

A metamodel-based approach for the dynamic reconfiguration of component-based software

Abdelmadjid Ketfi, Nouredine Belkhatir

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

Abstract. Non-stop and highly available applications need to be dynamically adapted to new conditions in their execution environment, to new user requirements or to some situations usually unpredictable at build-time. Bank, aeronautic, mobile and Internet applications are well known examples of applications requiring the dynamic reconfiguration. On the other hand the development complexity and cost constitute an important problem for the creation of applications supporting to be dynamically reconfigured. The work we present in this paper is centered around the dynamic reconfiguration of component-based applications. It is dedicated to describing DYVA, a virtual dynamic reconfiguration machine. The virtual aspect of DYVA means its independence from a particular application or a particular component model, which enhances its genericity and its reusability.

Keywords. Dynamic reconfiguration, Component model, Metamodel, DYVA.

1 Introduction

For some critical and highly available applications 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 reconfigure a running application to take into account some new conditions, sometimes unpredictable, without completely stopping it. In general the dynamic character of an application is proportional to its complexity, in other words, the more we want a system dynamic, the more we must complete a complex development work.

The work we present in this paper is centered around the dynamic reconfiguration of component-based applications [1]. It is dedicated to describing DYVA, a virtual dynamic reconfiguration machine. In this paper we consider that an application *configuration* is the set of components forming this application and their interconnections. A *reconfiguration* can be then defined as any operation whose role is to modify the initial configuration. An operation can be for example the disconnection of components, the creation of new connections, the modification of existing connections (reconnection), the addition or the removal of components or the replacement of components or groups of components. A *dynamic reconfiguration* can then be de-

fined as a reconfiguration performed on a running application without fully stopping it and whilst preserving its consistency.

This paper is organized as follows: in Section 2 we discuss our motivations for the notion of a virtual dynamic reconfiguration machine. Section 3 is dedicated to describing DYVA and before we conclude, Section 4 explains through example how to use DYVA to support the dynamic reconfiguration of a component-based application.

2 Motivations and objectives

Our motivations for the idea of a virtual machine to support the dynamic reconfiguration of component-based applications have two facets: the importance of the *dynamic character* of applications and the *virtual aspect* of our dynamic reconfiguration machine.

The dynamicity is in general relevant for some classes of critical, non-stop and highly-available applications where the continuity of service is very important. We need to dynamically reconfigure a running application for many reasons:

- The clients of this application need a continuous service and do not support the interruption of the functionalities provided by the application like in aeronautic and real-time applications.
- The execution environment changes frequently and in this case it is impractical to stop the application every time to take into account the new parameters. For example a multimedia presentation application is closely related to the variation of the bandwidth. Such an application must adapt its presentation policy according to this bandwidth.
- Stopping/restarting the application requires a lot of effort or it decreases the quality of service. A large distributed application, for example, requires complex and tedious work to be stopped, updated, rebuilt and correctly restarted, therefore it is easier and more preferable to perform the required modifications without completely stopping the application.

It is important to note that an application may be reconfigured for other reasons such as those discussed in [2]. In the previous paragraph we were interested only in the reasons for which the reconfiguration must be done dynamically. In [3] other examples of domains where the dynamic reconfiguration is needed are given.

In spite of the importance of dynamicity, it remains one of the key challenges facing software developers today. To develop an application that can be dynamically reconfigured the developer must deal with the reconfiguration code instead of focusing on its application logic. In [4] we presented and evaluated many approaches dealing with the dynamic reconfiguration of component-based applications. Approaches like [5,6] associate to each component a specific part dedicated to its management and responsible for providing the necessary functionalities for its dynamic reconfiguration.

DYVA, which stands for “*DYnamic Virtual Adaptation machine*”, is our solution to support the dynamic reconfiguration of component-based applications. It is the

result of several works which we completed previously [4,7,8,9]. DYVA is intended to be generic which means its independence from a particular application or a particular component model. The idea is to start from a concrete application, that can be instrumented, and to create an image of this application according to a canonical model. At run-time the canonical representation can be dynamically reconfigured which causes the reconfiguration of the concrete application (the canonical representation and the application are causally connected).

3 DYVA: our dynamic virtual reconfiguration machine

3.1 Overview of our approach

Recently we proposed and prototyped a dynamic reconfiguration framework for JavaBeans-based applications [7,10]. This framework enables one to dynamically reconfigure a JavaBeans-based application with the minimal participation of the user. In the context of another project, the same work has been done for OSGi-based applications [8,11], this second work allowed us particularly to validate the dynamic reconfiguration process proposed in [7].

After working on two different component models, namely JavaBeans and OSGi, some basic concepts seemed to be apparent and shared by both models. For instance the dynamic reconfiguration routines and process. This observation was a key element that motivated us to work on a virtual dynamic reconfiguration framework.

3.2 DYVA logical architecture

The global architecture of DYVA is presented in Figure 1.

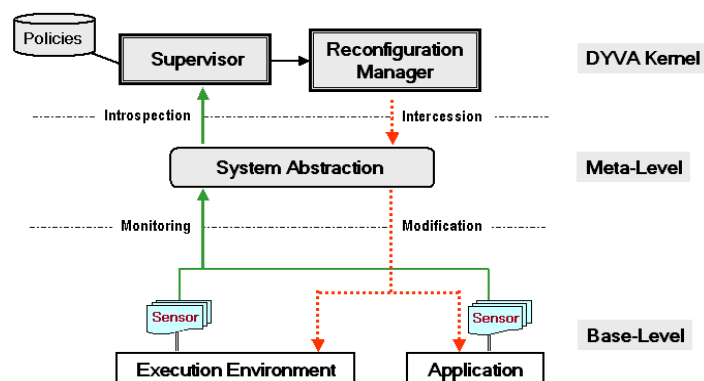


Fig. 1. DYVA global architecture
As shown in Figure 1 three parts can be distinguished:

- The base-level: represents the concrete application that provides the expected functionalities.
- The meta-level: it is transparent to the application users and can be seen as an abstract representation of the concrete application.
- DYVA kernel: the different operational modules of the dynamic reconfiguration machine.

3.2.1 Causal connection

It is important to note that the meta-level is not only an independent snapshot of the base-level but it is causally connected to it: modifying the meta-level causes a corresponding modification to the base-level, and vice versa. Monitoring and modification operations realize the causal connection between the base-level and its abstraction. They assure that the modifications occurring at one level are propagated in the other level in order to guarantee that each level gives an accurate view of the other. The causal connection is an important concept in reflective architectures [12].

3.2.2 Sensors

Sensors are the concrete elements that assure the monitoring. They are responsible for observing the application and its execution environment and sending up the relevant events occurring at the base-level. By relevant we mean a change at the base-level concerning an entity represented at the meta-level, for example if a property of a component instance is modified, an event is sent up only if this property is forming part of the instance state or representing an interaction point (receptacle), otherwise no event is needed. Notification events are only limited to information represented at meta-level for performance reasons. There are two kinds of sensors:

- Environment sensors: a sensor of this category is a module permanently executed (as a daemon) and controlling a specific environment resource (disk, memory, load, bandwidth...).
- Application sensors: an application sensor is an agent belonging to the application code, and responsible for informing the meta-level when a significant event occurs in the application. In the current version an event is sent in the following situations:
 - Component instance created: an event specifying the reference of the created instance is fired.
 - Method called: the user can chose to send an event for each called method or to select only some methods.
 - Property affected: if a property is forming part of a component instance state, an event is sent up if its value changes. If the property represents an interaction point between two instances and its value is changed, that means that the application architecture has been modified. An event is then sent up.

3.2.3 Supervisor

The role of the supervisor is to allow the auto-reconfiguration of applications. It represents a reasoning engine that introspects or receives events from the system abstraction and takes decisions according to these events and according to some re-configuration policies. A decision may have one of the following forms:

- Invocation of the reconfiguration manager operations, for example if some component instance is no longer available, the supervisor may ask the reconfiguration manager to create a new instance of the same component and to reconnect it in the place of the unavailable one.
- Delegation of the decision to an external actor if the supervisor is not able to reason about the received event and to take an appropriate decision. In this case the supervisor displays a message describing the situation and a human administrator, for example, must intervene to take a decision.

3.2.4 Reconfiguration policies

Some policies are required to make possible the auto-reconfiguration of applications. Auto-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.

3.2.5 Reconfiguration manager

The reconfiguration manager is the central part of the reconfiguration framework. It provides and implements the basic dynamic reconfiguration routines. These routines operate on the system abstraction (and not directly on the base-level) which guarantee the independence and the genericity of our machine. In the following points we describe the main operations implemented by the reconfiguration manager, all these operations can be applied at the abstract level then propagated thanks to the causal connection to the base-level:

- *Removing a component instance*: a component instance could be removed only if it is in a stable state. For example, if an instance is modifying a file or a database, it should not be removed before the end of its writing task. When a component instance is removed all its connections with other instances are systematically removed.
- *Removing a component*: removing a component instance does not affect other instances of the same component. Sometimes instead of removing one instance, it is hoped to remove a component. In this case the component and its resources are unloaded and all its instances are removed.
- *Dynamic disconnection*: two kinds of dynamic disconnection are supported by the reconfiguration manager:

- Port-level disconnection: the administrator must specify the source and the target ports concerned by the disconnection.
 - Instance-level disconnection: all the provided and required ports of the instance are concerned by the disconnection.
- *Dynamic connection*: to create a new connection the administrator must specify the required port and the provided port to be connected. These two ports must be compatible to allow the connection operation.
 - *Dynamic reconnection*: the dynamic reconnection uses the disconnection and connection operations described above. It is a fundamental operation which is useful for all other reconfiguration operations. The reconfiguration manager provides two kinds of dynamic reconnection:
 - Port-level reconnection: only one port is concerned by the reconnection.
 - Instance-level reconnection: all the provided and required ports of the instance are concerned by the reconnection.
 - *Creating a new component instance*: the extension or the adaptation/correction of the application functionalities may require the creation of new instances.
 - *Replacing a component instance*: for performance or correction reasons we need sometimes to replace one component instance by another one more suitable for the application requirements. The instance replacement is a complex process composed of a set of ordered activities that can be summarized as follows:
 - Creating the replacing instance if it does not already exist.
 - Passivating the old instance which means stopping all ingoing requests sent to this instance. The stopped requests should be saved and treated later to complete the reconfiguration operation.
 - Transferring the state of the old instance to the new one for consistency reasons.
 - Reconnecting the new instance in the place of the replaced one.
 - Activating the new instance and potentially removing the old one.

The state transfer is a very complex problem, in this paper the state of a component instance is considered simply as a subset of its data-structure values. These data-structures have to be accessible to be able to read the state of the replaced instance and to write it in the new one. Other complex aspects of the state transfer have not been addressed like inaccessible data-structures and the execution state (threads).

4 How to use DYVA in practice

DYVA can be used in two different ways: declaratively or programmatically.

- Using DYVA declaratively: a description of each component forming the application is necessary. A dedicated tool uses this description provided by the user and instruments the application according to it.
- Using DYVA programmatically: a description of the application components is also required in this case, however, the developer has to explicitly use an API provided by DYVA. Therefore in the second case, no instrumentation tool is needed.

In this section we will describe only the declarative usage of DYVA. We first show through an example how to create the description file of a simple application. Then we describe the instrumentation process and we focus particularly on the modifications introduced on the application during this process.

4.1. Application description

The first step before using DYVA is to provide a description of the application to be reconfigured. This description must be done according to the canonical model presented in the previous section. According to the application component model, it is possible to build semi-automatic description tools responsible for analyzing applications and discovering partially their description. If the targeted component model provides explicitly the application description in a given format (ADL for example), a transformation tool can be used to map this description to the canonical model.

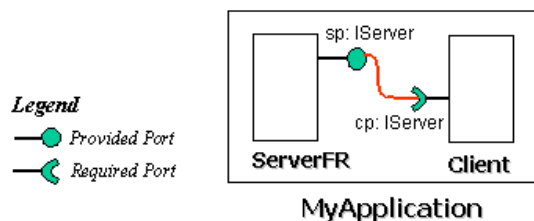


Figure 2. Application example

The description file of the application example presented in Figure 2 looks like the following.

```
- <Repository>
- <Components>
  - <Component name="Client">
    - <RequiredPorts>
      <RequiredPort name="cp" type="IServer" />
    </RequiredPorts>
    <Implementation name="example01.Client" />
  </Component>
  - <Component name="ServerFR">
    - <ProvidedPorts>
      <ProvidedPort name="sp" type="IServer" />
    </ProvidedPorts>
    <Implementation name="example01.ServerFR" />
  </Component>
</Components>
</Repository>
```

For simplification reasons, in the description file many details have been omitted like reconfiguration dependencies between components, reconfiguration compatibilities and other metadata describing the internal structures forming the state of each component.

It is important to note that the previous file describes only the components and not the application architecture. Information about the application architecture in terms of component instances and their interconnections are discovered at run-time. The response to how this can be done is given in the following section.

4.2. Application instrumentation

After having the application description, an instrumentation tool is mainly used to introduce some modifications in the application code. In the following we show the instrumentation process of the application example we presented above. This application example has been implemented in OSGi component model.

After the instrumentation, the following modifications are introduced in each component implementation:

- In each constructor, adding a call to DYVA to inform it when an instance is created.
- A call to DYVA is inserted at the beginning and at the end of public methods. This is optional and can be parameterized by the user to select only some methods or to call only at the beginning or only at the end of some methods.
- If a property is forming part of a component instance state, a call to DYVA is inserted to inform it if the property value is modified. If the property represents an interaction point between two instances (required port), a call to DYVA is then inserted to inform it if the property value is modified (that means that the application architecture has been modified).
- For each required port two methods are needed to connect and to disconnect the port. If these methods are not provided by the developer, they are inserted automatically by the instrumentation tool.

A call to DYVA means simply sending an event with the appropriate information to the meta-level. This allows the application architecture to be dynamically created.

4.3 Reconfiguring the application at run-time

After it has started, the first task the instrumented application transparently does is to create the control environment that allows to show graphically its architecture as illustrated in Figure 3 (the control environment is created after firing the first event of the application). This environment allows also to control and to dynamically reconfigure the underlying application.

Any reconfiguration operation acts on the meta-level and is systematically propagated to the concrete application thanks to the causal connection. Using the reconfiguration interface, the administrator can graphically reconfigure the running application.

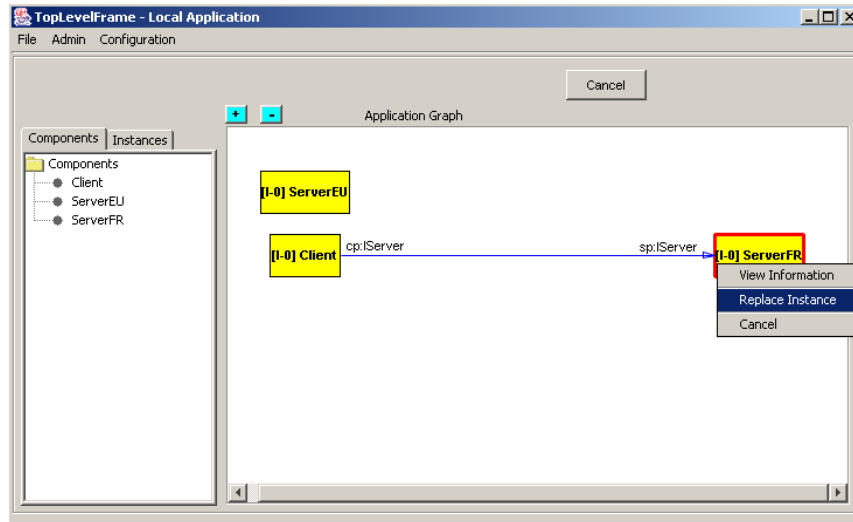


Figure 3. Reconfiguration interface

For example in Figure 3, a new component (“*ServerEU*”) has been added to the application, an instance of this component is created (“[I-0] *ServerEU*”), and as illustrated by the contextual menu, the old “*ServerFR*” instance will be replaced (by the new “*ServerEU*” instance).

5 Conclusion

A dynamically reconfigurable application is an application which supports some modifications in its architecture and behavior at run-time without being fully stopped, and whilst preserving its integrity and an acceptable quality of service. This property is very important for some classes of critical, highly available and non-stop applications and constitutes one of the key challenges facing software developers today. This paper presented DYVA, a generic support that assists developers to build dynamically reconfigurable applications and provides means to administrators by which to reconfigure these applications at run-time. Our approach, intended to be generic and reusable, serves as a support to enhance applications with the dynamic capabilities and provides means to dynamically reconfigure these applications at run-time. The automation is an important aspect of our approach which simplifies the creation of applications endowed with dynamic capabilities on one hand, and gives the ability to take automatically some reconfiguration decisions according to a set of strategies on the other hand.

The results we obtained after the development of DYVA allow us to conclude that an approach driven by the abstraction of technical and specific properties of applications seems to be very promising. The next step to follow the work we presented in this paper has three main facets: stabilizing and extending our current prototype,

enlarging the set of reconfiguration operations currently supported and extending the work to other component models. All these facets are needed to validate our approach and to prove its feasibility.

6 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. T. Mens, J. Buckley, M. Zenger, A. Rashid, Towards a Taxonomy of Software Evolution, International Workshop on Unanticipated Software Evolution, Warsaw, Poland, April 2003.
3. S. Sonntag, H. Härtig, O. Kowalski, W. Kühnhauser, and W. Lux, Adaptability Using Reflection, In proceedings of the 27th Annual Hawaii International Conference on System Sciences, pages 383–392, 1994.
4. A. Ketfi, N. Belkhatir and P.Y. Cunin. Automatic Reconfiguration of Component-based Software: Issues and Experiences, PDPTA'02, June 2002, Las Vegas, Nevada, USA.
5. 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.
6. 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.
7. A. Ketfi, N. Belkhatir and P.Y. Cunin, Dynamic updating of component-based applications, SERP'02, June 2002, Las Vegas, Nevada, USA.
8. A. Ketfi, H. Cervantes, R. Hall and D. Donsez, Composants adaptables au dessus d'OSGi, Journées Systèmes à Composants Adaptables et extensibles October 2002, Grenoble, France.
9. A. Ketfi, N. Belkhatir, Dynamic Interface Adaptability in Service Oriented Software, Eighth International Workshop on Component-Oriented Programming (WCOP'03), July 21, 2003 - At ECOOP 2003, Darmstadt, Germany.
10. JavaBeans Architecture, Sun Microsystems.
<http://java.sun.com/docs/books/tutorial/javabeans/>
11. Open Services Gateway Initiative (OSGi).
<http://www.osgi.org>
12. W. Cazzola, A. Savigni, A. Sosio, and F. Tisato, Architectural Reflection: Concepts, Design, and Evaluation, Technical Report RI-DSI 234-99, DSI, Università degli studi di Milano, May 1999.