A Generic Language for Dynamic Adaptation

Assia Hachichi[1], Gaël Thomas[1], Cyril Martin[1],
Bertil Folliot[1], and Simon Patarin[2]

[1] LIP 6 - Université de Paris6
{Assia.Hachichi,Gael.Thomas,Cyril.Martin,Bertil.Folliot}@lip6.fr
[2] DSI - Università di Bologna
patarin@cs.unibo.it

Abstract. Today, component oriented middlewares are used to design, develop and deploy distributed applications easily. They ensure the heterogeneity, interoperability, and reuse of software modules.
Several standards address this issue: CCM (CORBA Component Model), EJB (Enterprise Java Beans) and .Net. However they offer a limited and fixed number of system services, and their deployment and configuration mechanisms cannot be used by any language nor API dynamically.
As a solution, we present a generic high-level language to adapt system services dynamically in existing middlewares. This solution is based on a highly adaptable platform which enforces adaptive behaviours, and offers a means to specify and adapt system services dynamically. A first prototype[1] was achieved for the OpenCCM platform[2], and good performances were obtained.

1 Introduction

Computing systems are increasingly complex and difficult to maintain. Moreover, the various elements, that constitute an environment, are often physically distributed on heterogeneous nodes. Middlewares were introduced to solve these difficulties, by proposing common generic system mechanisms to the distributed applications.

The last generation of component oriented middlewares introduces the component and container concepts. A container manages system services, such as persistence, transaction, security or naming, in a way that is transparent for business code, which is encapsulated in components.

The adaptation of system services is often done statically, by stopping the middleware execution, which induces a high cost for critical applications. For this reason the dynamic adaptation is more efficient. Some platforms provide mechanisms that can be used to adapt services dynamically. Nevertheless, these mechanisms are specific to the targeted platforms: they are not reusable on

[1] This work was partially financed by the European project IST-COACH (2001-34445).
[2] OpenCCM is an implementation of the CORBA Component Model specification [1].

J.C. Cunha and P.D. Medeiros (Eds.): Euro-Par 2005, LNCS 3648, pp. 40–49, 2005.

other middleware platforms easily. Moreover, there is no standard nor model that unifies adaptation mechanisms independently of platforms.

We propose to use a domain-specific language (DSL) [2] technique: a programming language providing high-level abstractions related to a given domain. The expertise captured in the language allows behaviours to be expressed in an intuitive and high-level manner, permits verification, and allows generation of efficient code that is automatically integrated in the target platform.

In this context, our work proposes the Container Virtual Machine (CVM) approach, which defines a generic adaptation language. The CVM includes a DSL for writing adaptation behaviours. Each adaptation need is described on CVM language then is translated to different targeted platform language on the fly, in an automatic way. This approach allows separation between the adaptation logic and its implementation, by providing a high-level language.

A first translator has been implemented on the OpenCCM platform [3], an open source implementation of the CORBA Component Model (CCM) specification defined by the Object Management Group (OMG). This prototype allows the adding and reconfiguring of new system services, and offers administrators the possibility to specify and deploy system properties dynamically even if they were not taken into consideration initially.

In the following, Section 2 presents other proposals allowing to make middlewares flexible. Then, our proposal for offering high level language for dynamic adaptation is detailed in the Section 3. Section 4 describes the CVM and examples implementation, and Section 5 presents the conclusion of our work.

2 Related Work

Several component-based models exist such as: Microsoft .Net, Sun Microsystems Enterprise Java Beans, or OMG CORBA Component Model. These models are used to design and to deploy distributed applications. However, they do not allow easy integration and adaptation of system services[3] dynamically. Moreover, no standard envisages describing the integration and adaptation of services after initial deployment of the application.

The first middlewares were not designed to be flexible. However, adaptation techniques have been proposed, such as interceptors, and Portable Object Adaptor (POA) in CORBA ([4]). The interceptors [3] allow inserting code before the reception and after sending a request. The POA allows programmers to construct object implementations that are portable between different ORB products.

Several projects aim at making CORBA more flexible. DynamicTAO [5] (based on TAO), a reflexive CORBA environment, reifies the internal elements of the ORB in the form of components called configuration components. DynamicTAO keeps a compatibility with CORBA applications, by offering a high degree of adaptability. One of the difficulties that this project raises is the problem of coherence when a policy is replaced by another.

[3] System services such as: transaction and replication service.

AspectIX [6] adopts a fragmented object model based middleware [7]. The fragments can mask the replication of a distributed object, impose real-time constraints on the communication channel, put the object information in memory cache, etc. These non-functional (system) aspects can be configured via a generic interface of the object. Each global object can be configured by a profile that specifies the aspects that the fragments must respect. Four profiles are planned, in particular a CORBA profile that allows for these AspectIX objects to interact with CORBA objects. This approach allows a clear separation between the application and the middleware over which it is deployed.

JAC (Java Component Aspect) allows to weave an aspect dynamically: the relation between the wrappers and the advice codes can be redefined on the fly. However, the number of pointcuts is not extensible dynamically: if the class is already charged in the virtual machine, there is no means to add a new pointcut.

An architecture of open containers is proposed in [8]. This architecture allows dynamic adaptation and extension of the system functions, and it allows exposing some number of container properties, using control , interception and coordination mechanisms. OpenORB [9] is a flexible architecture of component oriented middleware. OpenORB is based on the reflexion. Each Object of the system is associated to a meta-space which offers structural representation. The ORB is configured or reconfigured by using the Meta-Object protocol. JavaPOD [10], is a component model which allows attaching system properties to the components. This attachment is achieved by means of open and extensible containers. Comet [11] is an events based middleware. It can be adapted by inserting pre/post hooks into the components. A language is associated to dynamic reconfiguration of the Comet middleware. However this language is not extensible dynamically, and is not generic since it is applicable only for Comet.

These various projects increase the middleware adaptation possibilities by re-coding it. Our work takes different direction, we propose a generic high-level language to adapt the system services dynamically in existing middlewares. Each adaptation behaviour is described on this high level language then is translated on all targeted platform language, in an automatic way. This abstract description allows separation between the adaptation logic and its implementation.

3 Container Virtual Machine Approach

Instead of providing adaptation behaviours that depend on the middleware platform, the Container Virtual Machine approach defines a generic language, which gives a high-level abstraction of system services adaptation behaviours, that is independent on the middleware platform. The abstraction behaviours are translated on the targeted platform, in automatic and dynamic way.

This approach allows (i) the unification of adaptation behaviours, independently on the targeted platform, (ii) the automatic generation of CVM scripts can be achieved by a design tool, and (iii) the generation of platform independent adaptation models (PIM - Platform Independent Model) and them translation on the trageted platform (PSM - Platform Specific Model).

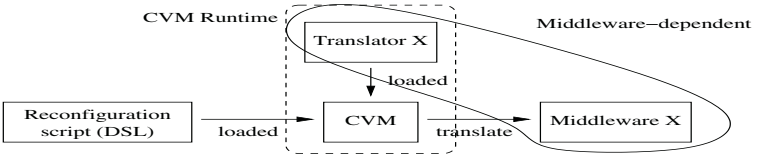

Fig. 1. CVM Concept (Container Virtual Machine)

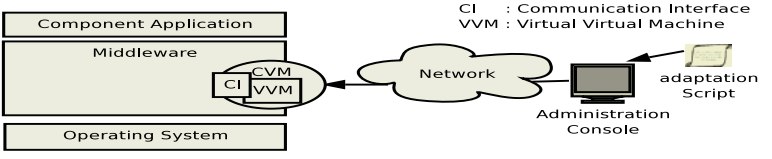

Fig. 2. CVM Processing (Container Virtual Machine)

3.1 CVM Design

The CVM approach aims to remain neutral with respect to the platform, and to separate the adaptation language from its execution. Figure 1 presents the CVM concept; its input is a configuration script, called a translator, which is dependent on middleware. The translator enables to translate an adaptation script written in CVM language for a specific middleware.

3.2 CVM Implementation

The main idea is to add an entry point to different platform middleware, at the initial deployment and in a transparent way. This entry point enables the interaction between the CVM platform and its targeted middleware platform, and is called a *Communication Interface (CI)*. It enables the translation of abstractions to the targeted middleware language. The CVM is mainly based on a *highly adaptive platform* to describe and to enforce the adaptation behaviours. This platform is generic with respect to middleware platforms, and interacts with each middleware platform through its associated *Communication Interface* (fig 2). The CVM allows to define new adaptation operations on the fly.

Virtual Virtual Machine: The CVM design requires a *highly adaptable language* to provide separation between the adaptation logic and its implementation, and to extend the access operations in order to enforce what can be adapted. The selected highly adaptable platform is the *Virtual Virtual Machine (VVM)* [12], which is a dynamic code generator that provides both a complete, reflexive language, and an execution environment. The VVM allows to modify the implemented mechanisms, to reconfigure the environment, and to extend or modify the associated language.

The main objectives of this environment are: (i) to maximize the amount of reflective accesses and intercessions, at the lowest possible software level, while

preserving simplicity and efficiency; (ii) to use a common language substrate to support multiple language and programming paradigms.

To achieve this, the VVM provides four basic services: (i) code generation: a fast, platform- and language-independent dynamic compiler producing efficient native code that adheres (by default) to the local platform's C ABI (Application Binary Interface); (ii) meta-data that are kept between compilations, thus allowing higher-level software to reason about its implementation or that of the environment, and modify them dynamically; (iii) introspection on dynamically-compiled code, the application and the environment itself; (iv) input methods, giving access to the compilation and configuration process at all levels.

The execution model is similar to C and the dynamically-compiled code has the same performance as a statically compiled and optimized C program. In the context of the CVM, we added a server which receives Abstract Syntax Tree from another VVM, compiles and links them, and executes the generated code.

The use of the VVM allows the separation of the adaptation logic from its implementation. This language must be both *extensible* to new adaptive needs dynamically, and *generic* with respect to the targeted middleware in order to ensure its reusability. It allows both to reduce the possibilities of reconfiguration by limiting the language symbols, and/or to extend the language by providing introspection of the environment and the creation of new symbols.

Remote Administration of the CVM: In order to ensure the adaptation of several network nodes from a remote administration console, we built a remote adaptation environment in the VVM platform; this environment must be loaded on all VVMs. It parses/lexes scripts that reconfigure the target environment; these scripts are transformed into abstract syntax trees. These trees are sent to the VVM, which is able to receive them on a communication channel (example: a TCP socket). These trees are then compiled and executed on the second entity.

In the adaptation context, a client opens a communication channel, and parses/lexes VVM scripts that reconfigure the remote machine (server), then sends the corresponding trees to the server. When a server receives the abstract syntax trees, it compiles and executes them.

4 Qualitative Evaluation

The CVM is evaluated on a CORBA Component Model implementation written in Java: the OpenCCM [3]. This section details the prototype implementation.

4.1 OpenCCM Translator Implementation

In the case of the OpenCCM platform, the translator is achieved by using a Java native method, which launches the VVM. The communication Interface (CI) between the VVM and the standard JVM is provided by JNI (Java Native Interface [13]). JNI is an interface between the native functions and the Java virtual machine.

The VVM is executed by a Java thread in competition with those of the application. The language of the VVM is then dynamically extended: the scripts written for the VVM can then interact with the JVM directly, and the VVM is able to handle the methods and the symbols of the Java application (Fig 2).

A reconfiguration comprises two important steps:

1. The first phase consists in building methods that allow dynamic adaptation into the VVM; for example: methods that integrate or remove components.
2. The second consists in writing a CVM script that contains the adaptation needs. This script is loaded remotely by the administration console and is executed by using a CI. Scripts can either extend the reconfiguration language, or use the keywords already built in to modify the OpenCCM application.

To illustrate the use of the CVM, two examples of reconfigurations are presented in the rest of this section.

4.2 Integration of Service

We classify the system services in two classes: not-intrusive services, which do not modify the treated data, and the intrusive services, which modify the data, and requires synchronization

In this paper we present two examples for integrating services, one is a monitoring service that is not-intrusive and the other one is an encryption service that is intrusive. These integrations are based on the *Portable Interceptors* and on *System Oriented Component* respectively.

Flexible Monitoring Service: The first example illustrates the dynamic integration of a flexible monitoring service based on interceptors.

This service was designed to collect statistics on the way components interact with each other, and to make this information available to a "reconfiguration service" that will use it to adapt the platform.

The monitoring service is composed of two concurrent processes. The first one collects all available information concerning the called requests, and records them in a log file. The second process scans the log generated by the first process periodically, and calculates the statistics of the call number and the average response time for given operations. The integration of this service is based on CORBA portable interceptors.

The CORBA specification [14] defines the portable interceptor interface as a way to insert hooks directly inside the ORB. These hooks are activated for every operation performed by the broker: mainly method invocations and result returning. Hooks may be located either on the client or on the server side. We conclude that the integration of the monitoring service on the level of interceptor hooks, allows to invoke the monitoring service code at every request by extracting several metrics, such as the number of times a specific method is invoked, and sums all the invocations of methods belonging to the same component. However, no standard language or interface enable to use Portable Interceptors dynamically to achieve an integration of System Services. For this reason the CVM is

used to integrate the monitoring service dynamically. This integration comprises two phases: (i) to specify, in the VVM platform, the new adaptation operations that allow adding code in the OpenCCM interceptor hooks dynamically. (ii) to write and to execute a VVM script that integrates service code in hooks through the Communication Interface.

Encryption Service: Considering an application, that contains two components "A" and "B", included in containers "CA" and "CB" respectively (see figure 4). Component "A" sends messages to "B" in a regularly way. During the execution, the administrator decides to send encrypted messages to "B". In order to achieve this, we use another mechanism to intercept requests: *System Oriented Components (SOC)*. This mechanism is used because it is generic; it can be applied for any middleware, and shows that it is possible to define other integration mechanisms on the fly.

The System Oriented Component mechanism consists in adding CCM components which containing the service code to be added, and in establishing the necessary connections with the components to which this service will apply.

In our example, the integration of an encryption service consists in integrating an encryption SOC component in container "CA", and a decryption SOC component in container "CB". Basically the integration consists in adding the necessary operations to the VVM, such as the operations which enable SOC component creation and handling the connections between any components. The second step is to write and execute the VVM script which allows us to: (i) add the encryption SOC in "CA" and another one in container "CB", (ii) disconnect "A" and "B"; (iii) establish the connections between "A", "B" and their respective SOC (see figure 4 and 3), by ensuring the synchronization. Problems that can occur are: encoded messages may be received before adding a decryption SOC, or non-encrypted messages will be sent after adding the decryption SOC.

1. (On_container CA
   – (Deactivate_Component A))
2. (On_container CB
   – (Deactivate_Component B))
3. (On_container CA
   – (Disconnect_components A B)
   – (Insert_SOC SocA)
   – (Connect_components A SocA))
4. (On_container CB (Insert_SOC SocB)
   – (Connect_components SocA SocB)
   – (Connect_components SocB B))
5. (On_container CA
   – (Activate_Component A))
6. (On_container CB
   – (Activate_Component B))

1. <programme>→<reconfiguration>*
2. <reconfiguration>→<SOC>*| <PI>*
3. <SOC>→ (On_container<atomeCont> <action>*)
4. <action>→ (<subaction><atome>)
5. <subaction>→Deactivate_Component| Disconnect_components <atome> | Insert_SOC | Connect_components<atome> |Activate_Component
6. <atomeCont>→ *Containerreference.*
7. <atome>→ *Compoenentreference*
8.

Fig. 3. (A) An example of the reconfiguration script that integrates encryption SOC (B) A part of the CVM grammaire.

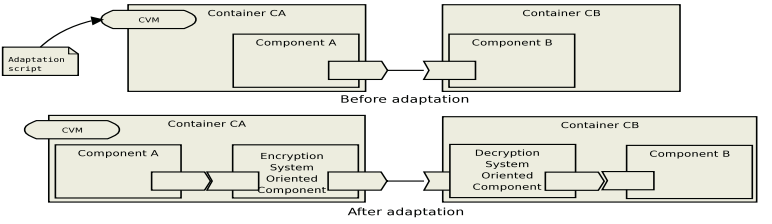

Fig. 4. Integration of the encryption service.

To avoid these problems, the synchronization must be ensured dynamically. A pseudo-algorithm that is proposed consists in deactivating "A" before "B", breaking the connections between "A" and "B", and then adding the encryption and decryption SOC in container "CA" and "CB" respectively, finally establishing the necessary connections, and activating "B" before "A".

In the case where the targeted middleware does not provide the possibility to activate or deactivate a component, we propose to use a queue. Messages from "A" will be redirected towards the queue, during integration of encryption and decryption service. Synchronization is not yet implemented in our encryption prototype.

4.3 Adaptation of the Encryption Service

To illustrate the adaptation of existing services, we adapt the encryption service of the previous example, during the execution.

Component behaviour adaptation can be achieved by replacing a component by a new one. However, it is simpler and less expensive to adapt a component by replacing some of its methods.

In the case of the encryption SOC adaptation, it is enough to adapt the Java method that contains the encryption service. The Java standard allows dynamic loading of a class and overload of the serialization methods. By coupling the Sun Java platform and CVM, we can adapt a Java method. Let us take the example of method "metA" from class "A", the adaptation of this class is done by charging a new class "A1" which inherits from "A", and which implements the new code of "metA", then redirecting all calls towards the new loaded method (Fig 5).

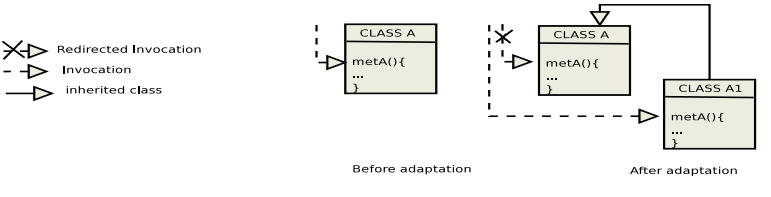

Fig. 5. Method adaptation.

4.4 Discussion

Two adaptation examples were presented; one is based on the SOC approach which is generic in the sense that it can be applied for any component-oriented middleware. However, the created SOC must have compatible ports with existing components, and the SOC code is compared to service code that it contains.

The second example is based on the Portable Interceptors (PI) approach, and is not generic, since the PI concept does not exist in all component-oriented middlewares, with EJB. But this code size is smaller than the SOC code size.

A Set of ten performance evaluation measures of the dynamic integration were performed, on a Pentium III 664MHz under Linux. These measures represent the duration between the old configuration and the new configuration after the end of service integration. (i) *The monitoring service integration average duration*, which is based on the portable interceptors, is 8.539 seconds. (ii) *The encryption SOC integration average duration* is 2.054 seconds.

Another set of ten SOC adaptation duration measures were done. This duration represents the time between the initial configuration and the end of the encryption code replacement. The average of the ten measures is $94 * 10^{-3}$ seconds.

We note that the integration based on Portable Interceptors is slower than service based on System Oriented Components. This can be explained by the cost resulting from the flowing of all requests through the interceptor layer. In [15], which studies three different ORB implementations, it is shown that the activation of portable interceptors increases latency by a factor varying between 2% and 10% and decreases request throughput by a factor ranging from 1.5% to 16%, This cost is then limited.

We note that the adaptation duration average is slower than integration service duration. However, these costs remain limited.

5 Conclusion

This paper presents the Container Virtual Machine, a platform which allows dynamic adaptation of system services and provides a generic language specific to adaptation domain (DSL) . This language offers a high-level abstraction of adaptation behaviour and is itself extensible. Adaptation CVM scripts can be translated for different target platforms during the execution automatically.

The CVM approach provides a separation between the adaptation logic and its implementation. CVM language is generic in the sense that it is independent from the middleware to be adapted. It language enables to describe any new adaptation and the related operations. It allows an adaptation remote administration which provides interoperability and synchronisation between several nodes; it can be aperated on different middleware platforms, such as EJB and CCM. The provided high-level abstractions are translated automatically for the targeted platform.

As future works, we aim to reuse the CVM language on the different platforms, such as EJB, then to refine the grammar of our DSL. To provide means

that ensure the coherence, atomicity, and verification of the dynamic adaptations and of there deployment. To achieve the automatic generation of CVM scripts design tool such as Rationalrose, then to offer mechanisms that execute models automatically.

References

1. Openccm user's guide (2004) http://openccm.objectweb.org/doc/0.8.1/user_guide.html.
2. Lawall, J., Muller, G., L.P.Barreto: Caputing os expertise in an event type system: the bossa experience. In: Tenth ACM SIGOPS European Workshop (EW 2002), France, Springer-Verlag (2002) 154–61
3. OMG: Interceptors Published Draft with Corba 2.4+ Core Chapters. (2001) Document Number ptc/2001-03-04.
4. Daniel, J.: Au coeur de Corba. (2001)
5. Kon, F., Román, M., Liu, P., Mao, J., Yamane, T., Magalhães, L.C., Campbell, R.H.: Monitoring, Security, and Dynamic Configuration with the dynamicTAO Reflective ORB. In: Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'2000). Number 1795 in LNCS, New York, Springer-Verlag (2000) 121–143
6. Hauck, F.J., Becker, U., Geier, M., Meier, E., Rastofer, U., Steckermeier, M.: AspectIX: An aspect-oriented and CORBA-compliant ORB architecture. Technical Report TR-I4-98-08, Univ. of Erlangen-Nuernberg, IMMD IV (1998)
7. Makpangou, M., Gourhant, Y., Narzul, J.P.L., Shapiro, M. In: Fragmented objects for distributed abstractions. IEEE Computer Society Press (1994) 170–186
8. Vadet, M., Merle, P.: Les conteneurs ouverts dans les plates-formes à composants. Journées composants: flexibilité du système au langage (2001)
9. Blair, G.S., Costa, F.M., Coulson, G., Duran, H.A., Parlavantzas, N., Delpiano, F., Dumant, B., Horn, F., Stefani, J.B.: The Design of a Resource-Aware Reflective Middleware Architecture. In: Proceedings of the Second International Conference on Meta-Level Architectures and Reflection, France, Springer-Verlag (1999) 115–134
10. Bruneton, E., Riveill, M.: Javapod: une plate-forme à composants adaptables et extensibles. Rapport technique 3850, Inria Rhone-Alpes (2000)
11. Peschanski, F., Briot, J.P., Yonezawa, A.: Fine-grained dynamic adaptation of distributed components. Middleware 2003 (2003) 132–142
12. Ogel, F., Thomas, G., Piumarta, I., Galland, A., Folliot, B., Baillarguet, C. In: Towards Active Applications: the Virtual Virtual Machine Approach. A92 Publishing House, POLIROM Press (2003) 28–47
13. Liang, S.: The JavaTM Native Interface: Programmer's Guide and Specification. Addison Wesley Longman (1999)
14. OMG: Corba / iiop specification 3.0. formal/024206 (2002)
15. Marchetti, C., Verde, L., Baldoni, R.: Corba request portable interceptors: a performance analysis. In: Proceedings of the 3rd International Symposium on Distributed Objects and Applications, Rome, Italy (2001)