

OpenCorba: a Reflective Open Broker

Thomas Ledoux

École des Mines de Nantes
4 rue Alfred Kastler
F-44307 Nantes cedex 3, France
Thomas.Ledoux@emn.fr

Abstract. Today, CORBA architecture brings the major industrial solution for achieving the interoperability between distributed software components in heterogeneous environments. While the CORBA project attempts to federate distributed mechanisms within a unique architecture, its internal model is not very flexible and seems not to be suitable for future evolutions. In this paper, we present OpenCorba, a *reflective open broker*, enabling users to adapt dynamically the representation and the execution policies of the software bus.

We first expose the reflective foundations underlying the implementation of OpenCorba: i) metaclasses which provide a better separation of concerns in order to improve the class reuse; ii) a protocol which enables the dynamic changing of metaclass in order to allow run-time adaptation of systems.

Based on this reflective environment, OpenCorba enables the adaptability of the internal characteristics of the broker in order to change its run-time behavior (e.g. remote invocation, IDL type checking, IR error handling). OpenCorba gives a clear example of the benefits of reflective middleware.

1 Introduction

In the last couple of years, the success of the Internet emphasizes the need to quickly find solutions for interoperability between distributed heterogeneous environments. Reusability and composability of software components are one of the main concerns for the computer industry dealing with different languages, systems and locations.

By supporting specifications for portable and interoperable object components, the OMG (Object Management Group) consortium proposes a solution to deal with the construction of object-oriented distributed applications [OMG 95]. Indeed, OMG's standardization efforts led to the definition of a whole modular architecture approved by the computer industry. The specifications describe independent modules of a large system, from technical aspects to business objects, reviewing essential distributed services such as security, transactions, event notification, etc. known as the CORBA Services [OMG 97]. The CORBA (Common Object Request Broker Architecture) software bus is the main module of this architecture and has the responsibility of

achieving a transparent communication between remote objects [OMG 98]. The *modularity* of this architecture is a major advantage of the OMG solution.

However, the complexity of the broker specifications negatively impacts in the intended flexibility of the CORBA model. For example, the invocation mechanism is a "black box" described by fixed specifications. The introduction of a small evolution leads to a new version of the specifications, making obsolete the last one. Dealing with the invocation, the introduction of the interceptors mechanism in CORBA 2.2 and the request for proposal Messaging Service for CORBA 3.0, attempt to improve the existing specifications (and resulting applications). Thus, we can be sceptical about the stability of such improvement. In the current style of distributed systems, we must support the *dynamic* modification of the broker mechanisms to deal with changing contexts of execution (e.g. load balancing, fault tolerance). This dynamic capability of the bus makes possible the evolution of execution policies (e.g. object migration).

We propose an overall solution that allows the object bus to be *adaptive*, making the broker "plug and play". The classification of concurrent and distributed programming proposed by Briot et al. [BRI 98] constitutes an interesting framework for tackling the adaptive issue. The authors distinguish three approaches:

- *library* approach applies object-oriented concepts in order to structure the concurrent and distributed systems through class libraries;
- *integrative* approach consists in unifying concurrent and distributed systems concepts with object-oriented ones;
- *reflective* approach integrates protocol libraries dealing with concurrency and distribution within an object-based programming system.

Using this classification, we can notice that the OMG model both corresponds to an integrative and a library approach, with the minimal object model and the CORBA services perspective respectively. In this taxonomy, the reflective approach is presented as a solution for combining the advantages of the two previous approaches. So, reflection is the best choice for handling the adaptability of the object bus. Reflection [SMI 82] [MAE 87] [KIC 91] allows the extension of the initial OMG model with libraries of meta-protocols customizing mechanisms of distributed programming. Then, it is possible to introduce – in a transparent way – new semantics on the initial model such as concurrency, replication, security, etc. including ones currently unthought of.

In this paper, we present an overview of OpenCorba [LED 98]: a CORBA broker based on a reflective approach. Its architecture enables the reification of the internal characteristics of the software bus in order to modify and adapt them at *run-time*. Then, OpenCorba allows introspection and dynamic modification of the representation and the execution policies of the CORBA bus. Its implementation is based on the reflective language NeoClasstalk [RIV 97]. This language results from an implementation of a MOP (Meta Object Protocol) [KIC 91] in Smalltalk [GOL 89]. Its main contribution consists of an extension of dynamic aspects in Smalltalk: an efficient solution for handling message sending and a way of achieving dynamic behavior of a class.

This paper is presented as follows. In section 2, we explain the reflective foundations underlying the making of OpenCorba and show the advantages of reflection for building open systems. In section 3, after an introduction of OpenCorba itself, we present three possible reflective aspects of the bus. In section 4, we present related work, and in section 5, we discuss about our works in progress. Finally, we draw our conclusions on the contribution of *dynamic adaptability* to middleware.

2 Reflective Foundations for Building Open Systems

In this section, we show the benefits of the paradigm of metaclasses for building reusable and adaptable architectures.

2.1 Metaclasses and Separation of Concerns

Reflection allows us to separate what an object does (the base level) from how it does it (its meta level) [McA 95]. A reflective language encourages a clean separation between the basic functionalities of the application from its representations and controls. In class-based languages integrating reflective features [COI 87] [DAN 94], the class of a class – a *metaclass* – defines some properties concerning object creation, encapsulation, inheritance rules, message handling, etc. We call *class properties* the properties that denote behavior for classes themselves, independently from the behavior for their instances. In [LED 96], we present a taxonomy of reusable metaclasses which represent class properties such as ensuring that a class has one sole instance (like the pattern Singleton [GAM 95]), a class cannot be subclassed (like Java final [GOS 96]), a class provides pre/post conditions for its methods (like the Eiffel assertions [MEY 92]), etc.

The previous taxonomy was implemented in the MOP NeoClasstalk, which is a based on a new kernel of metaclasses inside the Smalltalk world [RIV 96]. An additional metaclass named `StandardClass`, is defined and strapped into the initial Smalltalk meta-level architecture. `StandardClass` provides a starting point to the NeoClasstalk system. Then, new metaclasses can be derived from `StandardClass`, which describes common behavior for classes, by subclassing it.

Fig. 1 shows a class `Account`, instance of the metaclass `BreakPoint` that owns the responsibility to set breakpoints in the methods of the class `Account`¹. Message sending is handled by the method `#execute:receiver:arguments:` of the MOP NeoClasstalk: the metaclass `BreakPoint` intercepts messages received by the instances of `Account`.

¹ It could be interesting to control class interactions with the rest of the system during the debugging phase.

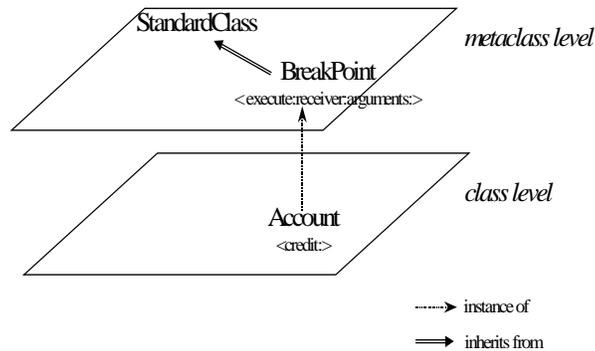


Fig. 1. Class properties and (meta)classes

The method `#execute:receiver:arguments:` described above, opens a debugger with the execution context of the trapped method (e.g. `#credit:`), then performs the originally intended method via traditional inheritance mechanisms.

metaclass code

```

BreakPoint>>execute: cm receiver: rec arguments: args
  "Set a breakpoint on my instance methods"
  self halt: 'BreakPoint for ', cm selector.
  ^super execute: cm receiver: rec arguments: args
  
```

class code

```

Account>>credit: aFloat
  "Make a credit on the current balance"
  self balance: self balance + aFloat
  
```

In this way, we avoid the mixing between the business code (bank account) and the code describing a specific property on it (breakpoints). By increasing the separation of concerns, class properties encourage readability, reusability and quality of code. Then, reusable metaclasses propose a better organisation of class libraries for designing open architectures.

2.2 Dynamic Change of Metaclass

The dynamic change of class introduced by NeoClasstalk is a protocol that makes it possible for the objects to change their class at run-time [RIV 97]². The purpose of

² This protocol compensates for the restrictions imposed by the method `#changeClassToThatOf:` found in Smalltalk.

this protocol is to take into account the evolution of the behavior of the objects during their life in order to improve their class implementations. The association of first class objects with this protocol allows the dynamic change of metaclass at run-time. Therefore, it is possible to dynamically add and remove class properties without having to regenerate code.

By taking again the previous example, we can temporarily associate the `BreakPoint` property with a class during its development. The class `Account` changes its original metaclass towards metaclass `BreakPoint`³ for debugging the messages, then returns towards its former state reversing its change of class (cf. **Fig. 2**).

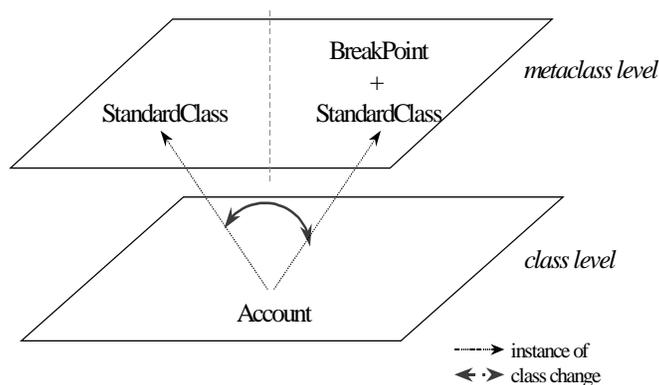


Fig. 2. Dynamic adaptability of class properties

The protocol for dynamically changing the class of a class (the metaclass) allows us to replace a class property by another, during execution. In our work dealing with adaptive brokers, this protocol is extremely helpful because it provides a "plug and play" environment for enabling the *run-time* modification of the distributed mechanisms.

3 OpenCorba

Our first implementation of an open architecture dealt with the CORBA platform [OMG 98] and gave place to the implementation of a software bus named OpenCorba. The goal of this section is to expose the major reflective aspects of OpenCorba. Others features of OpenCorba like the IDL compiler or the different layers of the broker are described in details in [LED 98].

³ Or a composition of metaclasses dealing with `BreakPoint` and other class properties (cf. 5.1).

3.1 Introduction

OpenCorba is an application implementing the API CORBA in NeoClasstalk. It reifies various properties of the broker – by the means of explicit metaclasses – in order to support the separation of the internal characteristics of the ORB. The use of the dynamic change of metaclass allows to modify the ORB mechanisms represented by metaclasses. Then, OpenCorba is a reflective ORB, which allows to *adapt* the behavior of the broker at *run-time*.

In the following paragraphs, we present three aspects of the bus that have been reified:

1. the mechanism of remote invocation via a proxy;
2. the IDL type checking on the server class;
3. the management of exceptions during the creation of the interface repository⁴.

First, we introduce the two basic concepts for creating classes in OpenCorba. This creation deals directly with the reflective aspects.

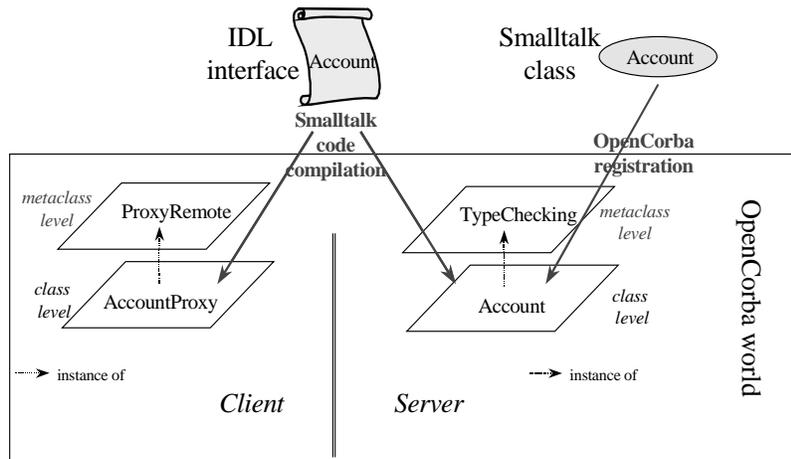


Fig. 3. Creation of OpenCorba classes

IDL Mapping in OpenCorba.

Following the Smalltalk mapping described in the CORBA specifications [OMG 98], the OpenCorba IDL compiler generates a proxy class on the client side and a template class on the server side. The proxy class is associated with the metaclass

⁴ Let us recall that the interface repository (IR) is similar to a run-time database containing all the IDL specifications reified as objects.

`ProxyRemote` implementing the remote invocation mechanisms; the template class, with the metaclass `TypeChecking` which implements the IDL type checking on the server. The left hand of **Fig. 3** shows the results from IDL mapping of the `Account` interface in OpenCorba: the proxy class `AccountProxy` is instance of `ProxyRemote` and the template class `Account` is instance of `TypeChecking`.

Feature Smalltalk2OpenCorba.

OpenCorba allows the Smalltalk developers to transform any Smalltalk standard class into a Smalltalk server class in the ORB. This feature was introduced to reuse existing Smalltalk code, and to free the programmer from the writing of the bulk of IDL specification. Technically, a semi-automatic process is applied to the Smalltalk class in order to generate the interface repository and the IDL file. Then, to become a server class in OpenCorba, the class must be an instance of `TypeChecking` (cf. **Fig. 3**, right hand). By analogy with similar techniques, we called this special feature `Smalltalk2OpenCorba`.

3.2 The OpenCorba Proxy

In distributed architectures, the proxy object is a local representation in the client side of the server object. Its purpose is to ensure the creation of the requests and their routing towards the server, then to turn over the result to the client. In order to remain transparent, a proxy class adapts the style of local call to the mechanism of remote invocation [SHA 86].

Separation of Concerns.

The remote invocation mechanism is completely independent of the semantics of the IDL interface (e.g. `Account`). That is, remote invocation refers to the control of the application and not to the application functionality; it deals with meta-level programming. The OpenCorba IDL compiler automatically generates the proxy class on the basic level of the application and associates the proxy class with the metaclass `ProxyRemote` in charge of calling the real object. The remote invocation remains thus transparent for the client. We can manipulate the proxy class with the traditional Smalltalk tools (browser, inspector).

Base level features. Methods of the proxy class are purely descriptive and represent interfaces like IDL operations [LED 97]. In the Smalltalk browser, the methods do not contain any code, only one comment denoting that OpenCorba generated them. **Table 1** presents some examples of mapping for OpenCorba. The type of the returned values is put in comments to announce IDL information to the programmer.

Table 1. Examples of IDL mapping

<i>IDL attribute/operation</i>	<i>Methods of a proxy class OpenCorba</i>
readonly attribute float balance;	balance "Generated by OpenCORBA" *** DO NOT EDIT *** "aFloat "
void credit(in float amount);	credit: aFloat "Generated by OpenCORBA" *** DO NOT EDIT *** "nil"

Meta level features. By definition, the sending of a message to a proxy object involves a remote message send to the server object it represents. The idea is to intercept the message at its reception time by the proxy object and to launch a remote invocation. Thus, the control of the message sending is suitable for our purpose. The metaclass `ProxyRemote` redefines the method `#execute:receiver:arguments:` of the MOP `NeoClasstalk` to intercept the messages received by a proxy. This redefinition carries out the remote invocation by using the DII CORBA API according to [OMG 98].

Dynamic Adaptability.

To allow the dynamic adaptability of the invocation mechanisms in `OpenCorba`, it is possible to develop others metaclasses. We distinguished them in two categories:

- The first one deals with the possible variations on the proxy mechanisms. We think of policies modelling Java RMI [SUN 98], a future version of the CORBA DII or a local invocation. This last mechanism was implemented in `OpenCorba` in order to model proxies with cache.
- The second category considers extensions of the proxy concept for introducing new mechanisms like object migration [JUL 88] [OKA 94] or active replication [BIR 91] [GUE 97].
 - Object migration consists in transferring a server object towards the client side in order to optimize the performances of the distributed system. This mechanism reduces the bottlenecks of the network and minimizes the remote communications.
 - Replication is another mechanism of administration of the objects distribution. It consists of a duplication of the server in several replicas, which are the exact representation of the original server object. The mechanism of active replication supposes that the message is sent by the client to the replicas – via the proxy – thanks to an atomic protocol of diffusion (broadcast).

Thus, the dynamic adaptability of metaclasses allows to implement some variations on the remote invocation mechanism, without upsetting existing architecture.

3.3 IDL Type Checking on the Server Classes

The CORBA standard [OMG 98] specifies that the server side uses the interface repository to check the conformity of the signature of a request by checking the argument and return objects. On the other hand, it does not specify how it must be carried out.

Separation of Concerns.

We are convinced that it falls within the competence of the server class to check if one of its methods can be applied or not. Indeed, if this kind of checking could be carried out more upstream by the ORB (during the unmarshalling of request by the server for example), that would introduce a more inflexible management: a little change in the mechanism of control involves a rewriting of the lower layers of the broker. On the contrary, we propose an externalisation of the control of these layers towards the server class, which will test if the application of a method is possible or not.

Moreover, the type checking is independent of the functionalities defined by the server class: it can be separated from the base code and constituted as a class property that will be implemented by a metaclass. Thus, the code of the server class does not carry out any test on the type of the data in arguments. The developer can then write, modify or recover its code without worrying about the control of the type handled by the metaclass `TypeChecking`.

Technically, this metaclass controls the message sending on the server class – via the method `#execute:receiver:arguments:` – to interrogate the interface repository before and after application of its methods. This query enables us to check the type of the arguments and the type of the result of the method.

In conclusion, OpenCorba externalises the type checking, at the same time from the lower layers of the ORB, and from the server class.

Dynamic Adaptability.

Thanks to our approach, we can bring new mechanisms for type checking, without modifying the existing implementation. For example, we can develop a new metaclass managing a system of cache for the types of the parameters. During the first query of the interface repository, OpenCorba locally backs up the type of each argument of the method⁵. Then, with the next invocations of the method, the metaclass questions memorized information in order to carry out the type checking.

Another example: we can also remove the control of the type for reasons of performance or when the type of the parameters is known before hand. A dynamic

⁵ For example, in a Smalltalk shared dictionary or in the byte code of the compiled method.

change of metaclass allows then to associate the server class with the default metaclass of the OpenCorba system (i.e. `StandardClass`).

3.4 Interface Repository and Error Handling

There are two ways used to populate the interface repository: the IDL mapping and the feature `Smalltalk2OpenCorba`. In the first case, the generation of each proxy class is handled in the creation of the objects in the repository. In the second case, this creation is allowed by a table of Smalltalk equivalence towards IDL (*retro-mapping*). Technically, these two mechanisms do not have the same degree of intercession:

- In the case of `Smalltalk2OpenCorba`, the Smalltalk compiler already carried out the syntactic analysis and the semantic checking of the class. Also, the installation of a Smalltalk class in the ORB does not cause errors during the creation of the objects in the repository;
- On the other hand, in the IDL case, it is an authentic compilation where the semantic actions check the integrity of IDL specifications before the creation of objects in the repository (e.g. the same attribute duplicated).

Thus, in order to distinguish the Smalltalk case from the IDL case where creation can lead to error handling, we must encapsulate the CORBA creational APIs of the interface repository. Let us recall that these creational APIs are implemented by the *container* classes of the interface repository specified by CORBA standard [OMG 98].

Separation of Concerns.

By analyzing the tests of integrity necessary for the IDL compilation, it appears that they are generic and independent from the creational APIs of the *containers*. Thus, they can be externalized from the *container* classes to constitute a class property which will be implemented by a metaclass. The code of the *container* classes does not carry out any tests. It is then helpful to differentiate the IDL case from the Smalltalk case, in order to associate a given metaclass or not to the *container* classes.

The metaclass `IRChecking` plays this role. It specializes the method `#execute:receiver:arguments:` of the MOP `NeoClasstalk` in order to intercept the messages received by the instances of the *container* class. For the creational APIs, the integrity tests are carried out and an exception is raised if an error occurs.

Dynamic Adaptability.

Our design leads to a greater flexibility to carry out or transform the integrity tests without having to modify the creational APIs. Thus, it allows the distinction between the Smalltalk case and the IDL case at run-time. In the Smalltalk case, creation is carried out normally (default metaclass `StandardClass`); in the IDL case, there is a dynamic adaptability of the behavior to carry out creation only if the integrity tests do

not raise an error (metaclass `IRChecking`). OpenCorba connects the appropriate metaclasses at run-time according to needs of the system.

3.5 Implementation and Performance Issues

Reflective aspects of OpenCorba are essentially based in the ability of handling message sends. Therefore, the extra cost of the reflective broker is highly dependent on the performance of the message sending.

The implementation of this control in NeoClasstalk is based on a technique called *method wrappers* [BRA 98]. The main idea of this feature is the following: rather than changing the method lookup process directly at run-time, we modify the compiled method objects that the lookup process returns. In NeoClasstalk, the method wrappers deal both with compile-time and run-time reflection. We briefly describe the two steps involved:

- *At compile-time*
The original method defined in a class is wrapped so that it sends to the class itself the message `#execute:receiver:arguments:` defined in its class (the metaclass). The arguments are i) the original compiled method which is stored in the method wrapper, ii) the object originally receiving the message, iii) the arguments of the message.
- *At run-time*
Method lookup is unmodified: the activation of the method wrapper does the call to the metaclass and starts the meta-level processing.

By analysing these two steps, we conclude that the cost is actually significant. First, the meta-level indirection occurring at run-time involves additional method invocations. Since the meta-level indirection is applied to each message send, the performance of the whole system decreases. However, we can notice that since OpenCorba deals with distributed environment, the network traffic moderates the impact of this overhead.

Secondly, at compile-time, there is a real difficulty in applying the method wrappers concept: which are the methods that we need to wrap? It depends on the problem at hand. For example, in OpenCorba, all the methods of a server class and all the inherited ones must be wrapped. Thus, the technique of method wrappers imposes some decisions at design time, which imply a cost that should not be overlooked.

In summary, the flexibility provided by reflection impacts on the system efficiency. Many works attempt to find a way for efficient reflective systems such as the optimization of virtual machine for meta-programming, the reification categories which provide the opportunity to specifically reify a given class [GOW 96], the partial evaluation for MOP [MAS 98], etc.

4 Related Work

4.1 Reflective Adaptive Middleware

Like the OpenCorba broker, other research projects are under development in the field of adaptive middleware. The ADAPT project of the University of Lancaster has investigated the middleware implementation for mobile multimedia applications which are capable of dynamically adapting to QoS fluctuations [BLA 97]. Its successor, the OpenORB project, studies the role of reflection in the design of middleware platforms [BLA 98]. The implementation of the current reflective architecture is based on a per-object meta-space (structured as three distinct meta-space models) and on the concept of open bindings, differing from the per-class reflective environment and the MOP of OpenCorba. These two design approaches result in two different ways of investigating reflective middleware.

The FlexiNet platform [HAY 98], a Java middleware, proposes to reify the layers of the communication stack into different meta-objects (in order to customize them). Each meta-object represents a specific aspect such as call policy, serialization, network session, etc. OpenCorba currently reifies characteristics dealing the upper layers of the invocation mechanism, and could reify the lower layers (e.g. marshalling, transport).

Researchers at the University of Illinois developed dynamicTAO, a CORBA-compliant reflective ORB that supports run-time reconfiguration [ROM 99]. Specific strategies are implemented and packed as dynamically loadable libraries, so they can be linked to the ORB process at run-time. Rather than implementing a new ORB from scratch as done in OpenCorba, they chose to use TAO [SCH 99], causing a dependency to this ORB.

Finally, since its experiment in the development of MOP [GOW 96], the distributed system team of the Trinity College of Dublin has recently started the Coyote project whose goal is to provide the adaptation of distributed system (e.g. administration of telecommunication network, CORBA bus).

4.2 Reflection in Distributed Systems

The use of reflection in the concurrent and distributed systems is certainly not recent [WAT 88], but seems to take a new rise in the last years. Indeed, many research projects in the field of reflection or in the distributed domain use the reification of the distributed mechanisms to modify them and specialize them. Let us quote the mechanisms of migration [OKA 94], marshalling [McA 95], replication [GOL 97], security [FAB 97], etc. The CORBA architecture is a federate platform of the various mechanisms of distribution. Also, it seems interesting to study these projects to implement their mechanisms within OpenCorba.

4.3 Programming with Aspects

The mixing in the system's basic functionality of several technical aspects (e.g. distribution, synchronization, memory management) dealing with the application domain, constitutes one of the major obstacles to the reusability of the software components. A possible solution is then to consider the isolation of these specific aspects for their individual reuse. The "tangled" code is separated and aspects can evolve independently thus formulating the paradigm of separation of concerns [HUR 95].

There are some models and techniques allowing the separation of concerns: composition filters [BER 94], adaptive programming [LIE 96], aspect-oriented programming (AOP) [KIC 97]. However, the latter implement particular constructs to achieve the programming with aspects. We preferred a reflective approach that does not impose a new model, but extends the existing languages to open them. Moreover, contrary to these models, our solution allows the dynamic adaptability of aspects.

5 Work in Progress

5.1 Metaclass Composition

Distributed architectures are complex and require many mechanisms for their implementation. The question of how to compose these mechanisms is essential. For example, the functionality of "logging within a class the remote invocations" uses the mechanism of "logging" combined with the mechanism of "remote invocation". As we have seen previously, each of these mechanisms corresponds with a class property and is implemented by a metaclass. Thus, the combination of several mechanisms raises the problem of the composition of the metaclasses: this composition causes conflicts a priori when there is a behavior overlap.

To tackle this problem, our recent work consists in the definition of a "metaclass compatibility model" in order to offer a reliable framework for the composition of the metaclasses [BS 98]. We attempt to define a classification of the various distribution mechanisms to prevent possible overlappings of behavior.

5.2 Towards Specifications of a Reflective Middleware

By studying concurrent and distributed architectures, we can notice that they deal with well-known mechanisms and policies, which are independent from the system's basic functionality (i.e. business objects) and could be implemented at the meta-level. Then, we can suggest a first classification of middleware features related to reflective components:

- Distribution
proxy mechanism, replication, migration, persistence, etc.

- **Communication**
synchronous, asynchronous, no reply, future, multicast, etc.
- **Object concurrency**
inter-objects, intra-object (readers/writer model, monitor), etc.
- **Reliability**
exceptions, transactions, etc.
- **Security**
authentication, authorisation, licence, etc.
- **Thread management**
single-threaded, thread-per-message, thread-pool, etc.
- **Data transport**
sockets, pipes, shared memory, etc.

The richness of future middlewares will depend on their capabilities to dynamically adapt and (fine) tune such features. Then, we are strongly convinced that the first step to build such a middleware is to find a good reflective model and environment.

In OpenCorba, we experimented with a per-class reflective environment on the top of Smalltalk. We are currently investigating other models like meta-object or message reification models in order to compare them with our metaclass approach. We noticed that the design of reflective distributed mechanisms could be common to any reflective language dealing with similar features. For example, the handling of the message sending is an important feature because several mechanisms have to deal with it (in their implementation). Then, most of the reflective languages support the *same design* of the meta-level in order to implement a given distributed mechanism (e.g. primary replication [CHI 93] [McA 95] [GOL 97]).

To reason about a language independent environment, we sketched out an abstract MOP allowing the control and customization of the message sending and state accessing. Thus, we plan to describe the specifications of distributed mechanisms in the context of meta-programming, i.e. reflective distributed components.

6 Conclusion

In this paper, we developed the idea in which reflection is a tremendous vector for the building of open architectures. Languages considering classes as full entities allow separation of concerns through metaclasses, improving reusability and quality of code. Then, associated to the dynamic change of class, metaclasses offer an exclusive reflective scheme for the building of reusable and adaptable, open architectures.

In the context of distributed architectures, these reflective foundations offer a dynamic environment allowing the reuse and the adaptability of mechanisms related to the distribution. Our first experimentation enabled us to "open" some internal characteristics of the CORBA software bus, such as the invocation mechanism. The

result – OpenCorba – is an open broker capable of dynamically adapting the policies of representation and execution within the CORBA middleware.

Acknowledgements

The research described in this paper was funded by IBM Global Services (France) during the author's PhD (1994-1997). The author wishes to thanks Noury Bouraqadi-Saâdani, Pierre Cointe, Fred Rivard for many discussions about OpenCorba, Xavier Alvarez for his reviewing of the final version of the paper.

References

- [BER 94] BERGMANS L. — *Composing Concurrent Objects*. PhD thesis, University of Twente, Enschede, Netherlands, June 1994.
- [BIR 91] BIRMAN K., SCHIPER A., STEPHENSON P. — Lightweight Causal and Atomic Group Multicast. In *ACM Transactions on Computer Systems*, vol.9, n°3, p.272-314, 1991.
- [BLA 97] BLAIR G.S., COULSON G., DAVIES N., ROBIN P., FITZPATRICK T. — Adaptive Middleware for Mobile Multimedia Applications. In *Proceedings of the 8th International Workshop on NOSSDAV'97*, St-Louis, Missouri, May 1997.
- [BLA 98] BLAIR G.S., COULSON G., ROBIN P., PPATHOMAS M. — An Architecture for Next Generation Middleware. In *Proceedings of Middleware'98*, Springer-Verlag, p.191-206, N. Davies, K. Raymond, J. Seitz Eds, September 1998.
- [BRA 98] BRANT J., FOOTE B., JOHNSON R., ROBERTS D. — Wrappers to the Rescue. In *Proceedings of ECOOP'98*, Springer-Verlag, Brussels, Belgium, July 1998.
- [BRI 98] BRIOT J.P., GUERRAOUI R., LÖHR K.P. — Concurrency and Distribution in Object-oriented Programming. In *ACM Computer Surveys*, vol.30, n°3, p.291-329, September 1998.
- [BS 98] BOURAQADI-SAÂDANI N., LEDOUX T., RIVARD F. — Safe Metaclass Programming. In *Proceedings of OOPSLA'98*, ACM Sigplan Notices, Vancouver, Canada, October 1998.
- [CHI 93] CHIBA S., MASUDA T. — Designing an Extensible Distributed Language with a Meta-Level Architecture. In *Proceedings of ECOOP'93*, p.482-501, LNCS 707, Springer-Verlag, Kaiserslautern, Germany, 1993.
- [COI 87] COINTE P. — Metaclasses are First Class : the ObjVlisp Model. In *Proceedings of OOPSLA'87*, ACM Sigplan Notices, p.156-167, Orlando, Florida, October 1987.

- [DAN 94] DANFORTH S., FORMAN I.R. — Reflections on Metaclass Programming in SOM. In *Proceedings of OOPSLA'94*, ACM Sigplan Notices, Portland, Oregon, October 1994.
- [FAB 97] FABRE J.C, PÉRENNOU T. — A metaobject architecture for fault tolerant distributed systems: the FRIENDS approach. In *IEEE Transactions on Computers*. Special Issue on Dependability of Computing Systems, vol.47, n°1, p.78-95, January 1998
- [GAM 95] GAMMA E., HELM R., JOHNSON R., VLISSIDES John — *Design Patterns*, Addison-Wesley Reading, Massachusetts, 1995.
- [GOL 89] GOLDBERG A., ROBSON D. — *Smalltalk-80 : The Language*, Addison-Wesley, Reading, Massachusetts, 1989.
- [GOL 97] GOLM M. — *Design and Implementation of a Meta Architecture for Java*. Master's Thesis, University of Erlangen, Germany, January 1997.
- [GOS 96] GOSLING J., JOY B., STEELE G. — *The Java Language Specification*. The Java Series, Addison-Wesley, Reading, Massachusetts, 1996.
- [GOW 96] GOWING B., CAHILL V. — Meta-Object Protocols for C++ : The Iguana Approach. In *Proceedings of Reflection'96*, Ed. Kiczales, San Francisco, California, April 1996.
- [GUE 97] GUERRAOUI R., SCHIPER A. — Software Based Replication for Fault Tolerance. In *IEEE Computer*, vol.30 (4), April 1997.
- [HAY 98] HAYTON R., HERBERT A., DONALDSON D. — FlexiNet - A flexible component oriented middleware system. In *Proceedings of ACM SIGOPS European Workshop*, Sintra, Portugal, September 1998.
- [HUR 95] HÜRSH W.L., LOPES C.V. — *Separation of Concerns*. Technical Report NU-CCS-95-03, College of Computer Science, Northeastern University, Boston, MA, February 1995.
- [JUL 88] JUL E., LEVY H., HUTCHINSON N., BLACK A. — Fine-grained mobility in the Emerald system. In *ACM Transactions on Computer Systems*, vol.6(1), p.109-133, February 1988.
- [KIC 91] KICZALES G., DES RIVIERES J., BOBROW D.G.— *The Art of the Metaobject Protocol*. The MIT Press, 1991.
- [KIC 97] KICZALES G., LAMPING J., MENDHEKAR A., MAEDA C., LOPES C., LOINGTIER J.M, IRWIN J. — Aspect-Oriented Programming. In *Proceedings of ECOOP'97*, LNCS 1241, Springer-Verlag, p.220-242, Jyväskylä, Finland, June 1997.
- [LED 96] LEDOUX T., COINTE P. — Explicit Metaclasses as a Tool for Improving the Design of Class Libraries. In *Proceedings of ISOTAS'96*, LNCS 1049, p.38-55, Springer-Verlag, Kanazawa, Japan, March 1996.
- [LED 97] LEDOUX T. — Implementing Proxy Objects in a Reflective ORB. In *Workshop CORBA : Implementation, Use and Evaluation*, ECOOP'97, Jyväskylä, Finland, June 1997.

- [LED 98] LEDOUX T. — *Réflexion dans les systèmes répartis : application à CORBA et Smalltalk*. PhD thesis, Université de Nantes, École des Mines de Nantes, March 1998. (in french)
- [LIE 96] LIEBERHERR K.J. — *Adaptive Object-Oriented Software : The Demeter Method with Propagation Patterns*. PWS Publishing Company, Boston, 1996.
- [MAE 87] MAES P. — Concepts and Experiments in Computational Reflection. In *Proceedings of OOPSLA'87*, ACM Sigplan Notices, p.147-155, Orlando, Florida, October 1987.
- [MAS 98] MASUHARA H., YONEZAWA A. — Design and Partial Evaluation of Meta-objects for a Concurrent Reflective Language. In *Proceedings of ECOOP'98*, LNCS 1445, p.418-439, Springer-Verlag, Brussels, Belgium, July 1998.
- [McA 95] MCAFFER J. — Meta-level architecture support for distributed objects. In *Proceedings of IWOOS'95*, p.232-241, Lund, Sweden, 1995.
- [MEY 92] MEYER B. — *Eiffel: The Language*. Prentice Hall, second printing, 1992.
- [OKA 94] OKAMURA H., ISHIKAWA Y. — Object Location Control Using Meta-level Programming. In *Proceedings of ECOOP'94*, p.299-319, LNCS 821, Springer-Verlag, July 1994.
- [OMG 95] OBJECT MANAGEMENT GROUP — *Object Management Architecture Guide*, Revision 3.0. OMG TC Document ab/97-05-05, June 1995.
- [OMG 97] OBJECT MANAGEMENT GROUP — *CORBAservices: Common Object Services Specification*. OMG TC Document formal/98-07-05, November 1997.
- [OMG 98] OBJECT MANAGEMENT GROUP — *The Common Object Request Broker : Architecture and Specification*, Revision 2.2. OMG TC Document formal/98-07-01, February 1998.
- [RIV 96] RIVARD F. — A New Smalltalk Kernel Allowing Both Explicit and Implicit Metaclass Programming. In *Workshop "Extending the Smalltalk Language"*, OOPSLA'96, San Jose, California, October 1996.
- [RIV 97] RIVARD F. — *Évolution du comportement des objets dans les langages à classes réflexifs*. PhD thesis, Université de Nantes, École des Mines de Nantes, June 1997. (in french)
- [ROM 99] ROMAN M., KON F., CAMPBELL R.H. — Design and Implementation of Runtime Reflection in Communication Middleware: the dynamicTAO Case. In *Workshop on Middleware*, ICDCS'99, Austin, Texas, May 1999.
- [SCH 99] SCHMIDT D., CLEELAND C. — Applying Patterns to Develop Extensible ORB Middleware. In *IEEE Communications Magazine*, 1999. (to appear)
- [SHA 86] SHAPIRO M. — Structure and Encapsulation in Distributed Systems : The Proxy Principle. In *Proceedings of the 6th International Conference on Distributed Computer Systems*, p.198-204, Cambridge, MA, May 1986.

- [SMI 82] SMITH B.C. — *Reflection and Semantics in a Procedural Programming Language*. PhD thesis, MIT, January 1982.
- [SUN 98] SUN MICROSYSTEMS — *Java Remote Method Invocation (RMI)*. At <http://java.sun.com/products/jdk/rmi/index.html>
- [WAT 88] WATANABE T., YONEZAWA A. — Reflection in an Object-Oriented Concurrent Language. In *Proceedings of OOPSLA'88*, ACM Sigplan Notices, p.306-315, San Diego, California, September 1988.