Mixing Aspects and Components for Grid Computing

Jean-Marc Menaud, Jacques Noyé, Pierre Cointe
OBASCO project, EMN-INRIA, LINA
Ecole des Mines de Nantes
4, rue Alfred Kastler - BP 20722
F-44307 Nantes Cedex 3

Christian Pérez
PARIS project, IRISA
Campus Universitaire de Beaulieu
Avenue du Général Leclerc
F-35042 Rennes Cedex

Abstract

It is generally accepted that software architecture design should support required properties such as robustness, reusability, and adaptability. To ensure such requirements, it is also commonly believed that the associated relevant concerns should be identified at design time. Aspect-Oriented Software Development (AOSD) is an emerging technology revisiting the principle of separation of concerns. The main idea behind AOSD is that explicit abstractions and mechanisms should be provided in order to separate, model, and compose (or weave) concerns that tend to cross-cut multiple system components. Then, from a software engineering perspective, the challenge is to localize and specify concerns as individual and reusable components. Following the “object tradition”, a number of software architectures have been proposed to explicit fundamental architectural abstractions leading to Component-Based Software Design (CBSD). This approach has been advocated by OMG when designing the CORBA Component Model (CCM) dedicated to distributed computing. More recently, GridCCM has extended CCM by introducing parallel components dedicated to Grid Computing.

This paper presents preliminary work on mixing AOSD and CBSD in the context of Grid applications. It introduces the problems raised by dynamically plugging aspects into components and shows how this dynamic composition can be supported by introducing aspects in CCM components, based on an extension of Arachne, a dynamic weaver for C.
1 Introduction

Component-Based Software Design (CBSD) is a line of research that enforces the design and development of applications based on the development and interconnection of components as black boxes. The interconnection of components is achieved by the usage of ports. Ports are used to describe the services that one component provides to the outer world and the services it uses from this outer world. For example, one component can define the interfaces it provides, the events it publishes, as well as the interfaces it uses and the events it consumes. Components are deployed into an environment called a component container along with a specification of how ports are to be connected.

This approach intends to boost software development by means of high reuse, clear interface definition, and encapsulation of code, but it suffers from the same limitations as the Object-Oriented approach when trying to implement functionalities that cut across the encapsulation hierarchy of the application. This kind of functionalities, where the implementation expands over a number of software units (objects, components, procedures) are called crosscutting concerns and are addressed by the research line known as Aspect-Oriented Software Design (AOSD). The Aspect-Oriented approach aims to improve the modularization of crosscutting concerns by defining units of software that, while declared in a centralized way, are introduced or woven into the execution flow of the application cutting across the encapsulation hierarchy of the system. This approach follows a white-box model in order to weave aspects into the internals of software units, therefore breaking the encapsulation principle. The trade-off between the black-box approach of CBSD and the white-box approach of AOSD is at the heart of the CBSD/AOSD integration problem.

The goal of the present work was to make a step towards understanding the relationships between the CBSD model and the AOSD model, looking for ways to integrate these models. Our approach was to work on one specific, but important, use case: dynamically adding a port to a component. This was achieved using concrete CBSD and AOSD technologies: Mico CCM [3], the implementation of a CCM ORB [6] written in C++, and Arachne [1], a dynamic weaver for C applications. One possible scenario consists of dynamically adding a port to a component assembly in order to fulfill monitoring requirements over the behavior and running state of each component in the assembly.

2 CCM Overview

The CORBA Component Model (CCM) [6] is an extension to the CORBA object model [7], conceived by the OMG. The CCM specification is a major part of the CORBA 3.0 standard, released in 2002. CCM enforces distributed component-based software design using CORBA as its middleware infrastructure.

Components are the building blocks of a CCM system. Developers define components using IDL 3.0 language constructs, which are compiled into CORBA objects, packaged into dynamic libraries (JAR, DLL) or executables, and deployed in a component container. Containers are the runtime environment for a component on the server side.
In CCM, a component may have attributes, may support interfaces, and may have any of four types of ports. Attributes allow higher reuse of components by encapsulating general behavior in attribute values that can be configured at deployment time, or runtime. The use of interfaces is similar to the use of Java interfaces. Ports make it possible to express interconnectivity between components. Ports allow one component to declare the interfaces it uses and provides, as well as the events it consumes and publishes. Component ports are interconnected later on, in an assembly stage, before deployment, which results in a programming model driven by the assembly of prefabricated components rather than application development. The interconnection of components may be done using an assembly tool provided by the CCM implementation provider.

3 Arachne for CCM

The Arachne prototype is developed by the OBASCO research team [1]. It initially worked on legacy C applications, and was further extended in order to support C++ and conduct the experiment we report in this paper. Arachne allows the definition of aspects capable of intercepting function invocation and variable access. This makes it possible to extend and modify the behavior of an application. Classically (see for instance [5]), an aspect basically consists of two parts: a pointcut, which tells where in the execution of the base program new behavior should be added, and an advice, which is this new behavior. In Arachne, an advice over a function application is similar to an around advice in AspectJ (the de facto AOP standard [2]): the advice replaces the function invocation. The weaving of aspects in the base program is achieved dynamically, giving the possibility of turning aspects on and off at execution time. The following sections give an overview of the definition of aspects and weaving in Arachne. After introducing our specific aspect language for CCM, we present the architecture of Arachne. The last section focuses on the extension of Arachne to deal with C++ applications, and Mico, a C++ implementation of a CCM ORB.

4 An Aspect-Oriented Language for CCM

This section presents an aspect-oriented language dedicated to components. It is inspired by the CCM model and all the examples are based on this model. The language introduces an aspect construct, meant to centralize the implementation of crosscutting concerns. This construct acts over a set of components and makes it possible to react to internal events by external actions or to react to external events by internal actions. In order to do so, the component has somehow to be a white box in order to access to some details of its internal execution both in terms of (internal) events and in terms of (internal) actions. This access is however performed in a disciplined and well-modularized way.

The declaration of an aspect has the general form shown in Fig. 1. The add keyword changes the structure of the component whereas the modify keyword only changes its behavior. A detailed description of the effect of each keyword follows.
add [receptacle|facet|attribute] TYPE IDENTIFIER to COMPONENTSET

modify [receptacle|facet] COMPONENTSET.PORTSET
when POINTCUT
[around|before|after [EXCEPTIONIDENTIFIER]]
{ C++ code };

Figure 1: Aspect definition

add Adds one port or attribute to a set of components. If the added port is of type receptacle, the definition of a pointcut is added in order to invoke the new receptacle as a reaction to some component behavior or state change. If the added port is of type facet, some source code is added. This source code will be executed upon facet invocation. If the added port is of type attribute, the type and identifier of the added attribute must be declared. Code may be added to specify attribute setters and getters, otherwise default setter and getters are generated.

modify A modification over a receptacle implies intercepting the client side of the request response flow, while a modification over a facet implies server-side interception. A modification can be of three types: around, before, or after, à la AspectJ. An around advice replaces an invocation by an alternative implementation. Before and after advices intercept the communication flow, before or after the request is sent or received. The when clause makes it possible to define a pointcut restricting the modification of the port to certain component states or behaviors.

5 Arachne

The implementation of this language is based on dynamic weaving, implemented by an extension of Arachne, as described in the following two sections. We first describe the architecture of Arachne, as a dynamic weaver for C, and then show how Arachne was extended in order to apply to CCM components.

5.1 The Architecture of Arachne

The architecture of Arachne is shown in Fig. 2. It is composed of three parts: the aspect runtime environment, a kernel manager, and an aspect compiler. The aspect compiler translates the aspects into C code that is compiled with gcc. The runtime environment is responsible for the actual aspect weaving (and unwrapping): it rewrites, at run time, the
compiled code of the base program interpreted by the processor such that the advices are executed at the appropriate join points, that is, the execution points in the base program that belong to the aspect pointcuts.

The main component of the runtime environment is the Arachne kernel dynamic link library (DLL). As the runtime environment is responsible for weaving aspects in the base program at runtime, it has to be able to rewrite the binary code of the base program and thus needs to be loaded in the same address space. Once the kernel DLL is loaded in the address space, it creates a thread in the base program process, waiting for weaving and unweaving requests. By using a thread for processing the weaving and unweaving requests, the execution of the base program does not have to be suspended. Upon reception of a weaving request, the Arachne kernel loads the corresponding aspect DLL. Then the kernel loads the rewriting DLLs required by the aspect if necessary. A rewriting DLL serves as an API that provides functions to rewrite/instrument one kind of join point (for instance, function calls). Metadata DLLs are used to ensure the independence between the aspects and the base program. They contain a mapping between the symbolic description of rewriting sites and the actual rewriting sites in the binary code of the base program. In case of unweaving, the kernel instructs the rewriting DLLs referenced by the aspect to restore the original code before unloading the aspect DLL.

The kernel manager serves as a link between the user who wants to weave and unweave aspects into different applications, and aspect kernels that actually weave and unweave the aspects into specific base programs.

The compiler first translates an aspect written in the aspect language into a C source
file that contains both the advices as executable functions and dynamic predicates, which test whether an execution point belongs to the associated pointcut. In a second step, the compiler generates a compiled aspect DLL by using a regular C compiler (gcc).

5.2 Extension of Arachne to C++

We now consider how we extended Arachne for dynamic weaving into C++ applications and therefore into Mico CCM components.

C++ can be seen as a typed object-oriented extension of C providing function overloading and overriding, instance variables, and compile-time code generation facilities (template). To ensure proper interoperability between compilers, the compiled representation of a C++ file has been normalized [4, 8]. Except from the language features specific to C++, this standard closely follows the ANSI C specification. Therefore, the techniques used in Arachne to instrument C programs are directly applicable to C++ programs and only the features specific to C++ require further considerations and will be discussed in the following.

C++ implements function overloading by encoding the types of the signature in the function name. This encoding process is defined by the standard and allows tools such as GNU nm to retrieve the exact, source level, name from the encoded, binary level, function name. By using this property, Arachne properly handles overloaded functions.

Function overriding in C++ is implemented using vtables [4, 8]. The C++ compiler translates the invocation of virtual functions into binary code that first retrieves the address of the function to be executed from the vtable, before actually executing it. In addition, the C++ compiler holds each vtable as a global variable. Therefore, to trigger the execution of an action upon a virtual function, Arachne replaces the address stored in the vtable by the address of the advice.

Finally, C++ compile-time generation facilities do not interfere with Arachne for C++. Arachne however is not able to trigger advice on template functions, since these computations are performed at compile time and their results are inlined in the compiled executable. This is not surprising as Arachne for C is not able to trigger advice upon inlined C functions.

6 Conclusion

The goal of our work is to design an extension of the CCM model that retains the generality of aspect-oriented languages while keeping the power and flexibility of more vertical approaches in the CCM model. It is with this goal in mind that a proposal for an aspect-oriented language for components has been presented in this paper. This language is based on the concept of an aspect as an entity that acts over a set of components. These components lose their strict black-box property, but in a disciplined way. Through a dedicated aspect language, Arachne can be used to dynamically alter the behavior of the components by modifying either their implementation or their interface.
Future work includes developing a proof of concept of the aspect-oriented language, based on a non trivial scenario and concrete Grid application, making use of the mechanism developed in this work for dynamically adding a port to a component. For example, we propose to transform the visual monitoring of the HydroGrid application into an aspect. The HydroGrid application aims at the modeling and simulation of fluid and solution transport in subsurface geological media. We plan to study aspect integration with GridCCM, an extension of CCM introducing parallel components dedicated to Grid Computing. An advantage of the HydroGrid application is that there is a CCM as well as a GridCCM implementation of it.

References


