

Automatic Adaptation of Component-based Software

Issues and Experiences

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

This paper deals with the dynamic adaptation problem. By Dynamic, we mean the ability to introduce modifications in an application at run-time. A component-based application is a set of interconnected units. Adapting one or more of its components can enhance such an application. In general, adapting a component means disconnecting it from the application and connecting a new version. The aim of this paper is to highlight the reasons of performing an adaptation, to identify and to classify its different types, and to define the problems and the complexity raised by each type. Dynamic adaptation is not a new problem, the first works go back to 1976[1]. We first introduce a short background of the adaptation problem, followed by the presentation of the different adaptation reasons, types and how to evaluate an adaptation approach. The remainder of the paper presents et evaluates the evolution of the adaptation approach from a large spectrum covering the classical approaches to the recent ones.

Keywords: *Automatic adaptation, component based software, software deployment, taxonomy.*

1. Introduction

Component-based programming refers to software systems assembled from prefabricated components. That means the combination of some aspects of object-oriented programming, safety, and modularity with extensibility as a goal. Different developers might produce assembled components.

The deployment activity is not limited to the installation. It goes along with the software and ensures at any time its consistency and its correct functioning thanks to a process called adaptation. 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. Several problems have to be solved, for example, when the new component does not support the same interface as the old one, or how to adapt interconnections when the adaptation modify the application's architecture. In this paper we present a background of the adaptation problem, introduce and evaluate many adaptation approaches. Our main aim is to highlight the reasons of performing an adaptation, to identify and to classify its different types, to define the problems and the complexity raised by each type.

The paper is organized as follows: section 2

presents the motivations for which an adaptation operation can be performed. In section 3 we classify the different adaptation types that must be addressed by an adaptation approach, followed in section 4 by measures to evaluate a given approach. Section 5 and 6 present and discuss respectively three classical and three recent adaptation approaches. A global synthesis in which we compare and situate the different presented approaches is presented before we conclude in section 7.

2. Motivations

This section presents the reasons for which an adaptation may be performed. These reasons can be classified into four categories:

2.1. Corrective adaptation

If it is noticed that the running application does not have a correct behavior, it is necessary to identify what sub-component of this application does not run correctly and replace it by a new version assumed correct. This new version provides exactly the same functionality as the old one, it just removes the faulty behavior.

2.2 Adaptive adaptation

Even the application runs correctly, sometimes, its execution environment has to be changed (OS, hardware components or other applications on which the application depends). In this case, the application is adapted in response to the changes affecting its environment.

2.3. Extending adaptation

In response to new functionalities needed by the user, and that have not been considered at the application development or deployment time, the application must be extended by adding new components to provide the new needed functionalities.

2.4. Perfective adaptation

The aim is to improve an application even it runs correctly. For example, the implementation of a given component is not optimized enough, so

this component should be replaced by a new one with a new implementation. Another example is to duplicate a component if it is not able to satisfy all requests, and consequently, slowdowns the application.

3. Adaptation Taxonomy

This section presents some of the more important strategies of dynamic reconfiguration of applications categorized by the operation types.

3.1. Adapting the application's architecture

3.1.1. Adding a new component

The new added component should be connected to other components; therefore, adding a new component implies the modification of component's interconnection. It is possible to classify this adaptation into two categories:

- Instantiating the component to be added from a type model already loaded within the application.
- Creating the new component from a type not available in the application.

When it is added to a running application, in general, a component should take into account that the application is not in its initial state, so it should discover its current state and customize itself in order to ensure the global consistency of the application.

3.1.2. Removing an existing component

This operation should not affect the execution of other components; also the removed component must be in a stable state. For example, if a component is modifying a file or a database, it should not be removed before the end of its writing task. Another challenge concerns data or messages exchanged between the removed component and other components; these data or messages should never be lost. Special care is needed before effectively removing the component.

3.1.3. Modifying interconnections between components

In general, when two components are connected,

the types of their connected ports have to match. In a distributed application, the connection between two components, which specifies the type of communication between them, is dependent on their location. In other words, the communication between two local components is different from the communication between two remote components. It is necessary to ensure that no exchanged messages or data are lost.

3.2. Adapting the component implementation

This adaptation is motivated by performance, correction reasons or environmental changes not considered when the component was implemented. For example, replacing for performance reasons a data storage component using a 'vector' by a new version in which an 'hash-table' is used. Interfaces exposed by the component must be maintained the same.

3.3. Adapting the component interface

That means modifying the list of services provided by the component. In component-based software, this can be performed by adding (removing) an interface in (from) the list of interfaces supported by the component.

3.3.1. Adding a new interface

Functionalities exposed through existing interfaces are kept identical and new interfaces are added to provide new needed functionalities.

3.3.2. Removing an existing interface

When a component provides an interface, it must implement it. When a decision can be made that the interface will never be used, it is important for performance reasons to remove it and to remove its implementation.

3.4. Adapting the application geography

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. Also, the internal state of the moved component should be captured and injected into the component in its new location. Messages received and not processed must be taken into account as well.

4. Measuring dynamic adaptation

Multiple requirements must be satisfied to accurately perform an adaptation. In this section, we present the criteria to perform a comparison between different adaptation approaches, and to decide which one is more appropriate.

4.1. Application 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. This consistency can be local or global. In the first case, it concerns one component independently of others. For instance, a component is made in an instable state when one of its local resources is altered.. The global consistency concerns the whole application; for example, the global application data in traffic or messages should never be lost or modified. The adaptation operation should never hang because this implies that the running application hangs or leads it to incorrect behavior. Even when the adaptation operation fails, it must leave the running application in a consistent state, and support mechanisms to rollback the performed modifications if it is necessary.

4.2. 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.

4.3. 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. However, this is not always possible because some environmental changes may not be captured by the application or have not be anticipated at the time of application development. In this case, a human administrator has to explicitly decide and instigate the adaptation operation.

In the remainder of our paper, we present three classical approaches followed by three recent studies that deal with the adaptation problem. We conclude by a synthesis in which we classify and compare the presented approaches.

5. Classical adaptation Approaches

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.

5.1. Dynamic linking

In one hand, this term refers to the incremental loading of a program having holes that are filled in at load or at run-time such as the use of DLLs. In the other hand, that refers to the extension of a program that has not previously holes with new functionalities by adding new modules. Franz [2] classified the concept of dynamic linking in five categories: load-time indirection, load-time rewriting, run-time rewriting, load-time compilation and the dynamic compilation. This classification depends on when the module is loaded and the manner it is linked to the running program.

5.2. Redundant hardware

Many systems employ frequently redundant hardware to support fault tolerance. Therefore, many approaches exploit this redundancy to perform the adaptation. In general, to perform a run-time adaptation, a secondary standby machine is loaded with the new software version, it recovers the state from which it must start from the primary machine and finally, switches over to become the primary machine when the previously primary one switches to become secondary. The airline messaging

system ACARS [3] is an example of systems using this approach.

5.3. State transfer

The state transfer means capturing the running program state before it is stopped. When at a later time the program is reactivated, it restarts with the saved state. In general, it is necessary to identify the persistent states to be transferred, to provide means to encode and to decode the state, and to design the application in a fashion to be able to start with the decoded state. State transfer is used to implement the process migration like in [4] and in [5].

There is much work on adaptation. In the following section we present three recent approaches. Many classical ones such as PODUS, Argus, Conic are presented and compared in [6].

6. Recent adaptation approaches

In this section, we present three component-based approaches, i.e. the changing unit is the component.

6.1. Managers-based approaches

Many adaptation approaches associate to each component one or more managers to ensure the administration functionalities. In DCUP[7] (**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.

Figure 1 presents a typical structure of a DCUP component. A component is divided into two parts: *permanent part* and *replaceable part*. Therefore, it provides two kinds of operations, control operations and functional operations. The heart of the permanent part is the CM when the key of the replaceable part is the CB. The component functionality is provided by functional objects. These objects could be reached only via *wrappers* attached to the permanent part.

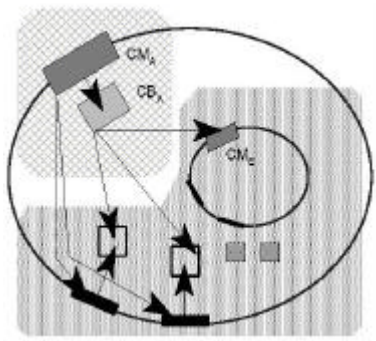


Figure 1: DCUP component model

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).

Discussion and Evaluation

DCUP is based on a powerful hierarchical component model. DCUP uses wrappers as a mechanism to avoid direct link between objects and to facilitate interconnections modification. However, the system must buy the cost of the indirection at any call, which may decrease its performance. About the used model, if all functional objects were hidden behind interfaces, it could be no necessary to manage individual objects names and to associate a wrapper to each object. Also, an object outside the component must not specify the name of the object that provides the service it requests like in DCUP. This is due to the fact that the boundaries between the inside and the outside of a component are not well traced.

Another weakness concerns 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. About auto-adaptation, DCUP does not provide any

degree of automation. Any adaptation initiative must be decided and executed by a human administrator. The addressed adaptation types are: the implementation change, the architecture change by adding, removing components and modifying the interconnections.

6.2. Framework-based approaches

In these kinds of approaches all administration information of all components that compose the application is maintained and managed by a framework, OSGi [8] is an example of such approaches. It 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 service is a public interface exported explicitly in the manifest associated to the JAR file. Manifest contains component's metadata such imports, exports and activator-class.

Thus, the framework manages the services it hosts and the bundles life cycle. When a bundle is started, it receives from the framework a *BundleContext*, which is an object allowing the bundle to register and unregister its services, to retrieve references to other services exposed by other bundles and to subscribe to events published by the framework. A bundle can be dynamically installed, activated (started), deactivated (stopped) and uninstalled.

In OSGi specification, the update is divided into two steps: (i) *update()*: stops the bundle, retrieves new bundle JAR file and restarts the new bundle, (ii) *refreshPackages()*: done by administrator or another bundle framework service called Package Manager.

Discussion and Evaluation

OSGi is an efficient environment that facilitates the development of network-based services, especially because components can be added and updated at runtime. This facility is increased by the event mechanisms supported by the framework. The vision of implementation

modification 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. If the updated objects are stateful, the programmer must also develop means to save and restore states. To resolve these weaknesses, it is possible to extend OSGi with a state transfer or a check-pointing mechanism. One interesting work is described in [5], it allows even the control flow (thread) persistence. About the changes in 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.

6.3. Transactional-based approaches

OLAN [9] is an example of transactional-based approaches. Its aim is the building, execution and administration of distributed component-based applications. It provides an ADL and a set of graphical tools to describe the application's architecture. Its component model is presented in figure 2.

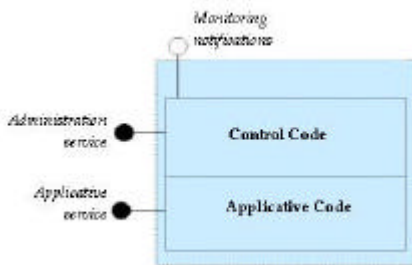


Figure 2: OLAN component model

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, after development, an application follows many steps: (i) *configuration*: components instantiation, initialization and starting, (ii) *monitoring*: spying component states, performance and communication, (iii)

reconfiguration: modifying the internal structure of the running application.

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 (extension of 'wait or die' algorithm). Due to the lack of space, we invite the reader to look at [10] for more details.

Discussion and Evaluation

OLAN is based on a component model expressing applications as a hierarchy of interacting components, it covers all the life cycle of applications, from development to administration. 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 ensures the global consistency of the application, but decreases its 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. In this case, the system must provide means to save and to restore the component state respectively before and after the adaptation. 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).

6.4. Synthesis

The three presented recent approaches are classified according to the taxonomy and some important measuring criteria discussed in section 3 and 4 (see table 1).

Table 1: Synthesis of presented approaches

	Automation	Reliability	State Transfer	Addressed adaptation types			
				Structure	Implementation	Interface	Geography
DCUP	No	No	Yes	Yes	Yes	No	No
JES	No	No	No	Yes	Yes	No	No
OLAN	Yes	Yes	No	Yes	Yes	No	No

7. Conclusion and perspectives

It is difficult to define a generic solution to dynamically adapt any type of software. The work presented in this paper intends to clarify 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, geometry), and of course in performance (simplicity, duration, automation, consistency,...). So that it can be adapted, a component must be designed in a fashion to support administration requests and to participate in its proper adaptation.

Our work is a part of a large project on component-based software deployment. Our component model is an extension of CCM. Our study constitutes an essential step that allows us to precise our needs in the context of our component model. The next step is the extension of our model in order to support the adaptation process and to experiment the results of our study.

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] M. Franz, "Dynamic linking of software components", IEEE Computer, 30(3): 74-81, March 1997.
- [3] ARINC, Inc. Transportation Communications and Systems Engineering. <http://www.arinc.com>
- [4] S. Fünfroeken, "Transparent Migration of Java-based Mobile Agents (Capturing and Reestablishing the State of Java Programs)", Proceedings of Second International Workshop Mobile Agents 98 (MA'98), Stuttgart, Germany, September 1998. <http://www.informatik.tu-darmstadt.de/~fuenf>
- [5] S. Bouchenak, "Un Service pour la Mobilité et la Persistence des Applications Java", 3ème Colloque International sur les NOuvelles TEchnologies de la Répartition (NOTERE'2000), Paris, France, November 21st-24th 2000.
- [6] M. E. Segal and O. Frieder "On -the- fly program modification: Systems for dynamic updating", IEEE Software, pages 53-65, March 1993.
- [7] Plasil, F., Balek, D., Janeczek, R, "DCUP: Dynamic Component Updating in Java/CORBA Environment", Tech. Report No. 97/10, Dep. of SW Engineering, Charles University, Prague, 1997.
- [8] Open Services Gateway Initiative (OSGi). <http://www.osgi.org/>
- [9] Bellissard L., De Palma N., Freyssinet A., Herrmann M., Lacourte S., "An Agent platform for Reliable Asynchronous Distributed Programming", in Proc. of Symposium on Reliable distributed Systems (SRDS'99), Lausanne, Suisse, October 1999.
- [10] N. De Palma "SERVICES D'ADMINISTRATION D'APPLICATIONS REPARTIES" Thèse de doctorat, université Joseph Fourier, Grenoble, Novembre 2001.