

Dynamic updating of component-based applications

Abdelmadjid Ketfi, Nouredine Belkhatir, Pierre-Yves Cunin

Adele Team Bat C
LSR-IMAG , 220 rue de la chimie
Domaine Universitaire, BP 53
38041 Grenoble Cedex 9 France

Abstract

Component-Based Software Engineering (CBSE) focuses on building large software systems by integrating existing software components. The old notion of developing a system by writing code has been replaced by assembling existing components. The main aim of CBSE is to enhance the flexibility and maintainability of systems. Some of these systems are critical, and should not be stopped to be maintained. Such systems must be adapted on the fly. In this paper, we focus on the dynamic adaptation problem. Dynamic means the ability to change an application at run-time. In the first part, we provide a survey of the main concepts of the dynamic adaptation. The remainder of the paper presents our adaptation approach experimented firstly on the JavaBeans component model. The approach consists of dynamically choosing an application configuration most appropriate to resource conditions and application progress. We envision an approach that takes advantage of the availability of alternate components configurations. This latter provides a powerful model for the construction and execution of adaptive applications.

Keywords: *Dynamic adaptation, component, deployment, JavaBeans, XML.*

1. Introduction

Current applications would no longer be large monolithic structures built by writing code but rather assemblies of existing components, competitively produced by different developers

who would have the specialized skills to concentrate on issues such as quality and reliability for the components they provide.

In general, it is necessary to customize a component according to specific requirements of the application in which it will be plugged. Traditionally, the application to be adapted must be stopped for maintenance. This approach is not suitable for critical systems that have to be non-stop and highly available like bank, Internet or telecommunication services. In this kind of systems, the adaptation is more complex and must take place at run-time.

Adapting a component-based application means adapting one or more of its components, and in general, adapting a component at run-time means disconnecting it from the application and connecting a new version of this component. In this paper we start by a survey of the adaptation problem. We introduce some key concepts, and try to highlight briefly some existing works. Our approach is presented in the remainder of the paper. To experiment the proposed approach, we first focus on JavaBeans component model and, in future work, we generalize our experimentations on other models such as EJB and CCM.

The paper is organized as follows: section 2 provides a global survey of the adaptation problem. In section 3 we present and discuss some adaptation works. Our adaptation approach

is introduced in section 4. Section 5 discusses the experimentation of the presented concepts on the JavaBeans component model using XML technologies, before we conclude in the last section.

2. Survey of the dynamic adaptation problem

A dynamic adaptation may be performed for multiple reasons, that can be classified into four categories: corrective, adaptive, extending or perfective.

- Corrective adaptation: removes the faulty behavior of a running component by replacing it by a new version that provides exactly the same functionality.
- Adaptive adaptation: adapts the application in response to changes affecting its environment (OS, hardware components,...).
- Extending adaptation: extends the application by adding new components to provide the new needed functionalities.
- Perfective adaptation: aims to improve the application even it runs correctly. For example, replacing a component by a new one with more optimized implementation.

Several adaptation types may be identified:

- Architecture adaptation: affects the structure of the application. This can be performed by adding new components, removing existing components or modifying their interconnections.
- Implementation adaptation: motivated by performance, correction reasons or environmental changes not considered when the component was implemented. Interfaces exposed by the component must be maintained the same.
- Interface adaptation: modifies the list of services provided by the component. In component-based software, that can be performed by adding (removing) an interface in (from) the list of interfaces supported by the component.
- Geography adaptation: That corresponds to the migration of components from a site to another one, to load balance for example. That does not affect the application's architecture,

however, the communication between the moved component and other components should be adapted according to the new location.

Multiple requirements must be satisfied to accurately perform an adaptation.

- Consistency: the adaptation operation which is instigated by the control application (the application used to adapt the running application) has high priority. However, that does not give all rights to the control application to do anything at any time. The adaptation operation must preserve the application consistency.
- Performance: even it is considered that the adaptation operation is a seldom event during the running application life cycle, it should be efficient, and its duration should be as minimal as possible. Also, the number of components affected by the adaptation operation should be minimal.
- Degree of automation: it represents the ability of an application to adapt itself; this is possible because during run-time, the application has all information and capabilities needed to carry out such an operation.

In the next section, we present and briefly discuss some works related to dynamic adaptation problem.

3. Related work

Dynamic adaptation is not a new problem, R. Fabry [1] explained in 1976 how to develop a system in which modules can be changed on the fly. Several other works dealt with the adaptation problem such as *Dynamic Linking*, *Redundant Hardware* and *State Transfer*. Other more recent approaches are presented in this section.

3.1. DCUP

3.1.1. Presentation

Many adaptation approaches associate to each component one or more managers to ensure the administration functionalities. In DCUP[2,3] (**D**ynamic **C**omponent **U**Pdating), each component defines one component manager (CM) and one component builder (CB), that are

responsible of managing the particular component. A component may have several implementation objects and/or sub-components that provide its functionality. A component is divided into two parts: *permanent part* and *replaceable part*. Therefore, it provides two kinds of operations, control operations and functional operations.

Adapting a component means replacing its replaceable part by a new version at run-time. The adaptation process can be summarized as follows: the CM locks adapters and sends an adaptation request to the CB; the CB stops the execution of all functional objects, save their states and destroy the replaceable part; the CM downloads and instantiates the new version of the CB, the new CB builds the component (functional objects and sub-components).

3.1.2. Advantages

- Hierarchical model (powerful).
- Indirect link between objects which facilitates interconnections modification.

3.1.3. Weaknesses

- The indirection decreases the application performance.
- Functional objects are not hidden behind interfaces, in other words, the boundaries between the inside and the outside of a component are not well traced.
- About the adaptation granularity, when a sub-component of a global component has to be adapted, all the global component is affected, and all its replaceable part is redeployed, therefore, we can imagine the consequence if a link between two components at the top level of the application is adapted: all the application will be thus redeployed.
- DCUP does not provide any degree of automation.

3.1.4. Addressed adaptation types

DCUP allows the implementation change, the architecture change by adding, removing components and modifying interconnections.

3.2. OSGi

3.2.1. Presentation

In OSGi[4] all administration information of all

components is maintained and managed by a framework. OSGi defines a component model in which components plug into the framework, therefore, an application can be developed in an incremental fashion at runtime. In OSGi, a component is called a *bundle*; it is a JAR file containing one or more classes and resources that implement one or more services. Each bundle defines an *Activator* class that provides two methods: *start* and *stop* that allow the framework to activate/deactivate the bundle.

A bundle can be dynamically installed, activated (started), deactivated (stopped) and uninstalled.

3.2.2. Advantages

- OSGi is an efficient development environment because components can be added and updated at runtime.
- Powerful event mechanisms supported by the framework.

3.2.3. Weaknesses

- The vision of implementation change in OSGi is simplistic. When the update is instigated, the framework deactivates the bundle to be updated, loads the new classes and calls the start method. The programmer must develop means to stop effectively the running object and at least to release the resources it holds such as files and windows.
- OSGi does not take into account the state of a component (bundle) when it is updated.
- No degree of automation is supported by OSGi.

3.2.4. Addressed adaptation types

OSGi supports the implementation change. About the application's architecture, it is possible to dynamically add new components, however, in case of component removing, no new instances can be created, but existing instances continue to be running.

3.3. OLAN

3.3.1. Presentation

OLAN [5] is an example of transactional based approaches. Its aim is the building, execution and administration of distributed component-based applications. An application is organized hierarchically. The hierarchy nodes are the composite components that constitute the

structuring units. Leaves are the units that encapsulate applicative software (primitive components). At run-time, to each composite corresponds a controller, and to each primitive corresponds an administrable component.

OLAN takes a particular care of the adaptation automation. To perform reconfiguration, OLAN uses a transactional model, the main aim is to ensure the application consistency. Two kinds of transactions are defined, reconfiguration and applicative. If a conflict is detected, a reconfiguration transaction is always authoritative and the other is aborted.

3.3.2. Advantages

- Based on a powerful hierarchical component model.
- Takes a particular care of automation.
- Primitive components can be implemented either in C, C++, or Python programming language. Therefore, primitive components are not dependent of a specific language.

3.3.3. Weaknesses

- Using a transactional model in adaptation decreases the administrated application performance. A component that computes for a long time, may be penalized if its applicative transaction is aborted, and must later restart its computation entirely. We can imagine if many other computations depend on the aborted transaction, they must be aborted too. Consequently this may stop the application for a long time. We think that it is necessary to allow the adaptation in near temporarily points that may be specified by the programmer or discovered by the system.

3.3.4. Addressed adaptation types

The different adaptation types addressed by OLAN are the implementation change (replacing a component by another), and the architecture change (adding, removing components and modifying their interconnections).

4. Our Adaptation model

Our adaptation model defines an *Adaptable* relation that associates to each component one or more other components that may replace it at run-time, as presented in figure 1.

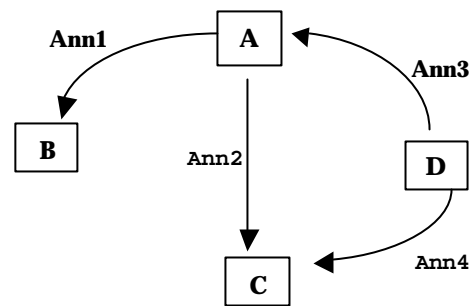


Figure1: Relation 'Adaptable' between components

The annotation Ann1 associated to the relation between A and B describes three information types:

- Constraints that should be satisfied to replace A by B, for example, some strategies impose that the services provided by B must be a superset of those provided by A and the services required by B must be a subset of those required by A.
- Script that describes the mapping between A and B states, as well as the mapping between their methods and how to replace effectively A by B.
- Constraints or actions to be performed after replacing A by B. For example, consider that the a new version of a component requires a new version of other one (dependency between components).

5. Experimentation on JavaBeans component model

5.1. The component model

A Bean is a reusable java software unit that can be visually composed into composite components or applications using visual application builder tools. Beans expose their features (public methods and events) to builder tools for visual manipulation. A Bean's features are exposed because feature names adhere to specific *design patterns*. Beans use *events* to communicate with other Beans. A Bean that wants to receive events (a listener Bean) registers its interest with the Bean that fires the event (a source Bean).

We extended the BeanBox [6] which is a Beans test container (developed by Sun) to support the dynamic adaptation according to our model. We

chose the BeanBox for the availability and the simplicity of its source code.

5.2. The BeanBox architecture

42 Java classes constitute the BeanBox code, figure 2 shows the interaction between the main six classes of the BeanBox.

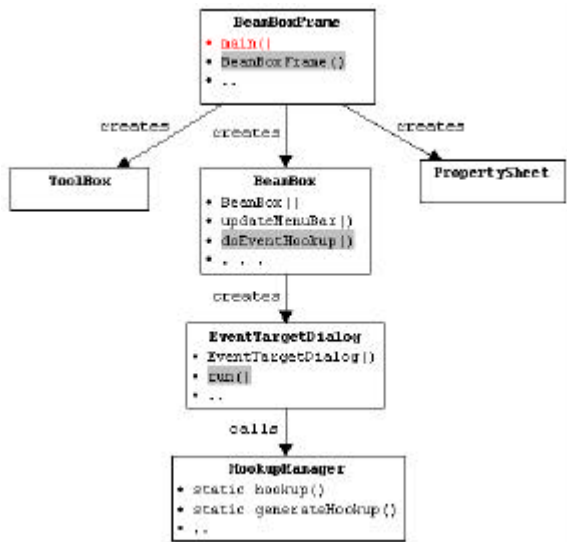


Figure2: The main BeanBox classes

BeanBoxFrame is the entry point of the BeanBox, it instantiates a *ToolBox* that contains the available Beans, a *PropertySheet* that shows the selected Bean properties and a *BeanBox* where the user drops its Beans. To connect two Beans, the user should:

- Select the source Bean.
- Select the listener method in the menu.
- Select the target Bean.

When the target Bean is selected, the *BeanBox* instantiates an *EventTargetDialog* that allows the user to select the target method of the target Bean. After the selection of the target method, the *EventTargetDialog* calls the static method *hookup()* defined in *HookupManager* class to create an adaptor between the source and the target Beans.

5.3. Our DBeanBox architecture

The *DBeanBox* extends the original *BeanBox* and allows to dynamically adapt a running application. Figure 3 shows the architecture of the

DBeanBox.

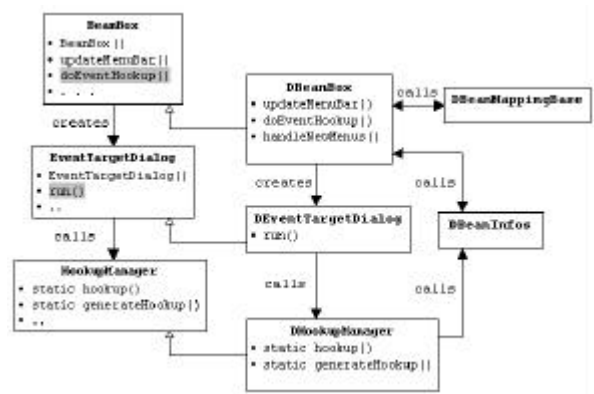


Figure3: The DBeanBox architecture

As shown in figure 3, the *DBeanBox* defines essentially five classes.

- *DBeanBox* extends *BeanBox*:
 - *updateMenuBar()*: defines new menus related to dynamic adaptation operations.
 - *doEventHookup()*: creates a *DEventTargetDialog* instance instead of the old *EventTargetDialog*.
 - *handleNewMenus()*: handles new menus.
- *DEventTargetDialog* extends *EventTargetDialog*:
 - *run()*: when the target method is selected, *run()* calls the static method *hookup()* defined in *DHookupManager* class.
- *DHookupManager* extends *HookupManager*:
 - *hookup()*: calls *generateHookup()* to create an adaptor class before loading the new adaptor in memory.
 - *generateHookup()*: generates a Java file representing the adaptor linking the source and the target Beans. The generated adaptor is widely different than that generated by the standard *BeanBox*. In the next section, we explain the new adaptor capabilities.

5.4. DBeanBox adaptors

The communication between the different Beans is indirect, each two connected Beans communicate via an adaptor automatically generated. Figure 4 shows an adaptor code generated by our *DBeanBox*.

As we can see in the code, the generated adaptor supports many new functionalities related to dynamic adaptation.

- The target Bean is generic and not limited to the current target Bean. This allows to update the target Bean by any other Bean type.
- The target method is not definitively specified and can be adapted dynamically according to the new target Bean and to its target method.

```
// DBeanBox - Extension of Sun's BeanBox
// Automatically generated hookup
public class ___Hookup_17boob15o1 implements ModelListener,
    java.io.Serializable {

    public void setTarget(Object t) {
        target = t;
    }

    public void setTargetMethod(Method m) {
        targetMethod = m;
    }

    public void passivate() {
        active = false;
    }

    public long getCurrentStamp() {
        if(waitingList.isEmpty()) { active = true; return -1; }
        return (long) ((StampedEvent)waitingList.get(0)).stamp;
    }

    public void sendTheCurrentEvent() {
        try {
            targetMethod.invoke(target, new Object[] {
                ((StampedEvent)waitingList.get(0)).argument});
            waitingList.remove(0);
            if(waitingList.isEmpty()) active = true;
        } catch(Exception e) { e.printStackTrace(); }
    }

    public void draw(ModelEvent arg0) {
        try {
            if(active)
                targetMethod.invoke(target, new Object[] { arg0});
            else
                waitingList.add(new StampedEvent(arg0));
        } catch(Exception ex) { ex.printStackTrace(); }
    }

    private Object target;
    private Method targetMethod;
    private boolean active = true;
    private List waitingList = new ArrayList();
}

```

Figure4: DBeanBox generated adaptor

- The adaptor can be either passive or active, it can be passivated thanks to the *passivate()* method.
- When the adaptor is passivated, the received events are stored in a waiting list.

The stored events are stamped, all stamps are automatically generated by a specific class. A stamp is a number (the current time) that specifies the order of the stored event. Passivate a Bean can be performed by passivating all its source adaptors. When the Bean is activated, the stamp allows to fire the stored events in the correct order.

5.5. Beans Mapping Base

As we explained above, each Bean is associated with an 'Adaptable' relation to one or more other Beans that can replace it at run-time. This relation specifies some constraints and the mapping

between the properties and methods of the old and new Beans.

Figure 5 shows an XML script specifying the mapping between two Beans 'Model' and 'NewModel'.

```
<?xml version="1.0"?>
<beansmapping>
  <oldbean> Model </oldbean>
  <newbean> NewModel </newbean>
  <properties>
    <property>
      <old>
        <type> int </type>
        <name> posX </name>
      </old>
      <new>
        <type> char </type>
        <name> abscisse </name>
      </new>
    </property>
  </properties>
  <methods>
    <method>
      <old>
        <returntype> void </returntype>
        <name> left </name>
        <parameters>
          <parameter>
            <type> ControllerEvent </type>
            <name> evt </name>
          </parameter>
        </parameters>
      </old>
      <new>
        <returntype> void </returntype>
        <name> gauche </name>
        <parameters>
          <parameter>
            <type> EventObject </type>
            <name> myEvent </name>
          </parameter>
        </parameters>
      </new>
    </method>
  </methods>
</beansmapping>

```

Figure5: Beans mapping script

5.5. Beans updating

Figure 6 presents the interconnections between Beans via adaptors.

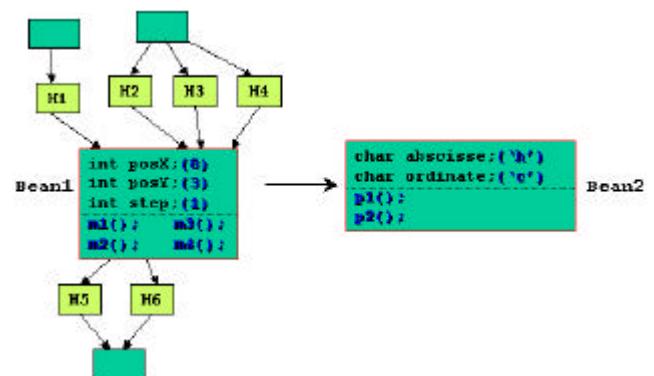


Figure6: Beans mapping script

Updating Bean1 by Bean2 can be performed in many steps:

- Bean1 must be declared updatable.
- If no XML script mapping is specified between Bean1 and Bean2, the user is invited to specify it.
- Bean1 is passivated, therefore, all its source hookups (H1, H2, H3, H4) are passivated.
- If Bean1 and Bean2 are stateful, the state of Bean1 is transferred in Bean2 according to the XML mapping script (using Bean1 getters and Bean2 setters).
- The old target Bean reference in the source hookups (H1, H2, H3, H4) are updated to the new Bean reference.
- The old target methods references are updated to the corresponding new target methods according to the mapping script. After these two last steps, all Bean1 source hookups will point now on Bean2.
- All target Bean2 hookups (H5, H6) should subscribe to Bean2 corresponding events and unsubscribe from Bean1 events.

6. Conclusion

The paper deals with different concepts of the dynamic adaptation problem in the context of component-based applications. Critical non-stop applications should be adapted on the fly and should be affected as minimal as possible during adaptation. Adaptation approaches are different in the adaptation granularity (procedure, module, object, component), in the supported adaptation types (implementation, architecture, interface, geography), and of course in performance (simplicity, duration, automation and consistency). So that it can be adapted, a component must be designed to support administration requests and to participate in its proper adaptation. The paper also gives a preliminary overview of our approach and its application to the JavaBeans component model. Relying on a repository supporting multiple alternate components configurations, we try to answer how to dynamically select the appropriate component based on changes of resources availability at run-time. Our current work involves automating the adaptation process at check pointing situations. When the component reaches an adaptable situation, instantaneous

states are captured and transferred to the new component. The automation process is supported by means of guards and triggers expressed in XML and attached to the adaptation relationship linking two components (upward adaptability). While the research reported here is validated on JavaBeans model, the work is planned to investigate the application of the proposed solution to other component models like EJB, CCM, .NETAb .

References

- [1] R.S. Fabry, "How to design a system in which modules can be changed on the fly", Proc. 2nd Int. Conf. on Soft. Eng., pp. 470-476 (1976).
- [2] Plasil, F., Balek, D., Janecek, R, "DCUP: Dynamic Component Updating in Java/CORBA Environment", Tech. Report No. 97/10, Dep. of SW Engineering, Charles University, Prague, 1997.
- [3] F. Plasil , D. Balek, R. Janecek "SOFA/DCUP: Architecture for Component Trading and Dynamic Updating", Proceedings of ICCDS'98, Annapolis, Maryland, USA, IEEE CS Press, May 1998.
- [4] Open Services Gateway Initiative (OSGi). <http://www.osgi.org/>
- [5] De Palma N., Bellissard L., Riveill M., "Dynamic Reconfiguration of Agent-based Applications", European Research Seminar on Advances in Distributed systems (ERSADS'99), Madeira, Portugal, April 1999.
- [6] JavaBeans Architecture, Sun Microsystems. <http://java.sun.com/docs/books/tutorial/javabeans/>