Adapting applications on the fly

Abdelmadjid Ketfi, Noureddine Belkhatir, Pierre-Yves Cunin

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

1. Introduction

When it is necessary to adapt an application, traditionally, the administrator must stop this application 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 these kinds of systems the adaptation must take place at run-time and the application should not be entirely stopped. Some of the challenges of dynamic adaptation are how to guarantee the consistency of the adapted application, how to transfer the state of the old application into the new one, how to do with the clients when the new application does not provide the same services as the old one, etc.

Several adaptation approaches have tried to provide efficient solutions to solve failures and introduce modifications in a running application by affecting it as minimal as possible. These approaches are different in their performance, their degree of automation, their consistency and in the granularity that they address. The granularity means the basic entity that can be adapted. Many solutions focused on how to replace a procedure, a module or an object in a running application. Our work is concentrated on component-based applications.

In Component-based software engineering, the old notion of developing systems by writing code has been replaced by assembling existing components. Therefore applications would no longer be large monolithic structures but 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**.**

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. The adaptation may also imply the modification of the connections between components, the migration of components from an execution site to another or the modification of the services provided by a specific component. In the remainder of this paper, we start by a presentation of the dynamic adaptation problem followed by a brief description of some existing solutions. Our approach will be presented in the last section.

2. Presentation of the dynamic adaptation problem

In the defect removal community, four reasons of software modification are well known: corrective,

adaptive, extending and perfective reasons. These reasons can apply to dynamic adaptation of applications. In this case the key difference is that modifications must be performed at run-time and without shutting down the system.

- Corrective adaptation: removes the faulty behavior of a running component by replacing it with 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 if it runs correctly. For example, replacing a component by a new one with more optimized implementation.

Several adaptation types may be identified and show how dynamic adaptation can be done:

- 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.
- Deployment-architecture 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 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. For instance, it is necessary to guarantee that data or messages in transit (that have not reached their destination) when the adaptation is performed are not lost.
- Performance: the adaptation event becomes more and more frequent during the running application life cycle, therefore the adaptation 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. Some existing solutions

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

applications.

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 **UP**dating), each component defines one component manager (CM) and one component builder (CB), that are responsible of managing the associated 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, saves their states and destroys 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. Discussion

DCUP associates indirect links between objects, which facilitates interconnections modification. However this indirection decreases the application performance. About the adaptation granularity, when a sub-component of a global component has to be adapted, the whole component is affected, and 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: the entire application has to be redeployed. DCUP does not provide any degree of automation; all adaptation operations must be instigated by the administrator. About the addressed adaptation types, DCUP allows the implementation change, the architecture change by adding, removing components and modifying their 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. A bundle may define 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. Discussion

OSGi is an efficient development environment because components can be added and updated at runtime. It supports powerful event mechanisms.

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 effectively stop the running object and 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 and does not support any degree of automation. OSGi supports the implementation change. About the application 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 authorized and the other is aborted.

3.3.2. Discussion

Unlike the two previous solutions, OLAN 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.

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.

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 Approach

4.1. Description of the adaptation model

All presented approaches like many existing others are based on specific component models. These models associate in advance to each component two different parts: functional part that provides the functionalities requested from the component and the control part, which defines the operations to manage the component life cycle. In other words, the administration operations are defined at design time and the component participates in its own adaptation. Such approaches are suitable for new models that have to be defined but cannot be applicable for existing models that have not be designed to support the dynamic adaptation.

In our approach we do not make any supposition on the component model of the application to be adapted. Thus we try to provide generic solutions independent of a specific model. The question asked before adapting a given component is "what are the components able to replace this component and to provide the needed services?" After the replacing components are identified, the administrator should specify the script that must be applied to effectively replace the component. 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.

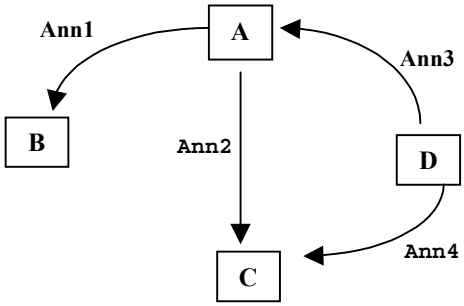

Figure1: Relation 'Adaptable' between components

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 effectively replace A by B.
- Constraints or actions to be performed after replacing A by B. For example, the new installed component requires a new version of another one (dependency between components).

4.2. Experimentation

4.2.1. The component model

One of our experimentations is based on JavaBeans 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. The 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).
In early work, we have extended the BeanBox [6], which is a Beans test container (developed by Sun) to support dynamic adaptation according to our model. The BeanBox generates an adaptor (hookup) to allow the communication between two beans. The class representing the adaptor is generated, compiled and loaded on the fly. We have mapped our solution on the BeanBuilder that represents the new version of the BeanBox. The BeanBuilder uses dynamic proxies to allow communication between components. In the following sections, we explain how we have extended the BeanBuilder with dynamic adaptation mechanisms.

4.2.2. The functional architecture

The functional architecture of the BeanBuilder is completely different of that of the BeanBox. The *BeanTest* class constitutes the entry point of the BeanBuilder; it is composed of three main parts:

- A component panel or a design panel where the user creates its application.
- A component palette that contains in one hand a standard list of components (swing), on the other hand it contains the available components created and loaded by the user. Any component present in the palette can be dragged and dropped into the design panel.
- A status bar that shows the selected component properties.

To connect two components, the user should:

- Select the source component and drag the mouse until the target one.

- The BeanBuilder displays the list of methods belonging to the listener interface, the user should select the listener method in this list.
- After this a list of methods implemented by the target component is shown to allow the user to select the target method that is called when the event occurred.
- A new constraint of the BeanBuilder that does not exist in JavaBeans standard model is that the source Bean must provide public methods that return the list of arguments passed to the target method (if they exist). This list of methods is provided and the user should explicitly specify which method corresponds to which argument.

When all previous information is specified, the BeanBuilder creates a dynamic proxy that implements the listener interface. This dynamic proxy represents the link between the source and the target components.

The BeanBuilder like its predecessor the BeanBox does not support dynamic operations, for example, it does not allow to dynamically replace a component, to perform the state transfer of the old component into the new one, to change existing interconnections between components or to preserve the consistency of the application when a component is removed. Our extension aims to solve these weaknesses.

To keep the model or the application structure, all connections between components are intercepted. A set of information is saved such as the source component, the target component, the listener method, the target method, the reference of the generated proxy, the reference of the object that handles this proxy and so on. All this information is needed to perform the adaptation at runtime.

4.2.3. Extended proxies

The communication between Beans is indirect, each two connected Beans communicate via an adaptor dynamically created. The extended proxy handler supports many 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 dynamically adapted according to the new target Bean and to its target method.
- The adaptor can be either passive or active, it can be passivated thanks to the *passivate()* method.
- When the adaptor is passivated, the received requests (events) are stored in a waiting list.
- Because the waiting list is not global (one waiting list associated to one adaptor), 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. Passivating a Bean can be performed by passivating all its source adaptors. When the Bean is activated, the stamp allows firing the stored events in the correct order.

4.2.4. Beans Mapping Base

Each adaptable Bean is associated with a relation ‘Adaptable’ 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.

4.2.5. Beans updating

Figure 2 presents the interconnections between Beans via adaptors. It shows also how these interconnections can be dynamically modified to perform the adaptation.

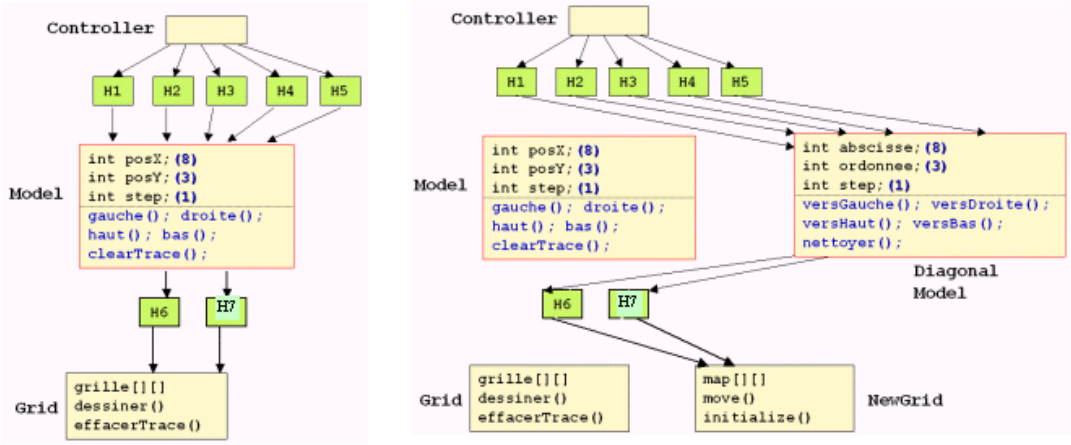

Figure 2: Beans updating

Updating Bean1 (Model) by Bean2 (DiagonalModel) can be performed in many steps:

- Bean1 must be declared updatable.
- If no mapping script 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, H5) are passivated.
- If Bean1 and Bean2 are stateful, the state of Bean1 is transferred into Bean2 according to the mapping script (using Bean1 getters and Bean2 setters).
- The old target Bean reference in the source hookups (H1, H2, H3, H4, H5) are updated with 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 on Bean2.
- All target Bean2 hookups (H6, H7) should subscribe to Bean2 corresponding events and unsubscribe from Bean1 events.

5. Conclusion

The article has described 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, deployment-architecture), and of course in performance (simplicity, duration, automation and consistency). Many existing approaches suppose that a component must be designed to support administration operations and to participate in its own adaptation. These approaches are not suitable for existing models that have not been designed to support dynamic adaptation. We try in our approach to separate the adaptation operations from the component to provide solutions independent from a specific model.

It is very hard to administrate an application if its components do not participate in their adaptation (do not define a control part). In this case, it is necessary to make indirect connections between components to be able to control their communications and to perform dynamic reconnections. It is especially hard to perform the state transfer between the old component and the new one, and to specify what is exactly this state because that depends on the semantic of the application. Therefore, some adaptation operations can be automated and separated from the component and some other

operations needs the participation of the component and must be taken into account at design-time.

References

[1] 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).

[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/