MODEL-DRIVEN FRAMEWORK FOR DYNAMIC DEPLOYMENT AND RECONFIGURATION OF COMPONENT-BASED SOFTWARE SYSTEMS

Abdelmadjid Ketfi
Adele Team Bat C
LSR-IMAG , 220 rue de la chimie
Domaine Universitaire, BP 53
38041 Grenoble Cedex 9 France
Abdelmadjid.Ketfi@imag.fr

Noureddine Belkhatir
Adele Team Bat C
LSR-IMAG , 220 rue de la chimie
Domaine Universitaire, BP 53
38041 Grenoble Cedex 9 France
Noureddine.Belkhatir@imag.fr

ABSTRACT. Permanent and uninterrupted functioning can be sometimes a requirement for some kinds of software systems. This is especially true in the case of complex and distributed systems where stopping and restarting the system constitute a tedious and costly task, also when the system must be highly available or when its execution environment changes frequently. Many component technologies exist today and solve hot (dynamic) deployment and reconfiguration issues offering ad-hoc solutions. This paper presents DYVA, a unified framework, that has been designed to be suitable to dynamic deployment and reconfiguration for most of the currently component technologies. Components from these technologies have to satisfy common features like encapsulation, interfaces, etc...). The proposed framework is based on a model-driven approach. This approach helps in developing specific hot deployment and reconfiguration systems by the personalization of the unified framework. The personalization is done via plug-ins which make transformation between specific component descriptions and more general ones.

Keywords:

MDA, Metamodel, Framework. Dynamic deployment / reconfiguration, Component.

1. INTRODUCTION

Component-based software (CBS) engineering focuses on building large software systems by integrating existing software components [1]. The old notion of developing a system by writing code has been replaced by assembling existing components. In general the deployment of a CBS is considered in a static way: only situations where the software is deployed from scratch are considered, and when the update is supported a complete software stop is required. Few existing component technologies (some EJB implementations for example) support a dynamic update of already deployed components without the obligation of restarting the application server.

Metainformatics Symposium, November 9–11, 2005, Esbjerg, Denmark.

For some critical and highly available software systems the *dynamicity* is considered as an important criteria to guarantee an acceptable availability and quality of service. In this paper, by dynamic we mean the ability to re-deploy and to reconfigure a running system to take into account some new conditions, sometimes unpredictable, without completely stopping it. The development complexity and cost constitute an important problem in front of the creation of supports responsible for the dynamic deployment of software systems. The work we present in this paper contributes in solving the problem of cost and complexity of dynamic deployment supports building process. Our main aim is to promote a new approach supported by a generic framework that helps developers in building dynamically reconfigurable software systems. The objective is to delegate the dynamic reconfiguration issues to a dedicated system in order to allow developers to concentrate on the business logic of software systems they build. This is very useful for many application domains where the software systems are the central elements that must be available 24 hours a day and 7 days a week. Supervision medical systems, aeronautic control systems and some embedded mobile systems are typical domains where our work finds its applicability. For genericity reasons, our approach shares the same ideas with MDA (Model Driven Architecture) [2] which is an approach to software development using models.

This paper is organized as follows: Sections 2 and 3 discuss our motivations for this work and present some basic dynamic deployment concepts. Section 4 presents some approaches dealing with the dynamic deployment problem followed by a synthesis. Before we conclude, Section 5 illustrates the main axes of our approach.

2. MOTIVATIONS

The dynamic maintenance, adaptation and extensibility of software systems is a subject that has been tackled since many years by a large community of researchers. The common interest is to provide methods and tools capable to apply on software systems at runtime the same modifications that can be applied at design time. Obviously, transiting from design to runtime implies very considerable challenges. By analogy, someone can replace a part of a car naturally when this car is parked, however trying to replace this part when the car is running is a task of a very higher degree of complexity. Especially when the consistency and the security must be guaranteed.

Recently, many researchers belonging to the metainformatics community, in particular, have attempt to show the increasing interest of the dynamic aspect in today software

systems. For example [3] discussed the limitations of closed and static systems and proposed an approach for modelling and implementing dynamically extensible software systems following a service oriented paradigm. [4] demonstrated with many examples that the static aspect of software systems and especially of the life cycle process used to build these systems is not suitable for many domains. This is because no attention is given to long-term and real-time considerations. Dynamic deployment (and adaptation) is strongly required for many application types. The following scenarios illustrate some of these applications:

E-commerce and highly available applications cannot be stopped for economical reasons. Stopping these applications may seriously affect the quality of service and decrease the clients confidence. Amazon portal for example has to respond daily to many millions of requests and stopping it for maintenance should be very negative.

Distributed and large scale applications require a considered effort to be stopped, updated, rebuilt and correctly restarted. It is interesting to update when necessary only specific parts of these applications without being forced to stop the whole application.

Embedded applications by their nature require a remote intervention while they are running. Satellite television terminals and also recent car systems are a good example of embedded applications remotely deployed and reconfigured.

Some applications (multimedia applications for example) must respond to requests of heterogeneous clients (PCs, PDA, Phones…). These applications must be able to dynamically adapt their answers according to the clients capabilities (memory, display, bandwidth…).

Most applications belonging to one of the listed domains require to be highly available. If such applications are component-based, our generic framework can be personalized and used to manage their dynamic deployment/reconfiguration. The foundation of this framework and its personalization are explained in the fifth section of this paper.

3. BASIC CONCEPTS

Figure 1 presents the deployment life cycle related to component-based applications. The main difference with the classic deployment life cycle (of monolithic applications) resides in the nature of the applications themselves: a component based application is made of a set of well defined pieces. The deployment can be made incrementally and the re-deployment may concern only some specific pieces and not the whole application. In the case of dynamic deployment the target application continues to be executed and only the re-deployed components are stopped.

The deployment life cycle includes many basic activities. These activities and potential relations between them are illustrated in Figure 1.

- **Configuration**: a component may have many versions and many implementations for different operating systems for example. The configuration consists of selecting the appropriate components according to the target platform, packaging them with their depending resources. According to the component model, a package may contain one or many components.

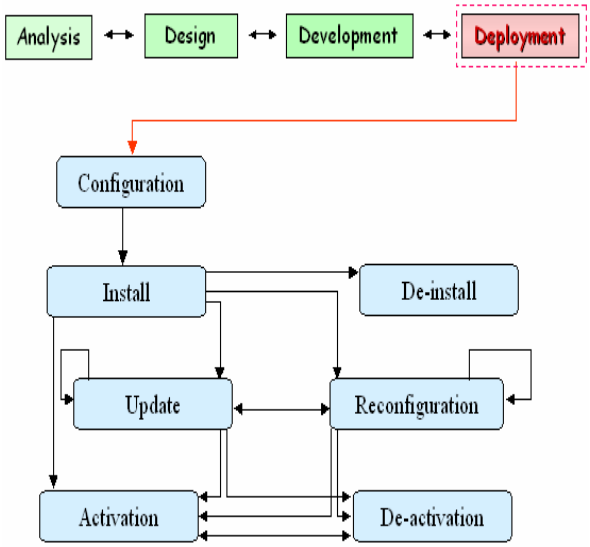


Figure 1. Deployment life cycle

- **Install:** consists of transferring the applications packages from a storage repository to the target site or sites.
- **Activation:** after the components are injected within the target environment, they have to be loaded and started in order to use their services.
- **De-activation:** for different reasons, a component or a set of components have to stopped and unloaded.
- **Update:** consists of reinstalling some components already installed, or adding new components to an application previously installed.
- **Reconfiguration:** means the modification of the installed application using the elements already available on the site (without need for re/installing other components). For example disconnecting two components or plugging a component in the place of another, both present on the site.
- **De-install:** consists of removing from the target site the components no longer used.

The main difference between static and hot deployment is related to the behavior of the application during the reconfiguration/update. Hot deployment means that only the components concerned by the reconfiguration are de-activated when the other components continue to be executed.

Many existing component models provide explicit support for deployment. Such a support is generally based on some deployment descriptors and some tools to effectively deploy the applications on the target environment with respect to the provided descriptors. These tools are also sometimes used to instrument as later as possible the components binary code and/or to generate objects commonly known as containers and responsible for the management of the deployed components. The following example illustrates the deployment descriptors related to EJB [5] component model:

- In EJB the bean provider is responsible for providing a deployment descriptor (called ejb-jar.xml) associated with the developed Enterprise Beans. As illustrated in the following example this file tells the EJB server which classes make up the bean implementation, the home in-

terface and the remote interface. If there is more than one EJB in the package, it indicates also how the EJBs interact with one another.

```
<?xml version="1.0" encoding="UTF-8" ?>
- <ejb-jar>
    <description>MyEJB Example</description>
    <display-name>Calculator Bean</display-name>
  - <enterprise-beans>
    - <session>
        <ejb-name>Calculator</ejb-name>
        <home>test.CalculatorHome</home>
        <remote>test.Calculator</remote>
        <ejb-class>test.CalculatorBean</ejb-class>
        <session-type>Stateless</session-type>
        <transaction-type>Bean</transaction-type>
      </session>
    </enterprise-beans>
  </ejb-jar>
```

Another descriptor may be also needed. This descriptor is specific and contains advanced information related to a particular application server (jboss.xml for JBoss [6], jonas-ejb-jar.xml for JOnAS [7]).

In general a deployment descriptor gives some information related to component types, interfaces, implementations, and to some system properties (transactions, security…). However, there is no information in general about the dynamic behavior of components. For example how to know that a component can be replaced, removed or migrated and how to know that a connection is required, dynamically modifiable, etc.

4. STATE OF THE ART

This section gives a synthetic view of research work in the field of dynamic deployment and adaptation. It is true that the deployment has been highlighted and have taken importance since few years. However, dynamic applications adaptation is a very old problematic [8].

Dynamic deployment and especially dynamic adaptation is very closely related to the low level mechanisms supported by the run-time infrastructure. Java for example provides the class-loader and reflection mechanisms that allow developers to respectively loading/unloading classes and to introspect the components behavior and structure at run-time. These two powerful mechanisms are not supported in C++.

According to the used programming model, many research approaches were proposed to support the deployment and adaptation of applications at run-time.

- ***Procedural approaches****:* they aim in general to dynamically replace one procedure by another. For example DYMOS (Dynamic Modification System) [9] developed in 1983 by Insup Lee, provides a complete embedded environment for application (based on StarMod language, similar to Modula) development, execution and adaptation. At run-time the developer can modify one or many procedures and ask the system to replace these procedures in memory. Another similar system called PODUS (Procedure-Oriented Dynamic Update System) [10] allows many versions of the same procedure to be loaded and used at run-time. The main challenge in such approaches is related to the replacement of active procedures (procedures in execution) and to the replacement of a procedure by another without the same signature.

- ***Modular approaches****:* based on modular programming languages, these approaches aim to replace at run-time one module by another. Polylith [11] developed at Maryland University since the middle of the Eighties is the more representative system in this category. Based on C language Polylith has the same concepts as today middleware systems. It proposes a support for developing, running and dynamically reconfiguring distributed applications. Applications are represented thanks to a module interconnection language. Polylith software bus is responsible for the distributed interaction and also for the dynamic adaptation of modules which includes: module re-deployment and replacement, reconnection and modules migration from a machine to another one.

- ***Object-based approaches****:* one of the main limitations (related to dynamic deployment/adaptation) of object-oriented programming languages like Java (versions under Java 1.5) and C++, is the inability to reload already loaded classes. Java offers the class loader mechanism[1] that can be used to *unload* and *reload* classes however the programmer is responsible for implementing the necessary code (of class unloading and reloading). Too much work have been done in order to extend Java (Dynamic Java Classes [12]) and C++ (Dynamic C++ classes [13]) runtime environment in order to transparently support these primordial two operations.

- ***Component-based approaches***: these approaches aim to provide support that allows to deploy components at run-time [14, 15]. These components are injected to extend the application or to replace other components already deployed and even activated. Unlike object-oriented applications component programming supposes that the implementation of components is explicitly separated of their assembling (components do not contain architectural information). This separation implies that an explicit architecture can be obtained at run-time which is very important to be able to dynamically acting on this architecture (plugging a component in the place of another, reconnection…).

- **Synthesis**

After analyzing the previous approaches and in particular those related to component-based applications, two aspects can be noted:

It is generally recognized that information related to deployment have to be specified outside the components implementation. This information is concretized as metadata stored in descriptor files. In general a deployment descriptor is a set of key/value pairs. Obviously the set of keys recognized in a component technology are not generally the same recognized in another technology.

The other important point is related to the techniques used to modify the applications at run-time. Many techniques can be listed however the “great” idea behind these techniques is the same. This idea is based on the ability to reify the connections between the application elements and to act dynamically on these connections. The reification can intervene at different levels (Figure 2) and here the used approaches can be different.

The first approach is based on the modification of the virtual machine in order to add a support for the dynamic adaptation. This technique has been applied in the Dynamic Java Classes described above. The advantage of this first technique is that all applications running on top of the modified virtual machine benefit automatically from the dynamic adaptation capabilities.

[1] A class loader is a Java class responsible for loading other classes.

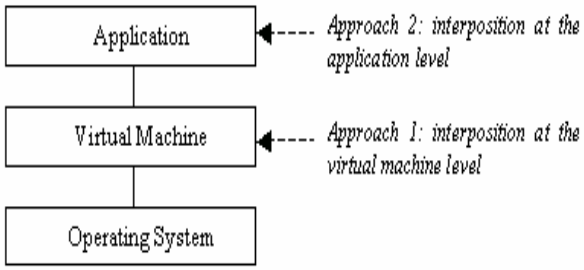


Figure 2. Interposition approaches

The second approach supposes the reification at the application level. Many techniques are generally used to concretize this reification. The containers[2] are for example the commonly used technique to reify connections in component based technologies. They are particularly inserted in order to apply some necessary system services (security, transactions, mapping with databases) before transferring the calls to the wrapped components. Also AOP (Aspect Oriented Programming) [16] is a widely used technique based on the instrumentation of applications in order to inject additional code. This code allows for example to catch the control flow at specific points in order to apply some actions (tracing calls, checks…).

5. UNIFIED FRAMEWORK FOR SUPPORTING DYNAMIC DEPLOYMENT AND RECONFIGURATION

5.1. Overview

Recently we specified and developed an ad-hoc dynamic reconfiguration framework for JavaBeans [17] based systems [18]. The role of this framework is to take in charge the dynamic reconfiguration of systems developed according to JavaBeans component model. The reconfiguration is performed almost transparently and needs a minimal participation of the user. This framework allows for example to replace a component by another one, either having the same interface or not (interface mapping facilities are supported), it allows also to change the system architecture by changing the connections between components, by adding or removing components, it guarantees some consistency of the reconfigured system by assuring the state transfer between the old and the new component, *passivating* and *activating* components as necessary.

In the context of another project, we developed a similar dynamic reconfiguration framework for OSGi [19], a component model in which components plug into a basic framework and allow to create applications in an incremental fashion at runtime [20]. This dynamic reconfiguration framework allows practically the same dynamic reconfiguration operations listed above.

After these experimentations and after studying the deployment and other dynamic aspects related to two other component technologies (EJB and Fractal [21]), some basic concepts seemed to be apparent and shared by both models as for instance the dynamic deployment routines and process. This observation was a key element that motivated us to work on a hot deployment and reconfiguration (HDR) framework. By HDR framework we mean the software responsible of the dynamic deployment and reconfiguration of a component based application.

5.2. The conceptual idea

A unified framework can be reached by introducing an *abstract* component model which is endowed by capabilities to support the dynamic deployment/adaptation. This abstract model is intended to be representative of the targeted component models and constitutes a basic step towards a dynamic deployment/adaptation framework. By analogy with MDA the abstract component model corresponds to a PIM. The next section describes this abstract model and shows in what way it can be used.

Figure 3 illustrates the basic idea of our approach. The first step is to analyze the most representative component technologies especially according to the deployment and dynamic aspects.

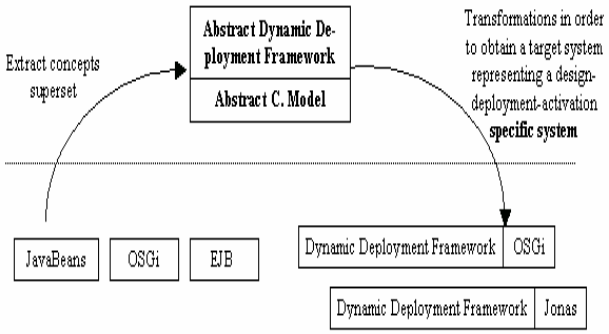


Figure 3. The conceptual idea of a dynamic deployment framework

The abstract component model presented in the following section is one of the pieces resulting from the unification process. The abstract framework which is based on this abstract component model needs then to be transformed, specialized and completed to finally obtain a dynamic deployment framework targeting a given technology.

It is important to know that modifying an application at run-time requires at least a precise knowledge of its architecture. In addition to this architecture it is necessary to master the interposition technique used to wrap the components and to transparently apply low level treatments (re-biding for example). In our approach, it is supposed that the framework is responsible for maintaining the application architecture (represented according to the abstract component model), for deploying the components binaries (and their resources) and also for the activation of these components. The activation is a critical task that must be controlled by the framework in order to insert the required containers that are responsible for managing the dynamic deployment and reconfiguration. At runtime the administrator can use a deployment/reconfiguration interface included in the framework to dynamically act on the application.

5.3. Abstract component model logical view

Figure 4 shows a logical view of our abstract component model which captures the main logical concepts that can be found in other component models like JavaBeans and OSGi. Some points deserve to be underlined:

A component may require some ports and provide some ports needed by other components.

[2] In practice a container can be a dynamic proxy (EJB/JBoss), a static generated interceptor (EJB/Jonas) or any other interception object.

We make an explicit distinction between a component (specification and implementation of a software entity) and a component instance (the component used in a specific system or part of a system).

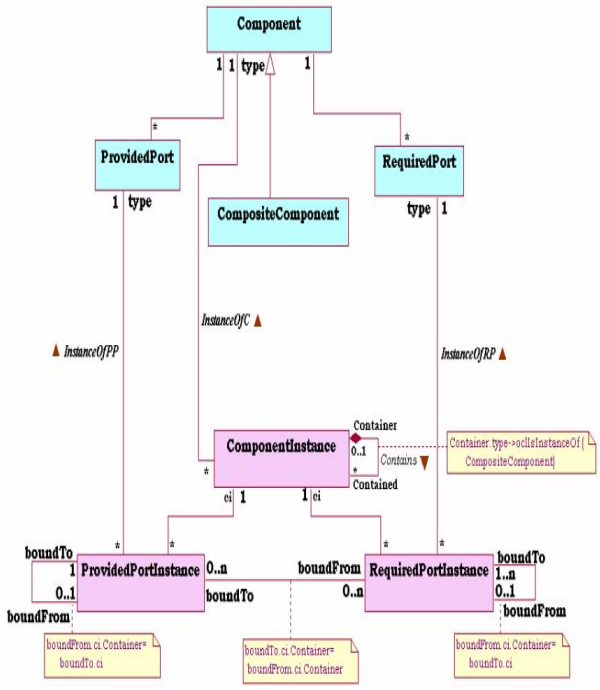


Figure 4. Abstract component model

- The model is hierarchical: a *Component* can be a primitive component or a *CompositeComponent* that is also considered as *Component* and made of a set of sub-components.

- The model is only descriptive and is not intended to be executable (there is no runtime machine associated to this component model).

- Annotations (metadata) can be associated to each entity of the abstract component model. An annotation describes a deployment aspect (target machine, persistency…) or a dynamic aspect (replacement and migration, compatibility, execution state…) of the software components to be deployed.

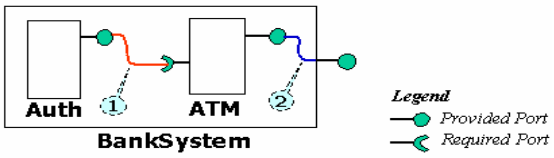


Figure 5. Application example

For example in Figure 5 the instance of the composite component *BankSystem* is made of two component instances *Auth (Authentication)* and *ATM (Automatic Teller Machine)*. *ATM* requires an authentication service to check clients identity, this service is provided by *Auth*. Concretely the required port of *ATM* is connected to the provided port of *Auth* (1). Also *BankSystem* provides a service to its clients materialized by a provided port. This port is bound to *ATM*'s provided port (2), this binding means that the provided service of *BankSystem* is in reality guaranteed by *ATM*.

5.4. Dynamic reconfiguration framework architecture

Our framework is based on the abstract component model presented above. It is structured in several modules as shown in Figure 6.

Deployment/Reconfiguration manager

It is the central part of the framework. It provides and implements the basic dynamic deployment and reconfiguration routines. These routines operate on the system abstraction (and not directly on the base-level) which guarantees the independence and the genericity of our unified framework.

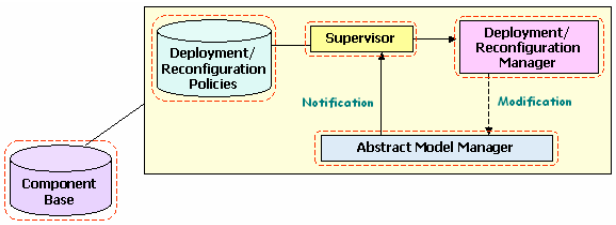


Figure 6. Dynamic reconfiguration framework architecture

Supervisor

The role of the supervisor is to allow the auto-reconfiguration/deployment of applications. It represents a reasoning engine that introspects or receives events from the system and its environment and takes decisions according to these events and according to some reconfiguration and deployment policies.

Reconfiguration policies

Some policies are required to make possible the self-reconfiguration of applications. Self-reconfiguration means the ability to take consistent dynamic reconfiguration decisions without the intervention of an external actor (usually a human administrator). The current specification of reconfiguration policies is very primitive. It considers a very simple form of reconfiguration rules. The improvement of this specification is one of our perspectives**.**

5.5. Personalization of the framework for a specific component technology

The unified framework is not a complete deployment and reconfiguration system ready to use. This is because the framework is not dedicated to a specific component technology and intended to be generic. To use the system in practice, it is necessary to complete and to personalize it for a specific component technology.

To personalize DYVA for a specific component technology, it is not required to master its internal structure. This task can be performed thanks to a set of provided interfaces and utility classes.

Figure 7 presents the external view of DYVA

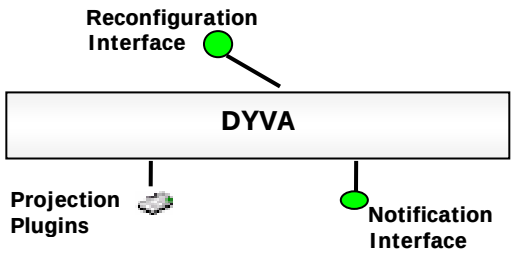


Figure 7. DYVA external view

Section 6 describes the personalization of DYVA for a specific component model called OSGi. In this sub-section we present the interfaces to be used in this process. As explained above, the abstract model includes the image of the application to be dynamically reconfigured. The modifications at the application level must be reflected to its abstract representation and vice versa. The notification interface provided by DYVA is used to notify the modifications to the abstract representation. DYVA is not specific to a particular component technology. Consequently the notification in the opposite direction (projection of the modifications applied at the abstract representation on the concrete application) cannot be definitively defined. We have based DYVA on a plugins structure in order to be able to adapt the projection operations according to the targeted component technology.

Notification interface: defines a set of methods that can be used to send events to DYVA. An event is raised in one of the following situations: (1) a component instance is created, (2) a component instance is removed, (3) a connection between two instance ports is established, (4) a connection between two instance ports is removed. These four event types are related to the application architecture. They are required to build an architectural reflective system like that proposed in [22]. Two other event types can be distinguished:

- *Interaction between two instances:* operation calls are intercepted and redirected to the reconfiguration system. Thus additional treatments can be applied before and/or after or instead of the original operation call.
- *Environment change:* dedicated tools (like those developed in [23]) must be used to monitor the system environment (bandwidth, memory, processor load…) and to notify significant changes to the reconfiguration system. Consequently decisions can be made according to the deployment/reconfiguration policies described above.

Reconfiguration interface: includes the basic deployment/reconfiguration functions supported by the framework (component deployment, component instantiation, component instance remove, dynamic disconnection/connection/reconnection, internal state transfer…).

Projection plugins: the projection of the operations performed at the abstract representation depends on the concrete targeted component technology. Lets consider for example the disconnection operation between two component instances A and B. Concretely this operation is interpreted differently according to the component technology. In the case of JavaBeans for example, B is removed from the A event listeners list. In the case of OSGi, the disconnection is performed by calling a specific method of the source instance (A). However in the case of Fractal, the binding controller attached to each component is responsible for the disconnection.

To keep our framework generic, the projection operations must be integrated as plugins. Each plugin must respect a precise specification and is responsible for projecting the modifications for a specific component technology.

The following section illustrates the personalization process of our framework for OSGi and discusses its validation via a web portal example.

6. PERSONALIZATION OF THE FRAMEWORK FOR OSGI AND VALIDATION

As illustrated in
Figure 8, the personalization of the unified framework for a specific component technology consists of specifying and implementing specific notification and reconfiguration interfaces based on the framework common interfaces. To achieve the personalization the required projection plugins must also be implemented.

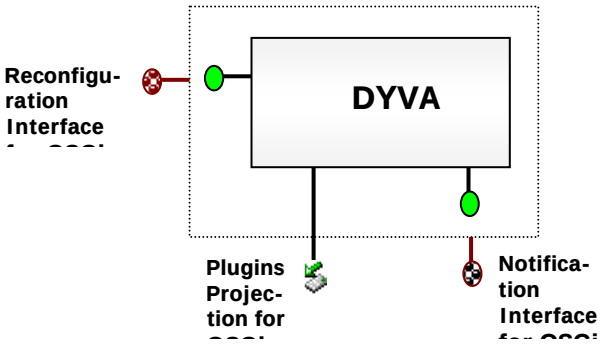


Figure 8. DYVA Personnalisation for OSGi

6.1. Notification interface for OSGi

This interface is used by OSGi applications to send notifications to the reconfiguration system. Figure 9 illustrates this interface.

```
public interface IOSGiNotification {
public void notifyServiceCreation(...);
public void notifyServiceConnection(...);
public void notifyServiceRemoval(...);
public void notifyServiceDisconnection(...);
public void notifyEvent(...);
}
```

Figure 9. OSGi Notification Interface

For example, when an OSGi service object is created, `notifyServiceCreation(…)` is called with the reference of the created object as argument.

6.2. Reconfiguration interface

It defines the OSGi applications specific deployment/reconfiguration operations. Figure 10 gives a partial view of the operations defined within this interface.

```
public interface IOSGiReconfiguration {
    public boolean freeServiceInterface(...);
    public boolean freeService(...);
    public boolean freeService(...);
```

```
    public boolean freeBundle(...);

    public boolean setService(...);

    public     boolean     reconnectToSer
                          vice(...);
    public boolean removeService(...);
    public boolean removeBundle(...);
    public long[] deployBundle(...);
    ...
}
```

Figure 10. OSGi Reconfiguration Interface

The implementation of the previous reconfiguration operations is closely based on the basic reconfiguration operations provided by the unified framework. Relatively to the *bundle* concept defined in OSGi (a bundle is a unit that includes a set of components), we would like for example to define a new operation called `freeBundle(…)` which role is to disconnect a bundle from a running application. This operation can be interpreted as the disconnection of all instances of all components included in the bundle. The following example shows a possible implementation of such operation:

```
 public boolean freeBundle(long bundleID)
{
// Find the bundle
  Bundle
bundle=bundleContext.getBundle(bundleID);
// Find the components contained in the bundle
 ComponentInstance cis[] = reconfManager-
Ref.getAllComponentInstances(bundle);
//Free services provided by these compo-
nents
 for(int i=0; i<cis.length; i++)
      freeService(cis[i].getName());
// Stop the bundle
      bundle.stop();
       return true;
```

6.3. Projection plugins for OSGi

The most important personalization effort is related to the projection plugins. The following points give an overview of the plugins we have implemented for OSGi.

- Instantiation plugin: responsible for the creation and for the destruction of OSGi component instances.
- Deployment plugin: allows to deploy a component or a set of components contained in the same OSGi bundle.
- State transfer plugin: this plugin is based on two operations (getState() and setState()) provided by each stateful component. It performs the internal state transfer between a replaced and a replacing component instances.
- Communication management plugin: it manages the life cycle of component instances. It also stores the requests sent to a given passivated instance, and handles these requests when this instance is activated.

6.4. Validation on a web portal application

The OSGi personalized version of our framework has been tested on a web portal application. The architecture of this application is shown in the framework control interface presented in

Figure 11.The *PortalServlet* component is the entry point that allows to use the services provided by the portal application. It receives the requests sent by clients before redirecting them to the appropriate application component. At runtime, the control interface can be used to manually reconfigure the application.

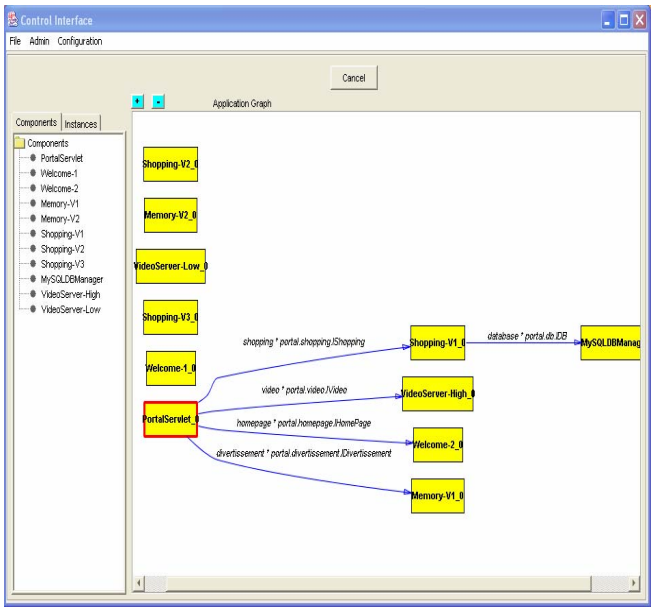


Figure 11. Web portal application architecture

The dynamic reconfiguration can also be automatically instigated by the system in response to appropriate events sent by monitoring tools.

Figure **12** for example shows a tool that monitors the network bandwidth and sends events to the reconfiguration system.

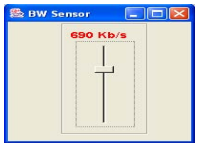


Figure 12. Bandwidth sensor tool

When the bandwidth decreases under a given value specified in the following reconfiguration rule, a reconfiguration decision is automatically taken and applied.

```
[RULE]
ON: BANDWIDTH_EVENT
```

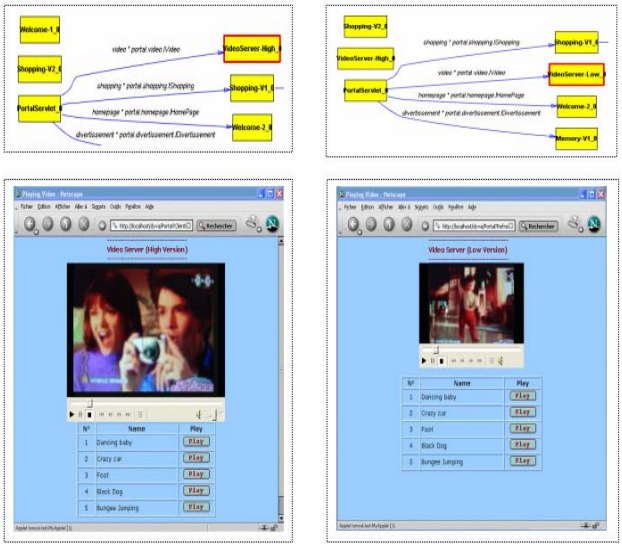
```
IF: ((NEWBW = 400) AND (NEWBW < OLDBW))
THEN:     replace     VideoServer-High_0
                      VideoServer-Low_0

[RULE]

ON: BANDWIDTH_EVENT
IF: ((NEWBW = 600) AND (NEWBW > OLDBW))
THEN:     replace     VideoServer-Low_0
                      VideoServer-High_0
```

Figure 13. Reconfiguration rule related to the bandwidth variations

When the previous rule is applied, the current used video instance (*VideoServer-High_0*) is dynamically replaced by another one (*VideoServer-Low_0*). The following figure shows the transition after the reconfiguration.



Application before the reconfiguration **Application after the reconfiguration**

The web portal application allowed us to test the OSGi personalization of our framework. The next imminent step is to validate this personalization on a real size application. This is one of the perspectives of our work.

7. CONCLUSION AND PERSPECTIVES

We have developed DYVA, a unified framework for dynamic reconfiguration of component applications. This framework is targeted to existing component models. Such a framework takes in charge all complex tasks related to dynamic reconfiguration and allows developers to focus only on the business logic of their systems. As proof of concepts, the personalization we implemented for JavaBeans and OSGI models confirm the validity of our approach that seems to be very promising.

The meta-modeling has proved a real importance especially with the appearance of many standards and methodologies like MDA, the recent OMG initiative. The approach we presented in this paper lies within the same scope, it is based on a meta-model that provides an abstract view of concrete software systems and allows to build a generic dynamic reconfiguration framework.

The next immediate step is to continue our experiments with others components models. The objective is to gather all common features of software components that are necessary for the unified framework. Other near future work includes the integration of D&C model of the OMG for configuration and deployment as a basis of the unified model as well as continuation of the case study and application of our work to other real world systems representing other domains, such as those related to autonomous computing.

8. REFERENCES

[1] George T. Heineman , William T. Councill: Component-based software engineering: putting the pieces together. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, 2001.

[2] John D. Poole: Model-Driven Architecture: Vision, Standards, And Emerging Technologies. Position Paper Submitted to ECOOP 2001, Workshop on Metamodeling and Adaptive Object Models.

[3]. P. King, M. Nanard, J. Nanard, and G. Rossi: A Structural Computing Model for Dynamic Service-Based Systems. MIS '03 Workshop, Graz, September 2003, Springer Verlag: LNCS3002 Ed. Hicks, David L. 2004.

[4]. D. Dalcher: Developing Software for Dynamic Systems. Metainformatics Symposium proceedings, David Hicks (Ed.), MIS2002, Aalborg University, Springer Verlag, 2003

[5] S. Ambler, T. Jewel: EJB fondamental. Eyrolles, May 2002.

[6]. JBoss Open Source Application Server, http://www.jboss.org

[7]. JOnAS: Java (TM) Open Application Server, http://jonas.objectweb.org

[8] R.S. Fabry. How to design a system in which modules can be changed on the fly. Proc. 2nd lnt. Conf. on Soft. Eng., pp. 470-476 (1976).

[9] Insup Lee. DYMOS: A Dynamic Modification System. Department of Computer Science, University of Wisconsin, Madison, April 1983.

[10]. M. E. Segal , O. Frieder: On-he-Fly Program Modification: Systems for Dynamic Updating. IEEE Software, v.10 n.2, pp.53-65, March 1993.

[11] J. M. Purtilo: The POLYLITH Software Bus. ACM TOPLAS, vol. 16 (N.1), pp. 151-174. January 1994.

[12] Malabarba , R. Pandey , J. Gragg , E. Barr , J. F. Barnes: Runtime Support for Type-Safe Dynamic Java Classes. Proceedings of the 14th European Conference on Object-Oriented Programming, p.337-361, June 12-16, 2000.

[13] G. Hjálmtýsson, R. Gray: Dynamic C++ classes - A Lightweight mechanism to update code in a running program. In proceedings of the USENIX Annual Technical Conference, pp. 65-76, June 1998.

[14] J. Dowling, V. Cahill: Dynamic Software Evolution and The K-Component Model. Technical report, Trinity College Dublin, TCD-CS-2001-51. December 2001. Presented in the Workshop on Software Evolution, OOPSLA 2001.

[15] F. Plasil, D. Balek, R. Janecek: DCUP: Dynamic Component Updating in Java/CORBA Environment. Tech. Report No. 97/10, Dep. of SW Engineering, Charles University, Prague, 1997.

[16] T. Elrad, O. Aldawud, A. Bader: Aspect-Oriented Modeling: Bridging the Gap between Implementation and Design. Generative Programming and Component Engineering, ACM SIGPLAN/SIGSOFT Conference, GPCE, Pittsburgh, USA, October 2002.

[17] JavaBeans Architecture, Sun Microsystems. http://java.sun.com/docs/books/tutorial/javabeans/

[18] A. Ketfi, N. Belkhatir and P.Y. Cunin: Dynamic updating of component-based applications: SERP'02. June 2002, Las Vegas, Nevada, USA.

[19] Open Services Gateway Initiative (OSGi) - http://www.osgi.org

[20] A. Ketfi, H. Cervantes, R. Hall, D. Donsez. Composants adaptables au dessus d'OSGi. Journées Systèmes à Composants Adaptables et extensibles Octobre 2002, Grenoble, France.

[21] The Fractal Project - http://fractal.objectweb.org/

[22] W. Cazzola, A. Savigni, A. Sosio, and F. Tisato. Architectural reflection : Concepts, design, and evaluation. Technical report, Technical Report RIDSI 234-99, DSI, Universita deglistudi di Milano, May 1999.

[23] Fractal LeWYS Project - http://forge.objectweb.org/projects/lewys/