 2: Component removing code

3.3.3 Component Replacement

Figure 4 shows the replacement of the component *C3* by the component *NewC3*. In the example, the *C3* component is in relation with the two components *C1* and *C2* through the provided interfaces and in relation with *C4* through the required interface.

 ISSN 1913-8989 E-ISSN 1913-8997

The adaptation consists then to replace the *C3* component by the *NewC3* component. Before starting the replacement operation the two components must be passed in a correspondence phase.

After the corresponding phase, the adaptation agent performs the replacement according to the following adaptation order:

1. Referring to the architecture description, the adaptation agent localises all connectors in relation (in provided interfaces) with the component to be replaced (« *con 1.3* » and « *con 2.3* » in the example).
2. The adaptation agent puts *C3* in passive state by giving instructions to each localised connector to block all incoming messages towards it.
3. The adaptation agent deploys the new component.
4. The adaptation agent sends the necessary information to each connector to make the redirection of the calls towards the new component.
5. The adaptation agent connects the new component on the provided interfaces with the C4 component, this operation implies the deployment of the connector associated to the *NewC3* component.
6. The adaptation agent makes a state transfer between the two components if this implies.
7. After a time *t* corresponding to the maximum of the response times of the component to be replaced, the adaptation agent deactivates this last one then activates the new component by releasing the blocked messages and announces the end of the adaptation process. This mechanism is used to make sure that the C3 component has finished all its current treatments before its suppression.

3.3.4 Component Migration

The migration consists of moving a component from an application server *S* to another server *NewS*. The new server must provide an execution environment for the moved component, i.e. it must have all the resources which the component needs. The migration of a C component towards a *NewS* is made by the A-A according to the following script:

```
Migrate (component C, server NewS) {
- deploy a copy of C on the NewS;
- locate all connectors referring C;
- make C in passive state (no incoming calls);
- send the address of the NewS to the localised connectors to locate the migrated component;
- If C is with state Then {
        - Waits a t time >= max (response time of C);
        - make a state transfer between C and its copy;
```

Listing 3: Component migration code

3.4 Adaptation Coherence

The adaptation coherence is ensured at the architectural level. Before establishing an adaptation at the executing level, a prior adaptation is made on the architectural level. The coherence checking is done through a set of rules incorporated in the architecture description base. Any incoherent adaptation incoherent at this level is cancelled. The following section describes the solutions used in our approach to ensure a coherent and safe adaptation.

1. *Preserving communication channels***:** the communication channels are preserved through a mechanism implemented in each connector. This mechanism suspends the incoming messages during the adaptation process. The end of adaptation resumes all the suspended messages, and the application continues its execution without loss of messages.

2. *Coherence of the interactions:* the coherence of the interactions between components is ensured through a manual a mapping phase between the components ports of the components to connect. This phase is performed using information provided for each port in the architecture description base. This information allows us to take into account the semantics of use of each port and component, hence avoiding any conflict when calling operations.
3. *Conflicts between adaptations:* to avoid any conflict between adaptations, we do allow only one adaptation at a time, i.e. there are no concurrent or simultaneous adaptations. This leads to a total crash of the application. In case of many adaptations, we "serialize" the adaptation processes and we serve them in first in first out order. However, if we notice a conflict between adaptations, within the queue, we perform only the last one.
4. *The state transfer:* we do not introduce a specific solution in our approach; we adopt the solution proposed by [Ketfi, A., 2004]. This solution is specific to the components written in java. Each component before, its deployment, is instrumented using a byte code manipulation tool such as Javassist [Chiba, S. and Nishizawa, M., 2003] or BECEL [Dahm, M., 1999]. In this phase we inspect the component implementation then select the attributes which constitute the state of component, we add then the operations *getStat ()* and *setState()* in the component implementation. This mechanism increases the complexity of the operation of adaptation, for example, in component replacement it is also necessary to make a correspondence between the two components to see whether it is possible to make a state transfer. If the mapping fails, the replacement will then be impossible.

4. Implementation and Evaluation

A preliminary implementation of the system is done for the EJB (Enterprise Java Bean: Sun Micro Systems) component model. For implementing the interception and redirection mechanisms in the connectors, we have used the *reflexives* properties of Java language and the *Java scripting* programming technique [Bosanac, D., 2007]. This technique consists of executing a script code in a java class. The script origin can be a file or other application. This technique enables the adaptation agent to change the connectors' code dynamically. The script language used in the implementation is Groovy [Bordeie, X., 2007].

To evaluate the proposed system, the test application runs on a PC with Pentium IV 1.7MHz, 256M SDRAM. It consists of a component client sending a string and component server receiving and printing it on the screen before returning it back to the client.

The evaluation test is made by comparing two versions of the same application; one implements the adaptation mechanism on its connectors, the other one without this mechanism.

First, we have tested the adaptation mechanism influence on the application response time.The objective was to calculate the response time increase in the version which implements the adaptation mechanism in relation with the number of requests sent by the client. The Table 1 shows the response times before and after the adaptation mechanism.

According to the results, we notice that the influence of the adaptation mechanism is at least stable; it induces a response time increase of about 16%. This increase is acceptable because the use of an interpreted language in connector code and will be less if a compiled version. We cannot say that is acceptable or not, as it depends on the nature of the application and whether we prefer dynamic adaptability more than performance or not.Thus, there is always a trade off between the dynamic adaptability and performance.

The second evaluation is to measure the adaptation executing time, which is also the inactivity time, since the communication channels are blocked during adaptation. The adaptation time is calculated as follow:

$$\mathbf{T}_{adaptation} = \mathbf{T}_{rspt\text{-}wa} - \mathbf{T}_{rspt\text{-}na}$$

With: $\mathbf{T}_{adaptation}$ **:** adaptation time.

$\mathbf{T}_{rspt\text{-}wa}$**:** response time including adaptation.

$\mathbf{T}_{rspt\text{-}na}$**:** response time without adaptation.

According to the results shown in Table 2, we observe that the adaptation time average is greater than 3 seconds. This time is very large compared to the response time of only one request which is approximately 14 ms (response time/a number of requests). This value presents an unacceptable inactivity rate, especially if we manipulate a highly critical or real time application.

The last experimentation, Table 3, aims to evaluate the ability of the system to preserve the communication channels during adaptation. For this reason, we have implemented on the client component a mechanism that counts the number of the positive acknowledges received from the server component. We repeat the previous experience, in which we make adaptation during request execution.

The obtained results show that the rate of altered messages during an adaptation is almost null, thanks to the call blocking/unblocking mechanism implemented on the connectors.

According to the previous results, we can say that the proposed approach is well adapted for the applications with less frequent adaptation rate and in which we prefer performance losses to data losses.

5. Conclusion

In this paper we have presented an architecture based approach for dynamic adaptation in component-based software. The major advantage of the proposed system is the separation of the adaptation mechanism from the executed application. This makes the system independent from particular component models and platforms. The implementation for a specific component model (Enterprise Java Bean, CCM (Corba Component Model: OMG) or Dot Net: Microsoft) is made with the least effort in connectors' level and without changing the main adaptation concepts. The main advantage is the preserving of the communication channels because of a reliable adaptation.

However the evaluation of the proposed solution has revealed some limits, for example the long adaptation time makes our solution adapted only for special kinds of applications. Others limits like the non support of simultaneous adaptations, leads us to plan the following prospects:

- The improvement of the adaptation system in terms of performances;
- To extend the MAS by introducing for example several adaptation agents and ensuring their collaboration to achieve more than one adaptation at the same time.

In the long term, we plan to continue our experiments with others components models like CCM and Dot Net, and study the possibilities to extend our adaptation solution to be supported by other kinds of applications like web services. Also, we plan to enumerate and specify the kind or the type of systems or applications where our approach is well suitable and appropriate.

References

Aksit, M. and Choukair, Z. (2003). Dynamic, adaptive and reconfigurable systems overview and prospective vision. ICDCSW'03 0-7695-1921-0/03 IEEE Computer Society.

Belabed, A. and Chouarfia. A. (2008). Une approche orientée aspect pour l'adaptation dynamique des applications à base de composants. COSI'08. Tizi-Ouzou, Algérie.

Bordeie, X. (2007). Aborder Groovy, langage de script pour Java, JDN Développeurs.

Bosanac, D. (2007). Scripting in Java languages, Frameworks and patterns, Addison Wesley, ISBN 0321-32193-6.

Brinkschulte, U. et al. (2002). *Distributed real-time computing for microcontrollers – The OSA+ Approach. ISORC,* Washington D.C.

Brinkschulte, U., Schneider, E. and Picioroag, F. (2005). Dynamic real-time reconfiguration in distributed systems: Timing issues and solutions. Proceedings of the Eighth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'05), 0-7695-2356-0/05.

Chiba, S. and Nishizawa, M. (2003). An easy to use toolkit for efficient Java bytecode translators, Proc of 2nd International Conference on Generative Programming and Component Engineering (GPCE'03), LNCS 2830, pp 364-376, Springer Verlag.

Dahm, M. (1999). Byte code engineering with the JavaClass API, Technical Report B-17-98, Institut fur Informatik, Freie Universitat Berlin.

Dashofy, E.M., et al. (2001). A highly-extensible, XML-based architecture description language. In Proceedings of the Working IEEE/IFIP Conference on Software Architecture (WICSA 2001). Amsterdam. doi:10.1109/WICSA.2001.948416, http://dx.doi.org/10.1109/WICSA.2001.948416.

Huang, G., et al. (2006). Runtime recovery and manipulation of software architecture of component-based systems. *Automated Software Eng,* pp 257-281. doi:10.1007/s10515-006-7738-4, http://dx.doi.org/10.1007/s10515-006-7738-4.

John, C., et al. (2004). Towards a knowledge-based approach to architectural adaptation management, *WOSS'04*, Newport Beach, CA, USA.

Kell, S. (2007). *Rethinking software connectors.* SYANCO'07 Dubrovnik, Croatia.

Ketfi, A. (2004). Une approche générique pour la reconfiguration dynamique des applications à base de composants logiciels. Thèse de doctorat de l'Université Joseph Fourier de Grenoble France.

Kiczales, G. et al. (1997). Aspect-oriented programming. In Proceedings of the ECOOP'97.

Medidovic, N. and Taylor, R.N. (2000). A classification and comparison framework for software architecture description languages, *IEEE Transactions on Software Engineering,* 26(1).

Oreizy, P. et al. (1998). Architecture-based runtime software evolution. ICSE '98. Kyoto, Japan, April 19-25.

Pinto, M. et al. (2003). DAOP-ADL: An architecture description language for dynamic component and aspect-based development, GPCE 2003, LNCS 2830. (C) Springer-Verlag Berlin Heidelberg.

Plasil, F. et al. (1997). DCUP: Dynamic component updating in Java/CORBA Environment. Tech. Report No. 97/10, Dep. of SW Engineering, Charles University.

Qun Yang, et al. (2006). A mobile agent approach to dynamic architecture-based software adaptation: ACM SIGSOFT Software Engineering Notes, 31(3). doi:10.1145/1127878.1127889, http://dx.doi.org/10.1145/1127878.1127889.

Samuel, K., et al. (2006). Applying Prolog to semantic Web Ontologies & Rules moving toward description logic programs, ALPW.

Szyperski, C. (1998). *Component software: beyond object oriented programming,* Editor Addison- Wesley & ACM Press.

Table 1. Increase rate time

Requests Numbers	**Response time average : without adaptation mechanism**	**Response time average : with the adaptation mechanism**	**Increases rates**
10	108,71 ms	138,80 ms	**27.67 %**
30	339,27 ms	407.71 ms	**20,17%**
50	526.70 ms	614.36 ms	**16.64 %**
100	1129,05 ms	1315.55 ms	**16.51 %**
200	2154.55 ms	2506.02 ms	**16.31 %**
300	3250.64 ms	3777.05 ms	**16.19 %**
500	5417.43 ms	6292.61 ms	**16.15%**
1000	11176.91 ms	13007.83 ms	**16.38 %**

Table 2. Adaptation time

Requests Numbers	Response time without adaptation	Response time with adaptation	Adaptation time
500	7375.20 ms	10780.66 ms	**3405.46 ms**
600	8551.33 ms	11685.16 ms	**3133.83 ms**
700	9815.40 ms	13053.20 ms	**3237.80 ms**
800	11051 ms	14589.75 ms	**3538.75 ms**
Average			**3328.96 ms**

Table 3. Altered messages rate

Requests	Altered messages
500	**0%**
600	**0%**
700	**0%**
800	**0%**
1200	**0.01%**
1600	**0.01%**

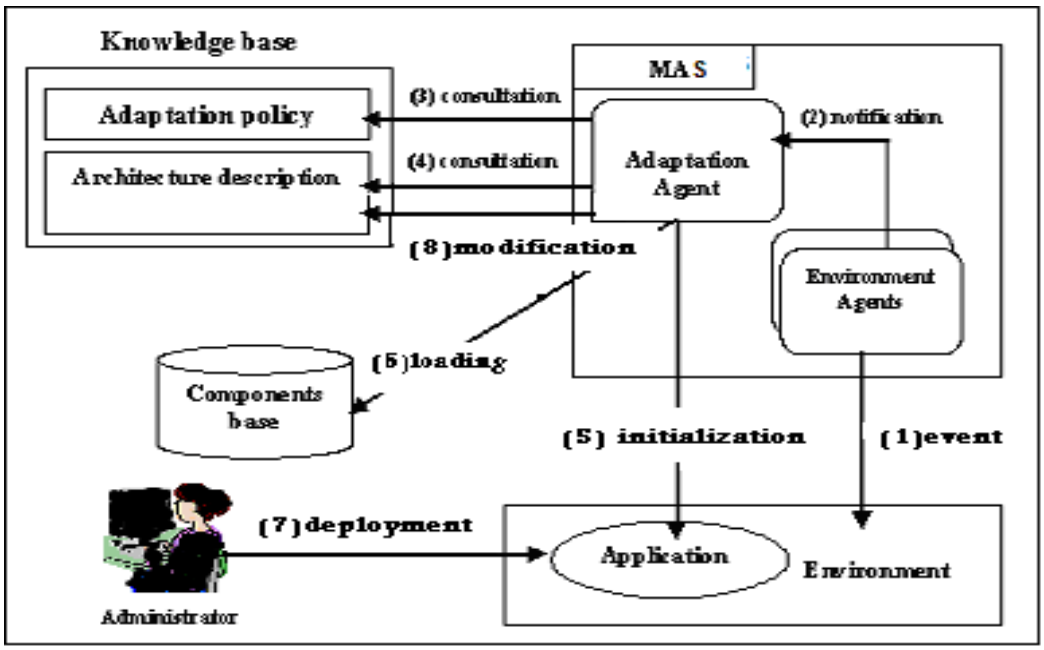

Figure 1. Proposed architecture

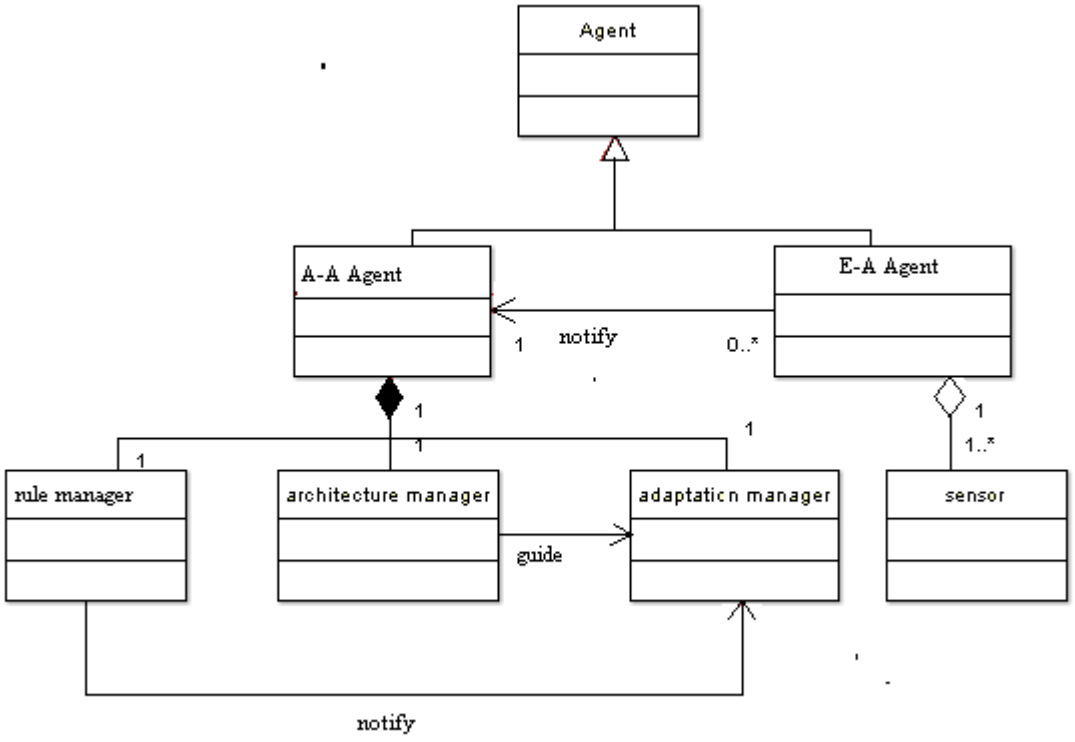

Figure 2. Multi agent system

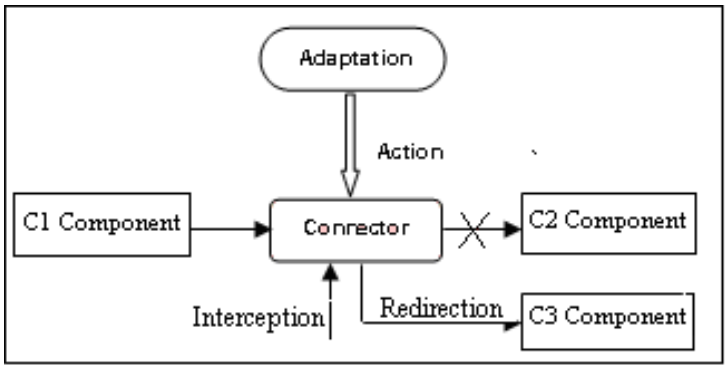

Figure 3. Adaptation principle

ISSN 1913-8989 E-ISSN 1913-8997

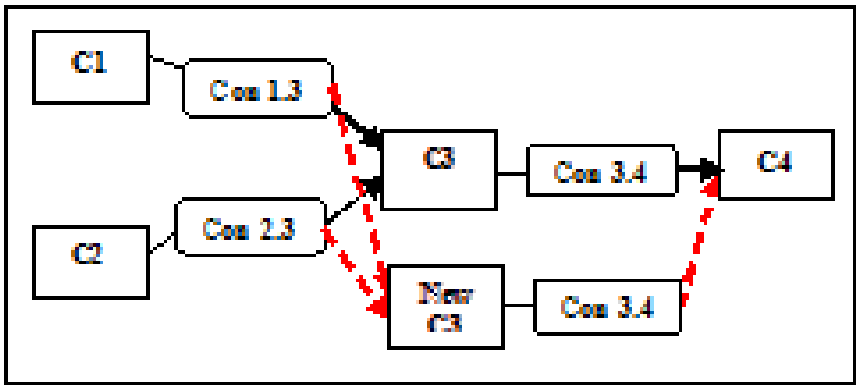

Figure 4. Component replacement