Automatic Code Generation for Non-Functional Aspects in the CORBA–$\mathcal{LC}$ Component Model*

Diego Sevilla[1], José M. García[1], Antonio Gómez[2]

[1] Department of Computer Engineering
[2] Department of Information and Communications Engineering
University of Murcia, Spain
{dsevilla, jmgarcia}@ditec.um.es, skarmeta@dif.um.es

Abstract. Component Technology allows a better modularity and reusability of applications. Components are even better suited for the development of distributed applications, as those applications can be partitioned in terms of components installed and running (deployed) in the different hosts participating in the system. Components, apart from implementing their own functionality, have other requirements in term of non-functional aspects such as CPU power utilization, load balancing, fault-tolerance, etc. The code for ensuring these aspects can be automatically generated based on the requirements stated by components and applications, thus leveraging the component implementer of having to deal with these non-functional aspects. In this paper we present (1) architecture of the automatic code generator, and (2) the characteristics of the generated code for dealing with the aforementioned non-functional aspects in the context of CORBA–$\mathcal{LC}$.CORBA–$\mathcal{LC}$ is a lightweight distributed reflective component model based on CORBA that imposes a peer network model in which the whole network acts as a repository for managing and assigning the whole set of resources: components, CPU cycles, memory, etc.

1 Introduction

Component-based development[1], resembling integrated circuits (IC) connections, promises developing application connecting independently-developed self-describing binary components. These components can be developed, built and shipped independently by third parties, and allow application builders to connect and use them.

As applications become bigger, they must be modularly designed. Components come to mitigate this need, as they impose the development of modules that are interconnected to build the complete application. Components, being binary, independent and self-described, allow:

* Work under Grant TIC2003-08154-C06-03.

- Modular application development, which leads to maximum code reuse, as components are not tied to the application they are integrated in.
- Soft application evolution and incremental enhancement, as enhanced versions of existing components can substitute previous versions seamlessly, provided that the new components offer the required functionality. New components can also add new functionality to be used by new components, thus allowing applications to evolve easily.

When component technology is applied in a distributed environment, programmers can develop components that interact transparently with other components residing in remote machines. However this makes applications and components management harder.

CORBA *Lightweight Components* (CORBA–$\mathcal{LC}$)[2], is a distributed component model based on CORBA[3]. While traditional component models force programmers to decide the hosts in which their components are going to be run (deployment) using a "static" description of the application (*assembly*), CORBA–$\mathcal{LC}$ performs the deployment and component dependency management automatically. Thus, it offers the traditional component models advantages (modular applications development connecting binary interchangeable units) allowing automatic placement of components in network nodes, intelligent component migration and load balancing, leading to maximum network resource utilization. CORBA–$\mathcal{LC}$ introduces a more *peer* network-centered model in which all node resources, computing power and components can be used at run-time to automatically satisfy applications dependencies.

The paper is organized as follows: Section 2 offers an overview of CORBA–$\mathcal{LC}$. Section 3 outlines the design of the CORBA–$\mathcal{LC}$ Code Generator. Section 4 shows how automatic code can be generated to seamlessly and transparently offer fault-tolerance to component implementations. Finally, Section 5 offers related work in the fields of component models and aspects, and Section 6 presents conclusions, status and future work.

2 The CORBA–$\mathcal{LC}$ Component Model

CORBA Lightweight Components (CORBA–$\mathcal{LC}$) [2] is a lightweight component model based on CORBA, sharing many features with the CORBA Component Model (CCM)[4]. The following are the main conceptual blocks of CORBA–$\mathcal{LC}$:

- **Components**. Components are the most important abstraction in CORBA–$\mathcal{LC}$. They are both a *binary package* that can be installed and managed by the system and a *component type*, which defines the characteristics of component instances (interfaces offered and needed, events, etc.) These are connection points with other components, called ***ports***.

– **Containers and Component Framework**. Component instances are run within a run-time environment called **container**. Containers become the instances view of the world. Instances ask the container for the required services and it in turn informs the instance of its environment (its *context*).
– **Packaging model**. The packaging allows to build self-contained binary units which can be installed and used independently. Components are packaged in ".ZIP" files containing the component itself and its description as IDL and XML files.
– **Deployment and network model**. The deployment model describes the rules a set of components must follow to be installed and run in a set of network-interconnected machines in order to cooperate to perform a task. CORBA–$\mathcal{LC}$ deployment model is supported by a set of main concepts:

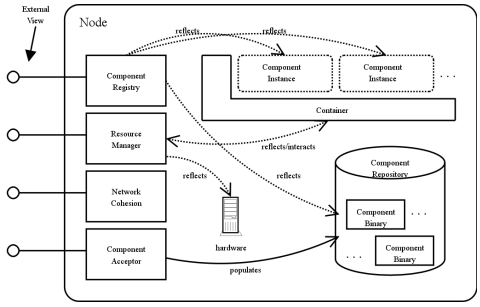

Fig. 1. Logical Node Structure.

- **Nodes**. The CORBA–$\mathcal{LC}$ network model can be seen as a set of nodes (hosts) that collaborate in computations. Nodes maintain the logical network connection, encapsulate physical host information and constitute the external view of the internal properties of the host they are running on. (Fig. 1). Nodes offer information about memory and CPU load, as well as the set of components installed.
- **The Reflection Architecture**. Is composed of the meta-data given by the different node services:
  * The ***Component Registry*** provides information about (a) running components, (b) the set of component instances running in the node and the properties of each, and (c) how those instances are connected via ports (assemblies),
  * the ***Resource Manager*** in the node collaborates with the ***Container*** implementing initial placement of instances, migration/load balancing at run-time.
- **Network Model and The Distributed Registry**. The CORBA–$\mathcal{LC}$ deployment model is a network-centered model: The complete network is considered as a repository for resolving component requirements.

- **Applications and Assembly**. In CORBA–$\mathcal{LC}$, ***applications*** are just special components. They are special because (1) they encapsulate the explicit rules to connect together certain components and their instances (*assembly*), and (2) they are created by users with the help of visual building tools. Thus, they can be considered as ***bootstrap*** components.

3 Dealing with Aspects: The CORBA–$\mathcal{LC}$ Code Generator

Components are not only a way of structuring programs, but a framework in which the programmer can concentrate in the functionality only, rather than other aspects such as reliability, fault tolerance, distribution, persistence, etc. For those aspects (called *non-functional aspects*), CORBA–$\mathcal{LC}$ as a component technology allows the programmer to specify those non-functional aspects in a declarative manner. Thus, the programmer has to write the functionality of the component and describe the needs in term of those non-functional aspects.

The framework is in charge of ensuring those requirements are met. This means generating the correct code for each component's requirements.

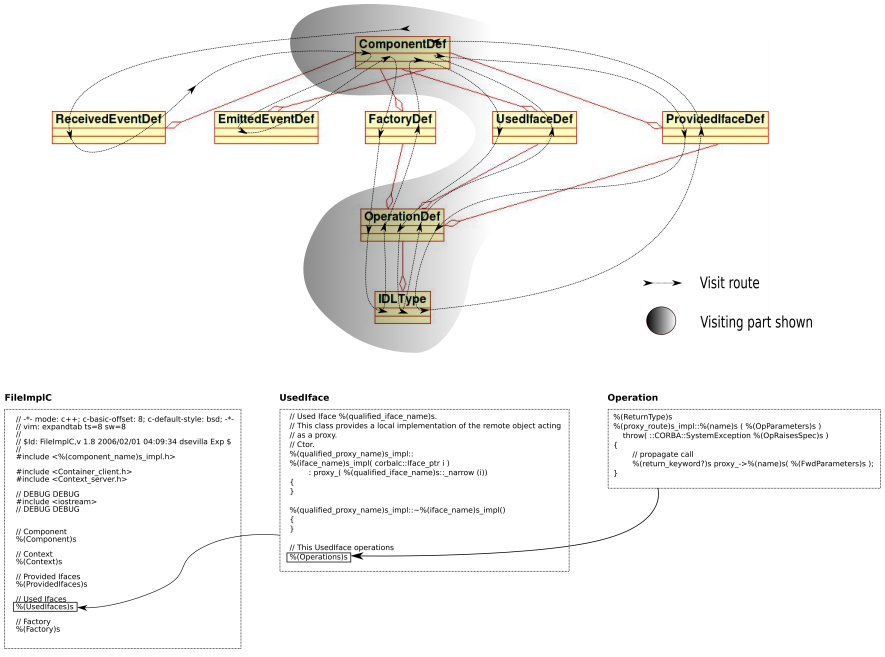

Fig. 2. Visiting path and Text Template construction.

3.1 Structure of the Code Generator

The CORBA–$\mathcal{LC}$ Code Generator uses the information of both (1) the standard CORBA *Interface Repository* (IR), and (2) the Component's XML file. While the XML describes the component ports and non-functional requirements, the IR describes all the IDL interfaces used in those ports. As output, the code generator produces the needed implementation files and boilerplate that can be used by the programmer to write the functionality proper of the component.

Figure 2 shows a part of the composition hierarchy of the IR entities that take part on the Code Generation. The dashed line shows the visiting order in which those entities are visited. Each visitor acts on one of the entities, extracting the needed information to generate the code. This information is used to build *Text Templates* as shown in the lower part of the figure. These templates act in turn as part of upper level templates to finally build the complete generated file. Using templates makes the code generator extremely flexible, meaning that the generated code can be changed without even recompiling the code generator.

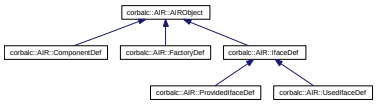

Fig. 3. Augmented IR hierarchy.

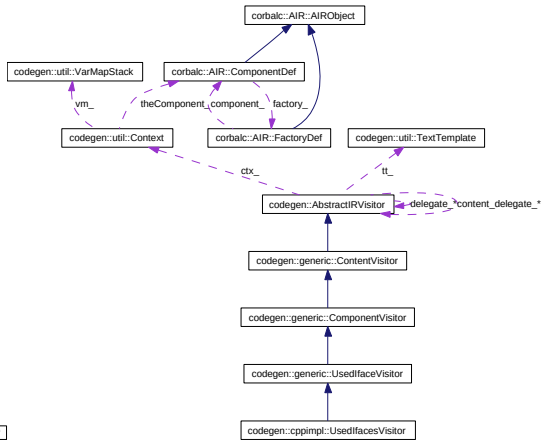

Fig. 4. Visitor inheritance and structure.

The visiting path includes all the characteristics of the component. As the standard CORBA IR does not include description of the ports and factory of a component, a set of augmented IR classes has been added to the framework. These classes inherit from the abstract base `corbalc::AIR::AIRObject`, and hold information about the different ports and the factory of the component (Figure 3).

In order to make the code generator as modular and reusable as possible, the Code Generation Framework is divided into several hierarchical layers:

- **Abstract IR Visitor** – As shown in Figure 4, this is the base for all the visitor classes in the framework. Each visitor has a set of delegates and content delegates (seen below), and is in charge of building a text template. All visitors share a ***context***, that includes a reference to the component being generated.
- **Generic IR Visitors** – Generic visitors store the functionality to visit each of the IR entities in an generic way.
- **File-specific IR Visitors** – Specific visitors inherit from the generic ones, and implement the specific functionality needed for the concrete file being generated.

In parallel to this inheritance relationships among the different visitors, they can also be composed or *connected*. When a visitor is approached to generate its template, the code inherited from the abstract visitor traverses also all the delegate visitors attached to this visitor. This makes the design of the code generator completely modular and flexible. Figure 5 shows the set of visitors that participate in the generation of one of the output files (the C++ implementation file for the component). Arrows from a visitor indicate a delegate. The complete delegate structure forms a tree. Note that each interface (either used or provided interface) has a delegate that visits all the operations, and this one in turn a delegate that visits all the parameters of each operation.

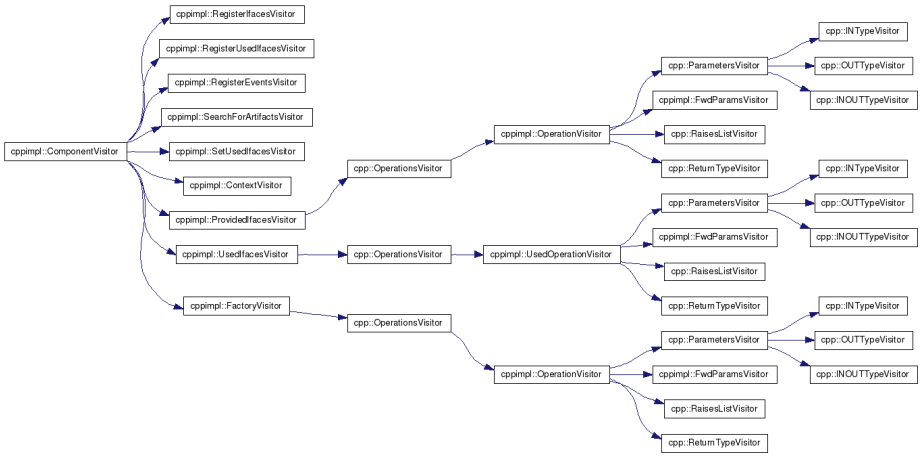

Fig. 5. Delegation Structure.

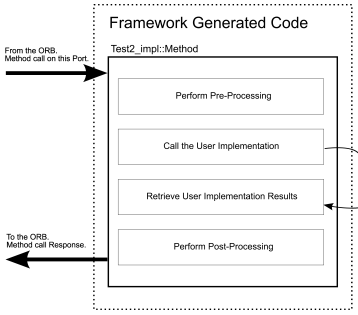

Fig. 6. Executor call sequence.

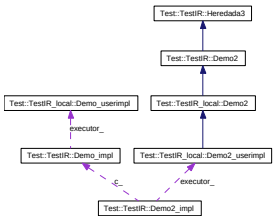

Fig. 7. Executor Structure for the `Test::TestIR::Demo2` provided port.

3.2 The Executor code

Ports are defined in terms of CORBA interfaces. In order for the framework to control the flow of calls between the user-programmed code of the component and framework-generated code, the framework generates all the CORBA objects and servants that realize the communication. Figure 7 shows the structure of an interface (`Test::TestIR::Demo2`) that acts as a provided port for a component (defined as the CORBA interface `Test::TestIR::Demo`).

The programmer code for component ports is called ***executor***. In the figure, the executor corresponds with the class `TestIR_local::Demo2_userimpl`. The framework implementation (`TestIR::Demo2_impl`) has a member called `executor_` that holds a reference to that programmer class.

Figure 6 shows the flow of a given method call. The ORB redirects the call to the CORBA implementation object (generated by the code generator) that does some pre-processing, then calls the actual method code (written by the component programmer), and then does some post-processing ("*point-cuts*" in Aspect-Oriented Programming (AOP)[5] terminology) and returns the call. This flow is very important, as it ensures all incoming and outgoing calls of this component are controlled by the framework, allowing it to control the non-functional aspects.

4 Assuring Fault Tolerance

While there are a lot of non-functional aspects to consider, this paper focuses on an example of fault-tolerance. As all the incoming and outgoing calls into component code are controlled by the framework, it can arrange several instances of the used components to provide the needed fault tolerance for a component that requested it. Suppose that the same component shown above that offered the `Demo2` interface as a port, uses the `Demo4` interface as an used port from another component. This is the code generated for the `op41` operation:

```
::CORBA::Long
Test::TestIR_proxy::Demo4_impl::op41 ( ::CORBA::Long x )
    throw( ::CORBA::SystemException  )
{
        ::CORBA::Long retval_;
        retval_ = proxy_->op41( x );
        return retval_;
}
```

The ***proxy*** object encompasses the pre- and post-processing needed to ensure fault-tolerance. In particular, the proxy code:

1. Create n threads to perform n "`op41`" calls to n real components distributed through the network.[3]
2. Each thread issues one call to its component. Some of the calls may fail or take too long. This information can be used to ignore failing objects.
3. When some or all responses are received, the proxy can implement several approaches, such as voting, taking the first response or distributing the calls to perform a basic load balancing as well.

Thus, automatic code generation through the CORBA–$\mathcal{LC}$ Code Generator jointly with this interceptor (point-cut) mechanism save the programmer of the burden of including these characteristics in its components, allowing him to concentrate in the component functionality proper.

5 Related Work

To date, several component models have been developed. Although CORBA–$\mathcal{LC}$ shares some features with them, it also has some key differences.

Java Beans[6], Microsoft's Component Object Model (COM)[7], .NET[8] offer similar component models, but lack in some cases that are either limited to the local case or do not support heterogeneous environments of mixed operating systems and programming languages as CORBA does.

In the server side, SUN's EJB[9] and the new Object Management Group's CORBA Component Model (CCM)[10] offer a server programming framework in which server components can be installed, instantiated and run. Both are fairly similar. In fact, CCM "*basic*" level makes both models totally compatible. EJB is a Java-only system, while CCM continues the CORBA heterogeneous philosophy. Both are designed to support enterprise applications, offering a container architecture with support for transactions, persistence, security, etc. They

[3] Note that the threads could have been created before, and that the set of n components is selected automatically by the framework among all the nodes of the network.

also offer the notion of components as binary units which can be installed and executed (following a fixed assembly) in Components Servers.

Although CORBA–$\mathcal{LC}$ shares many features with both models, it presents a more dynamic model in which the deployment is not fixed and is performed at run-time using the dynamic system data offered by the Reflection Architecture. Also, CORBA–$\mathcal{LC}$ is a *lightweight* model in which the main goal is the optimal network resource utilization instead of being oriented to enterprise applications.

6 Conclusions, Status and Future Work

As we showed in this paper, component technology in general, and CORBA–$\mathcal{LC}$ in particular, offer a new and interesting way of approaching distributed applications. Services otherwise complicated can be offered by the framework just by specifying them in the characteristics and needs of components and applications.

The paper also showed the flexible structure of the CORBA–$\mathcal{LC}$ Code Generator, that allows changing the way the code is generated without recompiling, and that is modular enough to add new capabilities by just adding and connecting new specialized visitors. By now only C++ is supported, but by means of the flexible design, other languages also supported by CORBA, such as Java, could be used.

Finally, we showed how convenient the Aspect-Oriented approach is to seamlessly and transparently offer services such as fault tolerance, replication and load balancing to components.

References

1. C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. ACM Press, 1998.
2. D. Sevilla, J. M. García, and A. Gómez. CORBA Lightweight Components: A Model for Distributed Component-Based Heterogeneous Computation. In *EUROPAR'2001*, pages 845–854, Manchester, UK, August 2001. LNCS 2150.
3. M. Henning and S. Vinoski. *Advanced CORBA Programming with C++*. Addison-Wesley Longman, 1999.
4. Object Management Group. *CORBA: Common Object Request Broker Architecture Specification, revision 3.0.2*, 2002. OMG Document formal/02-12-06.
5. F. Duclos, J. Estublier, and P. Morat. Describing and Using Non Functional Aspects in Component Based Applications. In *International Conference on Aspect-Oriented Software Development, Enschede, The Netherlands*, April 2002.
6. SUN Microsystems. *Java Beans specification*, 1.0.1 edition, July 1997. http://java.sun.com/beans.
7. Microsoft. *Component Object Model (COM)*, 1995. http://www.microsoft.com/com.
8. Microsoft Corporation. Microsoft .NET. http://www.microsoft.com/net/.
9. SUN Microsystems. *Enterprise Java Beans specification*, 1.1 edition, December 1999. http://java.sun.com/products/ejb/index.html.
10. Object Management Group. *CORBA Component Model*, 1999. OMG Document ptc/99-10-04.