Università degli Studi di Milano
Dottorato in Informatica
(XII Ciclo)

PhD Thesis

Communication-Oriented Reflection: a Way to Open Up the RMI Mechanism

Candidate: Walter Cazzola,
Department of Computer Science
Università degli Studi di Milano

Supervisor: Francesco Tisato,
DISCo - Università di Milano Bicocca.

Advisor: Shigeru Chiba,
Institute of Information Science and Electronics
University of Tsukuba, Japan.

November 2000

Contents

Acknowledgements

I wish to thank Professor Shigeru Chiba from University of Tsukuba both for his interest in current and previous versions of this work and for his valuable contributions to the ideas presented in this thesis.

I wish to thank Professor Massimo Ancona from University of Genova for the opportunity he gave me introducing me to computational reflection, and Professor Francesco Tisato from University of Milano Bicocca for the unquestioning trust he had in me and in my work.

I would like also to thank my parents, my friends and my colleagues for their advice and their encouragement to improve my work and to go on this way.

I should also thank my former girlfriend Sabrina because without her this dream cannot be true, and Ombretta because if I didn't meet her this thesis would have been much better, but my life would have been less amusing.

Last but not least I wish to thank Ilaria, because with her yes she could represent my future.

That's all, Folks

— Chapter 1

Introduction

The Problem

From our experience, RMI-based frameworks and in general all frameworks supplying distributed computation seem to have some troubles. We detected at least three problems related to their flexibility and applicability.

Most of them lack in flexibility. Their main duty consists in providing a friendly environment suitable for simply realizing distributed computations. Unfortunately, interaction policies are hardwired in the framework. If it is not otherwise foreseen, it is a hard job to change, for example, how messages are marshaled|unmarshaled, or the dispatching algorithm which the framework adopts. Some frameworks provide some limited mechanism to redefine such details but their flexibility is limited from the possibility that the designer has foreseen.

Distributed algorithms are imbued in the applicative code breaking the well-known software engineering requirement termed as *separation of concerns*. Some programming languages like Java mask remote interactions (i.e., remote method or procedure call) as local calls rendering their presence transparent to the programmer. However their management, — i.e., tuning the needed environment to rightly carry out remote computations, and synchronizing involved objects — is not so transparent and easily maskable to the programmer. Such a behavior hinders the distributed algorithms reuse.

Object-oriented distributed programming is not distributed object-oriented programming. It is an hard job to write object-oriented distributed applications based on information managed by several separated entities. Algorithms originally designed as a whole, have to be scattered among several entities and no one of these entities directly knows the whole algorithm. This fact improves the complexity of the code that the programmer has to write because (s)he has to extend the original algorithm with statements for synchronizing and for putting in touch all the remote objects involved in the computation. Moreover the scattering of the algorithm among several objects contrasts with the object-oriented philosophy which states that data and algorithms managing them are encapsulated into the same entity, because each object can't have a global view of any data it manages, thus we could say that this approach *lacks of global view*. The lack of global view forces the programmer to strictly couple two or more distributed object.

A reflective approach, as stated in [19], can be considered as the glue sticking together distributed and object-oriented programming and filling the gaps in their integration. Reflection improves flexibility, allows developers to provide their own solutions to communication problems, and keeps communication code separated from the application code, and completely encapsulated into the meta-level.

Hence reflection could help to solve most of the troubles we detected. Reflection permits to expose implementation details of a systems, i.e., in our case allows to expose the interaction policies. It also permits to easily manipulate them. A reflective approach also permits to easily separate the interaction management from the applicative code. Using reflection and some syntactic sugar for masking the remote calls we can achieve a good separation of concerns also in distributed environments. Thanks to such considerations a lot of distributed reflective middleware have been developed. Their main goal consists both in overcoming the lacking of flexibility and in decoupling the interaction code from the applicative code.

By the way, reflective distributed middlewares exhibit the same troubles detected in the distributed middlewares. They still fail in considering each remote invocation in terms of the entity involved in the communication (i.e., the client, the server, the message and so on) and not as a single entity. Hence the global view requirement is not achieved. This is due to the fact that most of the meta-models that have been presented so far and used to design the existing reflective middlewares are object-based models. In these models, every object is associated to a meta-object, which traps the messages sent to the object and implements the behavior of that invocation. Such a meta-models inherit the trouble of the lack of global view from the object-oriented methodology which encapsulates the computation orthogonally to the communication.

Hence, these approaches are not appropriate to handle all the aspects of distributed computing. In particular adopting an object-based model to monitor distributed communications, the meta-programmer often has to duplicate the base-level communication graph into the meta-level augmenting the meta-program complexity. Thus, object-based approaches to reflection on communications move the well-known problem [50] of nonfunctional code intertwined to functional one from the base- to the meta-level. Simulating a base-level communication into the meta-level allows to perform meta-computations *either* related sending *or* receiving action, *but* not related to the whole communication or which involve information owned both by the sender and by the receiver without dirty tricks. This trouble goes under the name of *global view* lacking.

Besides, object-based reflective approaches and their reflective middlewares based on them allow only to carry out global changes to the mechanisms responsible for message dispatching, neglecting the management of each single message. Hence they fail to differentiate the meta-behavior related to each single exchanged message. In order to apply a different meta-behavior to either each or group of exchanged messages the meta-programmer has to write the meta-program planning a specific meta-behavior for each kind of incoming message. Unfortunately, in this way the size of the meta-program grows to the detriment of its readability, and of its maintenance.

Due to such a consideration, a crucial issue of opening a RMI-based framework consists in choosing a good meta-model which permits to go around the global view lacking, and to differentiate the meta-behavior for each exchanged message.

Our Solution

From the problem analysis we have briefly presented we learned that in order to solve the drawbacks of the RMI-based framework we have to provide an open RMI mechanism, i.e., a reflective RMI mechanism, which exposes its details to be manipulated by the meta-program and allows the meta-program to manage each communication separately and as a single entity. The main goal of this work consists in designing such a mechanism using a reflective approach.

To render the impact of reflection on object-oriented distributed framework effective, and to obtain a complete separation of concern, we need new models and frameworks especially designed for communication-oriented reflection, i.e., we need a reflective approach suitable for RMI-based communication which allows meta-programmer to enrich, manipulate and replace each remote method invocation and its semantics with a new one. That is, we need to encapsulate message exchanging into a single logical meta-object instead of scattering any relevant information related to it among several meta-objects and mimicking the real communication with one defined by the meta-programmer among such meta-objects as it is done using traditional approaches.

To fulfill this commitment we designed a new model, called *multi-channel reification model* [5]. The multi-channel reification model is based on the idea of considering a method call as a message sent through a logical channel established among a set of objects requiring a service, and a set of objects providing such a service. This logical channel is reified into a logical object called *multi-channel*, which monitors message exchange and enriches the underlying communication semantics with new features used for the performed communication. Each multi-channel can be viewed as an interface established among the senders and the receivers of the messages. Each multi-channel is characterized by its behavior, termed *kind*, and the receivers, which it is connected to.

$$\textbf{multi-channel} \equiv (\textbf{kind}, \textbf{receiver}_i, \ldots, \textbf{receiver}_n)$$

Thanks to this multi-channel's characterization it is possible to connect several multi-channels to the same group of objects. In such a case, each multi-channel will be characterized by a different kind and will filter different patterns of messages.

This model permits to design an open RMI-based mechanism which potentially overcomes the previously exposed problems.

In this way, each communication channel is reified into a meta-entity. Such a meta-entity has a complete access to all details related to the communications it filters, i.e. the policies related to both the sender, and the receivers side, and, of course, the messages it filters. A channel realizes a *close meta-system* with respect to the communications. It encapsulates all base-level aspect related to the communication providing the global view feature.

Of course, this model keeps all the properties covered by the other reflective models, such as transparency and separation of concerns. Hence the approach also guarantees to go around the problems already solved using reflection. Protocols and other realizative stuff are exposed to the meta-programmer manipulations, and the remote method invocation management is completely separated from the applicative code.

Moreover through the *kind* mechanism we can differentiate the behavior which is applied to a specified pattern of messages. So a set of multi-channels (each one with a different kind) can be associated to the same communication channel. Each channel will operate to a different set of messages. In this way the channel's code is related to a unique behavior it indiscriminately has to apply to all the messages it filters.

mChaRM [22] is a framework developed by the authors which opens the RMI mechanism supplied by Java. This framework supplies a development and run-time environment based on the multi-channel reification model. Multi-channels will be developed in Java, and the underlying mChaRM framework will dynamically realize the context switching and the causal connection link. A beta version of mChaRM, documentations and examples are available from:

`http://www.disi.unige.it/person/CazzolaW/mChaRM_webpage.html`

Such a system provided RMI-based programming environment. The supplied RMI mechanism is multi-cast (i.e., supplies a mechanism to remotely invoke a method of several servers), open (the RMI mechanism is fully customizable through reflection), and globally aware of its aspects. Some example of application are also provide.

This document is logically organized in: introduction to the problem, existing solutions to the problem, and finally my solution.

Practically, in chapter 2 I introduce the object-based distributed systems and programming, giving a little overview of the existing systems and stressing which are their limits and how the literature propose to overcome them (*introduction to the problem*). In chapter 3 I present the reflective middleware area and some existing reflective middleware which are considered as a possible solution to the object-based distributed systems limits (*existing solutions to the problem*). Chapters 4, 5, and 6 show my solution for customizing the Java RMI mechanism. Finally, in chapter 7 I show some applications of my solution extending the Java RMI behavior with new features considered nonfunctional to the implemented application.

— Chapter 2

Distributed, Object-Based Programming Systems

In this chapter we introduce some fundamental concepts about *distributed, object-based programming systems*, objects models issues in object management, and object interaction management, and briefly introduce some existing distributed, object-based programming systems. Finally we discuss some issues about the flexibility of the existing approaches and explain how some key concepts from the computational reflection area are useful to overcome these drawbacks.

2.1 Object-Based Distributed Systems and Programming

The development of distributed operating systems and object-based programming languages makes possible an environment in which programs consisting of a set of interacting objects may execute concurrently on a collection of *loosely coupled* processors. An object-based programming language encourages a methodology for designing and creating a program as a set of autonomous components, whereas a distributed operating system permits a collection of personal computers to be treated as a single entity. The amalgamation of these two concepts has resulted in systems that shall be referred as *distributed, object-based programming systems*. Some examples of distributed object-based programming systems are CORBA [48], Arjuna [72], Emerald [13, 14], Totem [68], the Java framework and its extensions like Jedi [3].

2.1.1 DOBPSs: Definitions and Properties

A *distributed, object-based programming system* (DOBPS) provides the features of an object-based programming system as well as a decentralized or distributed computing environment [29]. A distributed, object-based programming system typically has, but is not limited to, the following characteristics:

- *Distributions*. A distributed, object-based programming system combines a network of independent personal computer so that they provide a decentralized computing environment.

- *Transparency*. The system may hide the distributed environment or other underlying details from the users. It should also provide uniform access to all of the objects of the system.

- *Program Concurrency*. A distributed, object-based programming system should be able to assign the objects of a program to multiple processors so they may execute concurrently.

- *Object Concurrency*. An object should be able to serve multiple, nonmodifying invocation requests concurrently.

- *Improved Performance*. A well-designed program can typically execute more quickly in a distributed, object-based programming system than in a conventional system.

Object Granularity

A distributed, object-based programming system can be organized in either *large-grain objects*, or *medium-grain objects* or *fine-grain objects*. In the first organization each object is characterized by its large size, relatively large number of instructions it executes to perform an invocation and relatively few interactions it has with other objects. In the medium-grain organization each object can be created and maintained relatively inexpensively because it is smaller in size and in scope than larger-grain objects. In the last organization each object is characterized by its small size, small number of instructions it executes, and relative large number of interactions it has with other objects.

2.1.2 Objects Composition

The relationship between the processes and the objects of a distributed, object-based programming system characterizes the composition of the objects.

Passive Object

In the *passive object model*, the processes and objects of a distributed, object-based programming system are completely separate entities. A process is not bound nor is restricted to a single object. Instead, a single process is used to perform all the operations required to satisfy an action. Consequently a process may execute within several objects during its lifetime.

Active Object

In the *active object model* several server processes are created for and assigned to each object to handle its invocation requests. Each process is bound and restricted to the particular object for which it is created. When an object is destroyed, so are its processes. When a client makes an operation invocation, a process in the corresponding server object accepts the request and performs the operation on the client's behalf.

2.1.3 Object Management

Objects are the fundamental resources of a distributed, object-based programming system; therefore, their management is an essential function of these systems. A distributed, object-based programming system has to manage the activities of its actions. Actions should have the following three properties:

- *Serializability*. Multiple actions that execute concurrently should be scheduled in such a way that the overall effect is as if they were executed sequentially in some order.

- *Atomicity*. An action either successfully completes or has no effect.

- *Permanence*. The effects of an action that successfully completes is not lost.

We don't discuss about how these properties can be achieved because such a topic is largely discussed in the literature [90] and it is marginal to the scope of this thesis.

Another important function of a distributed, object-based programming system is to ensure that the activities of multiple actions that invoke the same object do not conflict or interfere with one another. To ensure that all actions have the property of seriazability and to protect the integrity of the objects' states, a synchronization mechanism is required. Many synchronization schemes exist, most of them can be classified as being either *pessimistic* or *optimistic* schemes [11].

Providing a security scheme to prevent unauthorized clients from successfully invoking an object, a scheme for detecting and recovering from failures, are also important function of a distributed, object-based programming system.

2.1.4 Object Interaction Management

A distributed, object-based programming system is responsible for managing the invocations between cooperating objects. When an action makes an invocation request, the system must locate the specified object, take the appropriate steps to invoke the specified operation, then possibly return a result.

Locating an Object

A distributed, object-based programming system should provide the property of *location transparency* so a client does not have to be aware of the physical location of an object in order to invoke it. Whenever an invocations is made, the system must determine which object was invoked and where (i.e., on which computer) the object currently resides in order to deliver the request to it. The system must assign an identifier to each object. These identifiers must be unique; they shouldn't change during the object's lifetime and once used shouldn't be reused. By the way, the mechanism for locating an object should be flexible enough to allow objects to migrate from a computer to another. The most used schemes for identifying and locating an object consist in:

- encoding the location of an object within its object identifier, or

- using a *(distributed) name server*.

The main drawback of the former approach is that an object is not permitted to move once it is assigned to a computer, since this would require its identifier to change. In the latter approach, there is a object or a group of objects, termed name server keeping up-to-date information about the location of each object in the system. The major problem with this approach is that the information maintained by the name server might be slightly out of synchronism since updating a name server is not an instantaneous operation and maintaining consistent information among the multiple components of the name server can be difficult.

Message Passing

A distributed, object-based programming system that provides the active object model typically supports the pure *message passing* scheme to handle object interactions. When a client makes an invocation on an object, the parameters of the invocation are packed into a request message. This message is then sent to a server process. A server process in the invoked object accepts the message, unpacks the parameters, and performs the requested operation. When the operation completes, the result is packed into a reply message, which is sent back to the client.

RPC/RMI

Remote Procedure Call (RPC) [12, 81] and *Remote Method Invocation* (RMI) [88] schemes slightly differ from the message passing scheme. In both schemes the packing, and the unpacking phases are encapsulated into stubs separated from the invoked objects playing the role of server processes. These stubs are usually created either by a specific designed preprocessor or by a compiler following a description (e.g., Java's interfaces or CORBA's IDL) of the services supplied by the remote object.

Direct Invocation

A distributed, object-based programming system that provides the passive object model typically supports the *direct invocation* scheme to handle object interactions. In the passive object model, a single process is responsible for performing all the operations associated with an action. As a result, a process will migrate from operation to operation and from object to object whenever the corresponding action makes an invocation. An invocation on a local object is similar to a procedure call (all the information are managed through the process stack, as for the procedure calls), whereas an invocation on a remote object is similar a remote procedure call, and involve the creation and destruction of worker processes which marshal, and unmarshal the request respectively on the two involved computers.

Brokers

A *request broker* [1] is a dedicated control mechanism that mediates interactions between client applications needing services and server applications capable of providing them. Brokers free clients from having to maintain information on where and how to obtain particular services. All objects, that belong to a distributed, object-based programming system, register the services they support as well as their locations, and the objects playing the role of client moreover interface with the broker. Typically, the broker interfaces with a naming server. There are two main broker models:

- A *forwarding broker* relays a client's request to the relevant server application, retrieves the response, and relays this reply back to the client.
- A *handle-driven broker* returns a service "handle" back to the client, which contains all the information required to interact with the server for the given service; the client uses this handle to issue its request directly to the server application, which then replies back to that client.

Broker method invocations and RPCs are similar, but there are important differences. With an RPC, you call a specific function

The direct invocation scheme should incur less performance overhead than the message passing scheme when it comes to local invocations since interactions between objects that reside on the same computer are relatively efficient. When it comes to remote invocations, on the other hand, the direct invocation scheme has the added expense of creating and destroying worker processes.

2.2 Object-Oriented Distributed Middleware: a Little Overview

A lot of efforts has been spent in designing and developing object-oriented and distributed systems in the last twenty years. In fact several DOBPS can be found in the literature, e.g., Amoeba [69], Argus [59], CHORUS [10], Delta-4 [73], Camelot [80], and so on. Nowadays the research trend in this area consists in designing and developing middlewares which provide the capabilities of a distributed, object-based programming system, i.e., researchers are designing and developing frameworks, which it is possible to build new DOBPS on without worrying about how achieve the DOBPS capabilities. In this section we give a brief overview of the most interesting and recent work in this area.

2.2.1 Java RMI Mechanism

Sun Microsystems few years ago has equipped its Java language with the support to remote method invocation [88]. Such an extension consists in a hierarchy of classes and a few tools which at compile time build the system structure.

RMI applications are often comprised of two separate programs: a server and a client. A typical server application creates a number of remote objects, makes references to those remote objects accessible, and waits for clients to invoke methods on those remote objects. A typical client application gets a remote reference to one or more remote objects in the server and then invokes methods on them. RMI provides the mechanism by which the server and the client communicate and pass information back and forth.

In the Java distributed object model, a *remote object* is one whose methods can be invoked from another Java Virtual Machine, potentially on a different

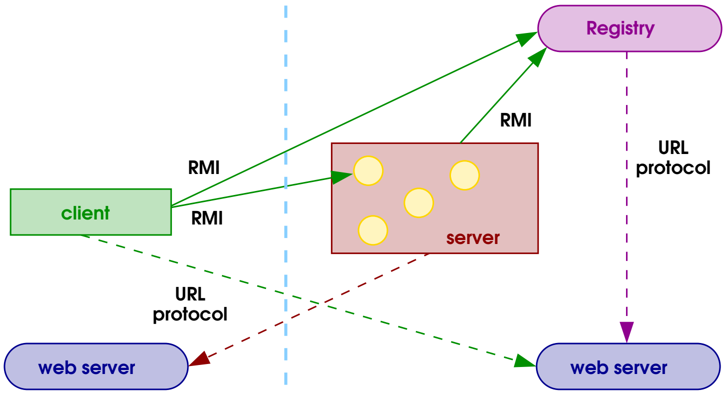

Figure 2.1: **Java RMI architecture.**

host. An object of this type is described by one or more remote interfaces, which are Java interfaces that declare the methods of the remote object.

Remote method invocation (RMI) is the action of invoking a method of a remote interface on a remote object. Most importantly, a method invocation on a remote object has the same syntax as a method invocation on a local object.

Figure 2.1 depicts an RMI distributed application that uses the registry to obtain references to a remote object. The server calls the registry to associate a name with a remote object — through a call to `Naming.rebind()`. The client looks up the remote object by its name in the server's registry — through a call to `Naming.lookup()` — and then invokes a method on it. The illustration also shows that the RMI system uses an existing web server to load Java class bytecodes, from server to client and from client to server, for objects when needed.

System Overview

RMI uses a standard mechanism for communicating with remote objects: stubs and skeletons. A stub for a remote object acts as a client's local representative or proxy for the remote object. The caller invokes a method on the local stub which is responsible for carrying out the method call on the remote object. In RMI, a stub for a remote object implements the same set of remote interfaces that a remote object implements.

When a stub's method is invoked, it does the following:

- initiates a connection with the remote virtual machine containing the remote object.
- marshals (writes and transmits) the parameters to the remote virtual machine
- waits for the result of the method invocation
- unmarshals (reads) the return value or exception returned

- returns the value to the caller

The stub hides the serialization of parameters and the network-level communication in order to present a simple invocation mechanism to the caller.

In the remote virtual machine, each remote object may have a corresponding skeleton. The skeleton is responsible for dispatching the call to the actual remote object implementation. When a skeleton receives an incoming method invocation it does the following:

- unmarshals (reads) the parameters for the remote method
- invokes the method on the actual remote object implementation
- marshals (writes and transmits) returns the result to the caller

Registry

The RMI system uses the `java.rmi.registry.Registry` interface and the `java.rmi.registry.LocateRegistry` class to provide a well-known bootstrap service for retrieving and registering objects by simple names.

A registry is a remote object that maps names to remote objects. Any server process can support its own registry or a single registry can be used for a host.

Remote Object Activation

Object activation is a mechanism for providing persistent references to objects and managing the execution of object implementations. In RMI, activation allows objects to begin execution on an as-needed basis. When a remote object is accessed (via a method invocation) if that remote object is not currently executing, the system initiates the object's execution inside an appropriate Java virtual machine.

During a remote method invocation the *activation protocol* is engaged. The activation protocol involves several entities: the *faulting reference*, the *activator*, an *activation group* in a Java virtual machine, and the *remote object* being activated.

The activator is the entity which supervises activation. An activation group is the entity which receives a request to activate an object in the Java virtual machine and returns the activated object back to the activator.

The activation protocols is as follows: a faulting reference uses an activation identifier and calls the activator to activate the object associated with the identifier. The activator looks up the object's activation descriptor (registered previously).

If the activation group in which this object should reside exists, the activator forwards the activation request to that group. If the activation group does not exist, the activator initiates a virtual machine executing an activation group and then forwards the activation request to that group.

The activation group loads the class for the object and instantiates the object using a special constructor that takes several arguments, including the activation descriptor registered previously.

When the object is finished activating, the activation group passes back a marshaled object reference to the activator that then records the activation identifier and active reference pairing and returns the active (live) reference to the faulting reference. The faulting reference (inside the stub) then forwards method invocations via the live reference directly to the remote object.

Example

The following code represents a small Java application which uses the remote method invocation mechanism. It implements a simulation for a stocks' market. The server supplies information about stocks' price.

```
public interface StockMarket extends java.rmi.Remote {
  float get_price( String symbol ) throws RemoteException;
}

public class StockMarketImpl extends UnicastRemoteObject implements StockMarket {

  public StockMarketImpl( String name ) throws RemoteException {
    try {
      Naming.rebind( name, this );
    }
    catch( Exception e ) {
      System.out.println( e );
    }
  }

  public float get_price( String symbol ) {
    float price = 0;
    for( int i = 0; i < symbol.length(); i++ ) {
      price += (int) symbol.charAt( i );
    }
    price /= 5;
    return price;
  }
}
```

whereas the client plays the broker's role, querying to the stocks' market information about stocks.

```
public class StockMarketClient {

  public static void main(String[] args) {
    try {
      if(System.getSecurityManager() == null) {
        System.setSecurityManager( new RMISecurityManager() );
      }
      StockMarket market = (StockMarket)Naming.lookup("rmi://localhost/NASDAQ");
      System.out.println( "The price of MY COMPANY is "
                          + market.get_price("MY_COMPANY") );
    }
    catch( Exception e ) {
      System.out.println( e );
    }
  }
}
```

2.2.2 Jedi: Java Environment for Distributed Invocation

The *Java Environment for Distributed Invocation* (Jedi) [3] is a framework for Java which provides a simple and flexible dynamic invocation service. Although message passing is a more general communication framework, developers are more comfortable reasoning about and using method calls to communicate between objects. Unfortunately, many existing remote method call systems are quite complex and have many steps to learn and repeat. Jedi's primary goal consists in simplifying or eliminating as many of these steps as possible. Conventional RPCs, for the most part, are based on static interface definition files, and on compiled and application dependent stubs, e.g., Java's interfaces [88], and CORBA's [47, 48] and DCE's [41] interface definition language. Through the dynamic invocation interface facilities, CORBA clients can discover resources dynamically, but dynamic invocation feature is somewhat difficult to use, many steps are required to construct a dynamic request object. Jedi's, instead, uses a proxy based approach, figure 2.2 shows how communications take place in the Jedi architecture.

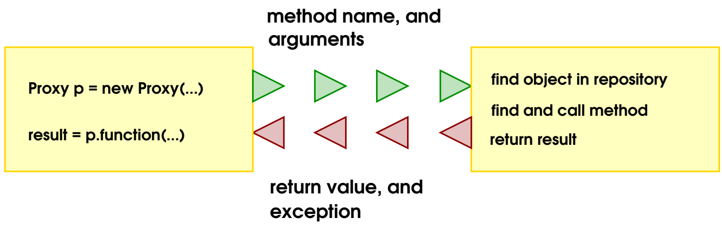

Figure 2.2: **Jedi's architecture**

Each object playing the server role registers itself in a repository using the `bind` method, as in the code below. The call to `Repository.local()` initialized the Jedi system to listen for incoming remote method calls on the default port.

```
import info.jedi.*;

public class ServerTest {
  public static void main(String args[]) {
    String string = "Hello";
    Repository.local().bind(string, "HelloString");
  }
}
```

The client in order to call a remote method has to create a `Proxy` object with the network address and the name of the object. Once a proxy has been created, any method can be called on a remote object by calling `function()`, with the method name and a vector representing its arguments, on the proxy. Underneath it all, the Jedi system will forward the method call information to the remote machine, which will find the object associated with the proxy. It will then look up the method with the correct name, invoke the method, and pass the return value back to the client.

```
import info.jedi.*;

public class ClientTest {
  public static void main(String args[]) throws Exception {
    Proxy proxy = new Proxy("thor.disi.unige.it", "HelloString");
    System.out.println("Its length is "+proxy.function("length"));
  }
}
```

The RPC scheme supplied by the Jedi system permits to use the intuitive remote method call paradigm without the complexity of many similar schemes. Jedi provides simple but dynamic remote method calls, giving programmers the ability to make run-time modifications. Because Jedi is a library-based system, it fits more naturally into the usual program-development cycle than precompiler-based RPC systems. A drawback of the dynamic scheme of Jedi is that there is no static type-checking. A method call can fail at run-time if the programmer makes a mistake and misspells the method or passes the wrong parameters.

2.2.3 CORBA

CORBA (Common Object Request Broker Architecture) [47, 48] is a specification, proposed by the OMG (Object Management Group) consortium, for a standard object-oriented architecture for applications. CORBA can be used to connect personal computers and their applications together without interfering with the existing hardware, network, and software infrastructure. CORBA provides the ability to:

- access distributed information and resources from within popular desktop applications;
- make existing data and systems available as network resources;

CORBA uses a *broker* to handle messages (called *requests*) exchanged between clients and servers in the system. The broker provides the ability to choose servers to best fill the client's request and separate the *interface* that the client sees from the implementation of the server. This separation lends itself well to producing a flexible, building-block approach where you can hide changes to the server from the client. Given that you do not modify the interface and its behaviors, you can create a new server or modify an existing server without changing the clients.

CORBA Architecture

Figure 2.3 depicts the CORBA's architecture. Each CORBA's application is composed by objects playing the role of either client or server. They interact through the ORB, and the BOA:

- The Object Requester Broker (ORB) is a single component, but has some functions specific to the client side and other functions specific to the server side. Figure 2.3 shows this connection using the shading around both the ORB (client-side) and the ORB (server-side) and their interfaces. The ORB handles invocation requests and the related selection of servers and methods. When an application sends a request to the ORB for an operation on an object, the ORB validates the arguments against the interface and dispatches the request to the server. On the server side, the ORB receives method dispatch requests, unmarshals the arguments, sets up the context state as needed, invokes the method dispatcher in the server skeleton, marshals the output arguments, and completes the invocation.

- The Basic Object Adapter (BOA) performs general ORB-related tasks, such as activating objects and implementations and registering server instances. The BOA connects the ORB to the methods in the implementation by using server skeletons. These server skeletons contain routines that map calls from the BOA to the method in a particular implementation that supports the requested operation.

Services provided by servers are described through *interfaces* written in a declarative C-like language, called *interface definition language* (IDL). From these interfaces are automatically generated the *server's skeletons*, and the *client's stubs* by an IDL-compiler. Stubs and Skeletons allow mapping CORBA's invocations into the server's and client's language-specific calls and vice versa. The following stub of code is an example of IDL interface for a stock quote system:

```
module Stock {
  exception Invalid_Stock {};
  interface Quoter {
    long get_quote(in string stock_name) raises(Invalid_Stock);
  };
};
```

The `Quoter` interface supports a single operation: `get_quote`. Its parameter is specified as an `in` parameter, which means that it is passed from the client to the server. When given the name of a stock as an input parameter, `get_quote` either returns its value as a `long` or throws an `Invalid_Stock` exception.

Clients can invoke the servers' methods via both *stub-style invocation* API and *dynamic invocation* API. The stub-style invocation API looks like a function call, and therefore is simpler to use than the dynamic invocation API. CORBA supports only synchronous communication for stub-style invocation. Instead the dynamic invocation API uses definitions found in the interface repository to enable the creation and invocation of requests to objects at run-time.

The CORBA software bus is the main module of this architecture and has the responsibility of achieving a transparent communication between remote objects. The modularity of this architecture is a major advantage of the OMG's 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 [48], and the request for proposal Messaging Service for CORBA 3.0, attempt to improve the existing specifications (and resulting applications).

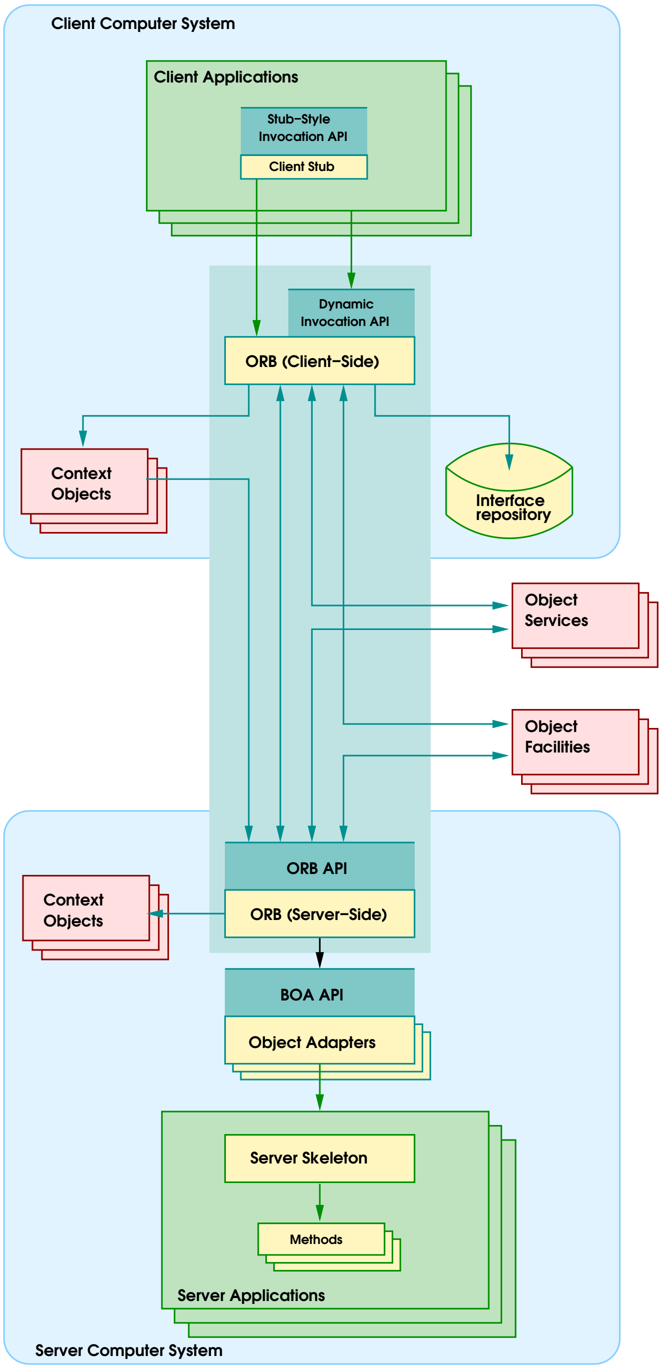

Figure 2.3: **CORBA's architecture**

2.3 Troubles and Limits of Middlewares

Lacks of Customizability

All the features of a distributed object-based programming system can be accessed either via direct call to the operating system, as it happens using CORBA, or through APIs supplied to the application by the underlying framework, e.g., using Jedi each communication has to pass through a `Proxy` object, and all provided services can be requested only calling `function()` method. In both cases the aspects characterizing the communications are difficult to customize by the programmer because they are out of the programmer's scope. Some customizations could be performed using the typical features of the object-orientation, such as inheritance, late binding, overriding, and so on. For example, in Jedi it is possible to change the method invocation behavior, in order to balance the load between two identical servers, building a new class, termed `BalancedProxy` which extends the `Proxy` class, and overrides the `function()` method, sorting the message to the least busy server.

Jedi

```
BalancedProxy proxy = new BalancedProxy("h1", "h2", "Test");
proxy.function("dummy_method");
```

Customizability of communications is restricted to what the designer provided to the programmer as customizable. That is, in spite of tips and tricks provided by the adopted methodology the programmer can adapt only the communication's aspects that the framework's designer foresaw as adaptable.

Hence, we can imbue the communication with an extra behavior like the load balancing feature, but we cannot change how the objects interact (e.g., from message passing to direct invocation). A similar kind of change continues to be a very hard job, because the choice about which scheme the system adopts is usually hardwired and hidden into the underlying framework, and not exposed to the application's programmer.

Separation of Concerns

A lot of specific code has to be written in order to render available remote services and to activate surrounding services like authentication processes. For example, objects playing the role of server have to register their services,

Jedi

```
Repository.local().bind(this, "stocks");
```

Java

```
Naming.rebind("stocks", this);
```

CORBA

```
int main(int argc, char *arv[]) {
  CORBA::ORB_var orb = CORBA::ORB_init(argc, argv, 0);
  CORBA::BOA_var boa = orb->boa_init(argc, argv, 0);
  My_Quoter quoter(quote_db);
  boa->impl_is_ready()
}
```

and objects requiring a remote service have to look for and bind to a representative of the remote object able to provide such a service;

Jedi
```
Proxy proxy = new Proxy("localhost", "stocks");
```
Java
```
StockMarket market = (StockMarket)Naming.lookup("stocks");
```
CORBA
```
ORB_var orb = ORB_init(argc, argv, 0);
Object_var obj = orb->resolve_initial_references("NameService");
NamingContext_var name_context = NamingContext::_narrow(obj);
Object_var obj = name_context->resolve("Quoter");
Quoter_var q = Quoter::_narrow(obj);
```

in order to respectively provide and get remote services.

Such steps are mandatory to carry out correctly remote cooperation, but many other steps should be done to configure the behavior of the required service, e.g., we have to choose which security policy, commitment policy, and synchronization policy to adopt for each specific communication. The following stub of Java code shows how it is possible to fix which security policy to adopt.

Java
```
public static void main(String[] args) throws Exception {
  if(System.getSecurityManager() == null) {
    System.setSecurityManager( new RMISecurityManager() );
  }
  StockMarketImpl myObject = new StockMarketImpl("stocks");
}
```

Thus the applicative code, we are developing, is intertwined with code necessary only to manage the remote computation. This extra code hinders the reusability of both the applicative code and the communication related code, because they are strictly coupled, i.e., applicative code contains statements that explicitly depend on distributed computing, and because the algorithm we are developing is based on information scattered among several entities, e.g., the code for synchronizing the remote computation is based on the status of the remote object.

Communications could be slightly decoupled from the applicative code using some syntactic sugar. Several details related to them could be hidden behind an appropriate syntax, rendering the code more clear and tidy. Anyway, such a syntactic sugar can only mask some details simplifying the code that the programmer has to write, but it cannot hide or avoid the use of synchronization code or other specific code used to carry out cooperation between processes, without limiting the system flexibility. Hence, in traditional approaches, distributed communications and their management cannot be completely decoupled from applicative code.

This phenomenon is due to the fact that the current distributed frameworks and languages don't respect the well-known properties of *separation of concerns* [50]. The absence of a good separation of concerns also hinders the reuse of modules implementing distributed properties.

Global View

As stressed in [37] the object-oriented distributed programming is not distributed object-oriented programming. It is an hard job to write object-oriented distributed applications based on information managed by several separated entities. Algorithms originally thought as a whole, have to be scattered among several entities and no one of these entities directly knows the whole algorithm.

This fact augments the complexity of the code that the programmer has to write because (s)he has to extend the original algorithm with statements for synchronizing, for retrieving remote information, and for putting in touch all the remote objects involved in the computation. Moreover, scattering of the algorithm among several objects contrasts with the object-oriented philosophy which states that data and the algorithms managing them are encapsulated into the same entity, because objects involved in distributed applications can't have local access to any data they have to manage. We could state that this approach *lacks of global view*. The lacking of global view forces the programmer to strictly couple two or more distributed objects, and limits the reusability of algorithms related to communications.

The lack of global view particularly affects algorithms related to communications, e.g., access control, checkpointing, load balancing, and so on. In order to clear the problem up, we consider the case of writing a load balanced system in which each service could be provided from several servers and we would choose to use the most unloaded server for each request. The load balancing algorithm relies on information held by the servers, i.e., their load. The natural, and naive, way to realize this algorithm consists in instructing the client to question its potential servers about their load and then evaluating which server is better to use. In this way, to get a remote service we introduce several extra communications from the client towards each server. On the other hand, it is a very hard job for the client to store loads information for every server in the system.

Hence the load balancing policy is scattered among client and servers, and its code is intertwined between their code, fruitlessly complicated from unnecessary code, and cannot be reused separately from the rest of the code. Whereas if the system benefits from the global view feature we will have an infrastructure which gather together such information and the decisional code simplifying the development of the load balancing algorithm.

2.4 Computational Reflection as a Solution

As discussed earlier, the object-oriented approach helps at structuring distributed programming concepts and mechanisms, thanks to encapsulation, genericity, class, and inheritance concepts. This approach offers a good compromise between customizability of the communication by the programmer and the amount of concepts that (s)he has to master in order to develop distributed applications. Unfortunately, a lot of code — code not directly related to the application that the programmer is developing — is intertwined with the rest of the code, reducing its readability and reusability. It would thus be interesting to make the approach more transparent and to expose more details of the framework to the programmer to simplify the development of distributed applications and the customization of the underlying framework they will use.

A reflective approach, as stated in [19], can be considered as the glue sticking together distributed and object-oriented programming and filling the gaps in their integration. Reflection improves flexibility, allows developers to provide their own solutions of communication problems, and keeps communication code separated from the application code, and completely encapsulated into the meta-level.

Reflection is a general methodology to describe, control, and adapt the behavior of a computational system. The basic idea is to provide a representation of the important characteristics/parameters of the system in terms of the system itself. Static representation characteristics, as well as dynamic execution characteristics of application programs are represented by a program which represents the default computational behavior (interpreter, compiler, execution monitor, ...). Such a description/control program is called a meta-program. Specializing such programs allows one to customize the execution of the application program, by possibly changing data representations, execution strategies, mechanisms and protocols. Note that the same language is used, both for writing application programs and for meta-programs controlling their execution. However, the complete separation between the application program and the corresponding meta-programs is strictly enforced.

Reflection helps to decorrelate libraries specifying implementation and execution models (execution strategies, concurrency control, object distribution) from the application program. This increases modularity, readability and reusability of programs. Reflection also provides a methodology to open up and make adaptable, through a meta-interface, implementation decisions and resource management, which are often hardwired and fixed, or delegated to the underlying operating system.

Reflection may also help to express and control resource management, not only at the level of an individual object, but also at a broader level such as: scheduler, processor, name space, object groups, such resources being also represented by meta-objects. This helps with a very fine-grained control (e.g., for scheduling and load balancing) with the whole expressive power of a full programming language [70], as opposed to some global and fixed algorithm (which is usually optimized for a specific kind of application or an average case).

Reflection provides a general framework for the customization of concurrency and distribution aspects and protocols, by specializing and integrating (meta-)libraries intimately within a language or system, while separating them from the application program.

As we show in section 3.2, reflective middlewares rely on such considerations. They develop, through a reflective approach, middlewares able to adapt themselves to the surrounding environment. A reflective approach allows them to naturally overcome the lacks of separation of concerns and of customizability, but it doesn't help with the global view trouble.

2.5 Conclusions

More about computational reflection and how this methodology can be used to overcome the problems of the existing DOBs showed in this section will be exposed in the next section. In that section we also show several of the existing approaches to reflective distributed middlewares pointing out how they try to treat the issues we raised.

— Chapter 3

Reflection, and Reflective Middlewares

In this chapter we face computational reflection from the point of view of its characteristics useful to overcome the DOBPS's limits, we have just exposed in the previous chapter. We also describe the most important reflective middlewares and how they treat such limits. Finally we briefly introduce which part of the trouble we face, and in which context.

3.1 Computational Reflection

Computational reflection or simply *Reflection* is a programming paradigm suitable to develop *open systems*. An open system is a software system which can be extended in a simple manner. Open systems provide significant benefits to software development. The system can comprise parts developed at different times by independent teams. It is possible to use specific meta-entities to test the system and then discard such meta-entities, when the test phase ends, removing the meta-level from the final system. Computational reflection improves the software reusability and stability, reducing development costs.

Nowadays, computational reflection has been used in several fields, for example for developing operating systems [96, 44], fault tolerant systems [36, 4], compilers [55], and also for building distributed frameworks [53, 57, 31, 61, 2, 65, 94].

3.1.1 What is Reflection?

Reflection appeared first in AI before propagating to various fields in computer science such as logic programming, functional programming and object-oriented programming [32].

It was introduced in object-oriented programming thanks to the famous works of Pattie Maes [62, 63].

Reflection is the ability of a system to watch its computation and possibly change the way it is performed. Observation and modification imply an "underlay" that will be observed and modified. Since the system reasons about itself, the "underlay" is itself, i.e. the system has a *self-representation* [63].

Bobrow et al. consider that observation and modification are two aspects of reflection:

> "Reflection is the ability of a program to manipulate as data something representing the state of the program during its own execution. There are two aspects of such manipulation: *introspection* and *intercession*.
>
> Introspection is the ability for a program to observe and therefore reason about its own state. Intercession is the ability for a program to modify its own execution state or alter its own interpretation or meaning. Both aspects require a mechanism for encoding execution state as data; providing such an encoding is called *reification*." [16]

An object-oriented reflective system is logically structured in two or more levels, constituting a *reflective tower*. The first level is the *base-level* and describes the computations that the system is supposed to do. The second one is the *meta-level* and describes how to perform the previous computations. The entities (objects) working in the base level are called *base-entities*, while the entities working in the other levels (meta-levels) are called *meta-entities*.

Each level is *causally connected* to adjacent levels, i.e. entities working into a level have data structures reifying the activities and the structures of the entities working into the underlying level and their actions are reflected into such data structures. Any change to such data structures modifies entity behavior. Each level, except the first and the last one, is a base-level for the above level and is a meta-level for the underlying level.

Meta-entities supervise the base-entities activity. The concept of *trap* could be used to explain how supervision takes place. Each base-entity action is trapped by a meta-entity, which performs a meta-computation, then it allows such base-entity to perform the action.

The infinite regression of the reflective tower can be managed in different ways. Brian Smith suggested the use of lazy evaluation in 3-Lisp [79]: an interpreter is not created unless needed. Others solutions are represented by the following techniques: *meta-circularity* [30] and *meta-helix* [52].

It is possible to observe, going beyond the reflective tower of compilers|interpreters, that each reflective computation can be separated into two logical aspects: computational flow context switching and meta-behavior. A computation starts with the computational flow in the base level; when the base-entity begins an action, such action is trapped by the meta-entity and the computational flow raises at meta-level (*shift-up* action). Then the meta-entity completes its meta-computation, and when it allows the base-entity to perform the action, the computational flow goes back to the base level (*shift-down* action).

With respect to the typology of actions performed by the meta-entities we can classify the computational reflection in two branches: *structural*, and *behavioral reflection*.

Structural Reflection

Structural reflection can be defined as the ability of a language to provide a complete reification of both the program currently executed as well as a complete reification of its abstract data types.

Structural reflection allows to reify and to manipulate the computational system code. From early times, functional languages (e.g., lisp) and logic languages (e.g., prolog) have statements allowing to manipulate the program representation. Such statements are based on the interpretative nature of such languages and they make it easier to introduce structural reflection into it. Most object-oriented languages are compiled (e.g., C++, Oberon and Eiffel) and there is no code representative at run-time. Only pure object-oriented languages, like SmallTalk, have code representative at run-time (i.e., the class) and it is such representative which realizes the structural reflection, see [76, 35]. Structural reflection in non pure object-oriented languages is realized in one of the following ways, but in all cases its potentials are limited: at compile-time as in OpenC++ from version 2 [26], or introducing run-time structures representing (reifying) program code [82]. In the former method all structural reflective actions are static, in the latter the limit is represented by which aspect of the code is reified (e.g., in [82] the structural reflection consists only in substitute methods code using first class procedures feature of the Oberon language).

Behavioral Reflection

Behavioral reflection can be defined as the ability of the language to provide a complete reification of its own semantics as well as a complete reification of the data it uses to execute the current program.

Behavioral reflection manipulates the behavior of the computational system. The manipulation is realized by two phases: *method look-up or shift-up action* and *message application or shift-down action*. In the former phase, when the base-entity sends a message (calls a method) the meta-entity intercepts it and looks for the meta-computation to perform for that message. The computational flow shifts from the base-level up to the meta-level. In the latter phase, the meta-entity has the control of the computation and applies, if needed by the meta-computation, the requested message and then it returns the results to the base-entity. The computational flow shifts from the meta-level down to the base-level. Examples of behavioral reflection realization are [25, 40].

3.1.2 Reflection's Characteristics

The main features of computational reflection, which both the development and the execution phase of the reflective system can benefit from, are *transparency, separation of concerns, extensibility, visibility*, and *granularity*.

Transparency

Transparency, as stated in [63], a reflective system is logically structured into a tower of several levels and the entities of each level work independently from the work of the entities of the level above. Thus, introspection and intercession should be performed transparently; the *transparency degree* defines how transparently introspection and intercession are performed, more on transparency in [83]. The transparency degree can be measured through the number of changes that must be made to the base-level code to integrate it with the meta-level.

Extensibility, and Separation of Concerns

Extensibility, and *Separation of Concerns*, in reflection philosophy, each different system's functionality is the concern of a different level of the reflective tower, i.e., the base-level performs its functional aspect[1] and each level extends the system, composed of the underlying tower levels, with different non-functional aspect[2] (e.g., fault tolerance [36], atomicity [85], concurrence [27] and persistence [58]). Thus reflection permits to extend a computational system. Entrusting a different aspect to a different level is termed *separation of concerns* [50].

Transparency, extensibility and separation of concerns are very desirable goodies during the development phase. They allow the separated development of each level saving time and money, improving component reuse and consolidating the correctness and the stability of each aspect of the system.

Visibility

With *visibility* we mean the scope of the meta-computation, i.e., which base-entities or base-entities' aspects can be involved by the meta-entity's meta-computation. We term *global view* the situation in which the meta-computation can involve all base-entities and aspects involved in the computation which it reifies. Visibility is a measure of the homogeneity of the meta-computation with respect to the normal computation.

Granularity

With *reflection granularity*, we mean the smallest aspect of the base-entities of a computational system (e.g., objects and methods) that can be reified by different meta-entities. The most interesting granularity levels are: *classes, objects, methods* and *method calls*. For example, if granularity is at method level, two methods of the same object can be reified by two different meta-entities and thus they can manifest two different meta-behaviors. A fine granularity permits more flexibility and modularity in the software system at the cost of meta-entities proliferation. Of course with any reflection granularity level it is possible to simulate the behavior achieved by the other levels. Using a coarse granularity, to simulate a finer granularity behavior, we must develop the meta-entities code as a large case, improving the complexity of the meta-entities and each case branch handles a different meta-behavior for a different base aspect. By contrast, with a fine granularity to simulate a coarser granularity behavior it is sufficient to reify all the base aspects needing the same meta-behavior by the same meta-entity.

[1] where for functional aspect, we mean the minimal computation needed to solve the problem aimed to the system.

[2] where for non-functional aspect we mean properties marginals to the problem to solve.

3.2 Reflective Middlewares

In the last few years the reflective middleware research branch has been spawned from the reflection area. Their main goal is to study, design, and develop *reflective middlewares*. From Geoff Coulson[3] words:

> *Reflective middleware* is simply a middleware system that provides inspection and adaptation of its behavior through an appropriate causally connected self representation.

Reflection in reflective middlewares wants to render middleware more adaptable to its environment and better able to cope with change. That is, the main goal of this research area consists in solving some of the issues we described in the previous chapter such as to open the communication mechanism, and improve the system adaptability.

In the sequel we give a brief overview of some existing reflective middleware and a brief analysis of their weak and strong points, with particular regards to detect what we need in order to solve the previously detected open issues.

3.2.1 GARF

GARF [42, 49] is an object-oriented distributed system that supports transparency "à la carte". That is, a distributed programming environment which provides programmers with a set of behavioral abstractions that hides low level features related to concurrency, persistence, distribution and fault-tolerance. Nevertheless, programmers must be able to refine such abstractions in order to obtain the desired efficiency. GARF provides built-in behavioral abstractions that deal with concurrency, persistence, distribution, and fault-tolerance, which can be used as such, or refined at programmers' convenience.

GARF promotes software modularity by separating the functional aspects of applications, from their behavioral features. The former are designed within data objects, similar to objects in classical centralized languages, whereas the latter are confined within particular objects named *encapsulators* and *mailers*. Encapsulators are used to control how data objects send and receive messages, whereas mailers are used to perform communications between encapsulators. GARF provides a built-in extensible library of encapsulator and mailer classes that implement very flexible behavioral features such as client/server asynchrony, active/passive replication, semantic-based concurrency control, etc. These classes can be used as such, or refined according to application's semantics.

[3] Geoff Coulson is member of the Distributed Multimedia Research Group of the Lancaster University. He is also the editor of the Reflective Middleware web page

`http://computer.org/dsonline/middleware/RM.htm`

such a page represents the main referring point of the reflective middleware community.

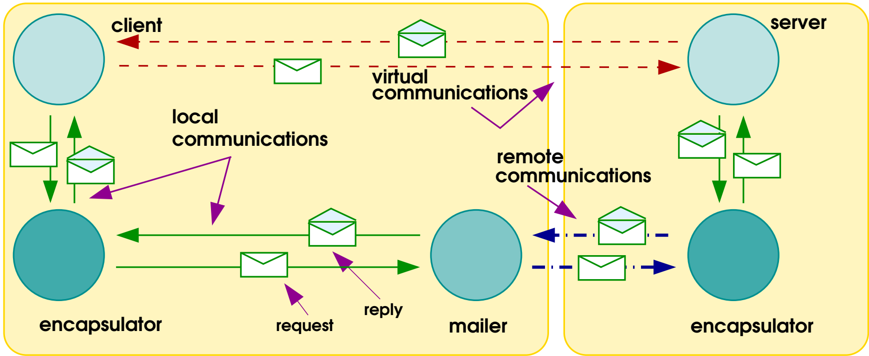

Figure 3.1: **GARF's Architecture**

GARF model

The GARF computational model is based on two programming levels. A high level, called *functional level*, in which the programmer describes aspects of the application that could be expressed in a centralized, volatile, and sequential object-oriented system; and a low level, called *behavioral level*, in which programmer describes behavioral features related to concurrency, persistence, distribution, and fault-tolerance.

Encapsulators

Encapsulators are used to wrap data objects by controlling how the latter send and receive messages. Each data object can be bound to an encapsulator. The encapsulator intercepts each message, sent or received by its associated data object, in a transparent way, i.e., data objects are not aware of encapsulators. So, every message sent to the data object by another data object passes through its encapsulator, as well as every message sent by the object. A data object and its encapsulator are always located on the same node. According to its semantics, the encapsulator performs pre and post-actions which handle asynchrony, concurrency control, persistence, replication, and so on.

Mailers

Mailers are used to perform (remote) communications between encapsulators. Each data object bound to an encapsulator, is also bound to a mailer class. A mailer of this class is created for each request sent to the encapsulator. The mailer is created on the client node and is used to transmit the request to the encapsulator and to get back the reply. According to their semantics, mailers perform appropriate communications, e.g. one to one, multicast, atomic multicast, and so on.

Figure 3.1 shows how a distributed system will be organized in GARF. Both client and server are respectively linked to encapsulators. Encapsulators cooperate with a mailer. Such a mailer is entrusted to effectively deliver the remote communication.

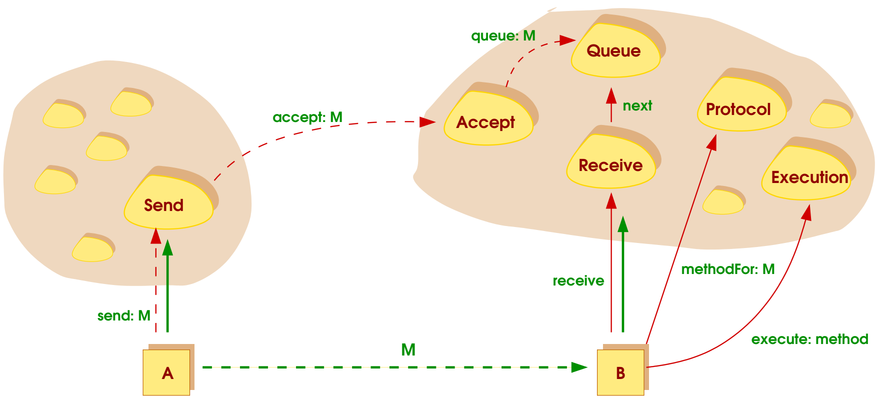

Figure 3.2: **CodA's Architecture**

3.2.2 CodA

CodA [65, 66] is a meta-level architecture for describing a wide range of object behavior models. CodA can be thought as a generic object engine framework in which users define, on a per-object or even per-use basis, how object's behave computationally.

The key concept in the design of the CodA meta-level architecture is the 'decomposition' or 'factoring' of the meta-level into fine-grained objects or meta-components. The meta-level is just an application whose domain happens to be the behavior of objects. The meta-level is defined as playing a number of roles in the description of base-level object behavior. Each role is filled by a meta-component and corresponds to some behavior such as; object execution (both mechanisms and resources), message passing, message to method mapping and object state maintenance. Roles may be filled by many different components and components can sometimes fill several roles. An object's behavior is changed by explicitly redefining components or by extending the set of roles.

The CodA meta-level architecture is largely run-time oriented. It does not provide integral support for language constructs like classes which are required for the static description of object behavior. Rather, these constructs are borrowed from whatever language is used to implement CodA.

CodA differs from the other reflective frameworks in two other related ways; granularity and decomposition into objects. A common approach to factoring object execution behavior (e.g., message sending or method execution) is to create public interface methods on a small number of meta-level objects. That is, object behavior is decomposed into code at the meta-level. Changes to a behavior are made by modifying its related interface methods.

In addition to the meta-level infrastructure of CodA, there is the set of seven components each of which describes some behavior in the basic execution model.

Meta-Level

CodA specifies that every object has a conceptual meta-level. The meta-level is not a single object but rather a set of meta-components each of which describes some aspect of base-level object behavior. The implementation of the meta-level is not defined except to say that a particular set, and thus a particular meta-level, can optionally be: fixed and allow no changes, changeable in that existing components can be replaced or extensible by allowing new roles and components to be added to the meta-level.

Meta-level programmers need some way of shifting to the meta-level and of accessing an object's meta-components. CodA adopts the technique of sending a special message called `meta` to some object. The `meta` is used for both shifting and accessing. All messages sent to a meta are executed at the meta-level and metas 'broker' (i.e., provide access to) meta-components.

Meta-Components

As a basis, CodA defines a default set of seven components which are present at the meta-level of all objects; `Send`, `Accept`, `Queue`, `Receive`, `Protocol`, `Execution` and `State`. They do not cover every possible aspect of object behavior, but the set is extensible and the standard seven cover the behaviors essential to common object models. See figure 3.2 for a sample meta-level configuration and interaction.

3.2.3 OpenCORBA

OpenCORBA [56, 57] is an application implementing the API CORBA in NeoClasstalk. It reifies various properties of the broker — by the means of explicit meta-classes — in order to support the separation of the internal characteristics of the ORB. The use of the dynamic change of meta-class allows to modify the ORB mechanisms represented by meta-classes. Then, OpenCORBA is a reflective ORB, which allows to adapt the behavior of the broker at run-time.
Two basic aspects of the CORBA software bus have been reified:

❶ the mechanism of remote invocation via a proxy;

❷ the IDL type checking on the server class;

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 meta-class `ProxyRemote` implementing the remote invocation mechanisms; the template class is associated with the meta-class `TypeChecking` which implements the IDL type checking on the server. Figure 3.3 shows the architecture adopted by OpenCORBA.

Proxy

The proxy object is a local representation 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 [78].

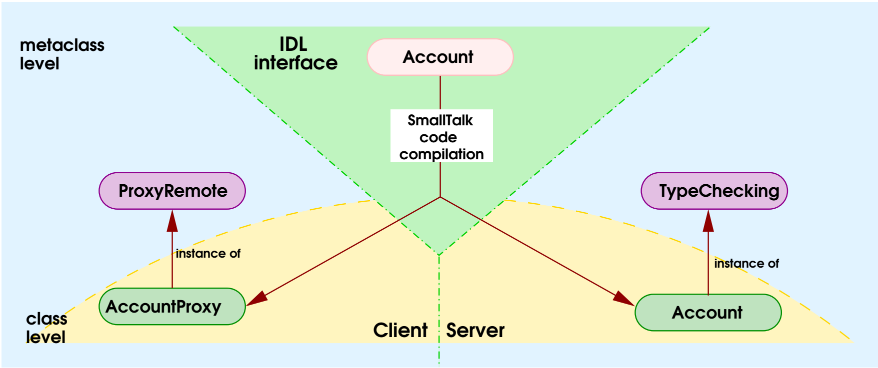

Figure 3.3: **OpenCORBA's architecture**

The remote invocation mechanism is completely independent of the semantics of the IDL interface. 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 meta-class `ProxyRemote` in charge of calling the real object. The remote invocation remains thus transparent for the client.

Methods of the proxy class are purely descriptive and represent interfaces like IDL operations.

The proxy object intercepts the exchanged message at its reception time then it launches a remote invocation. The meta-class `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 dynamic interface invocation CORBA API.

IDL Type Checking

The CORBA specification states 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. 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 meta-class. In OpenCORBA, the code of the server class does not carry out any test on the type of the arguments. The meta-class `TypeChecking` 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 the programmer to check the type of the arguments and the type of the result of the method. Thus, OpenCORBA externalizes the type checking, at the same time from the lower layers of the ORB, and from the server class.

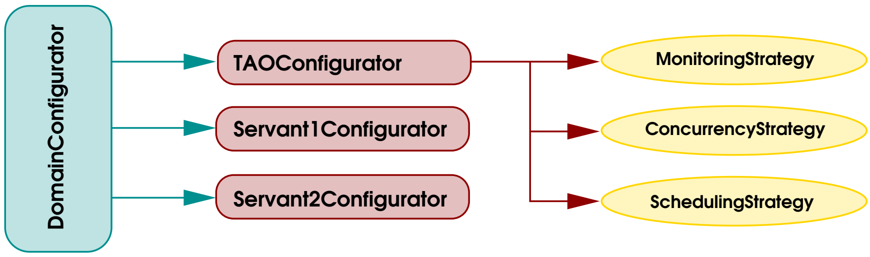

Figure 3.4: **DynamicTAO Architecture**

The dynamic adaptability both of the invocation mechanisms, and of the type checking in OpenCORBA is achieved developing others meta-classes respectively from `ProxyRemote`, and `TypeChecking`.

3.2.4 DynamicTAO

DynamicTAO [53, 54] is a CORBA reflective ORB. It allows inspection and reconfiguration of its internal engine. It achieves that by exporting an interface for:

❶ transferring components across the distributed system,

❷ loading and unloading modules into the ORB runtime, and

❸ inspecting and modifying the ORB configuration state.

Reification in DynamicTAO is achieved through a collection of entities known as *component configurators*. A component configurator holds the dependencies between a certain component and other system components. Each process running the DynamicTAO ORB contains a component configurator instance called `DomainConfigurator`, which maintains references to instances of the ORB and to servants running in that process. Each instance of the ORB contains a customized component configurator called `TAOConfigurator`.

`TAOConfigurator` contains hooks to which implementations of DynamicTAO strategies (e.g., for security, scheduling and so on) are attached. Hooks work as *mounting points* where specific strategy implementations are made available to the ORB. With this architecture (depicted in figure 3.4) is possible to manage strategy reconfiguration consistently.

Component implementations are shipped as dynamically loadable libraries, so they can be linked to the ORB process at runtime. They are organized in categories representing different aspects of the ORB internal engine or different types of servant components.

A *Network Broker*[4] receives reconfiguration requests from the network and forwards them to the *Dynamic Service Configurator*[4]. The latter component contains the `DomainConfigurator` and supplies common operations for dynamic configuration of components at runtime. It delegates some of its functions to specific component configurators (e.g., `TAOConfigurator` or a certain `ServantConfigurator`).

3.3 Reflective Middlewares Analysis and Open Issues

In the previous section we showed only few of the most interesting and representative reflective middleware. CodA and GARF are two milestones in the area. Several current projects on reflective middlewares have been inspired by them. Whereas OpenCORBA and DynamicTAO represent the next generation in reflective middleware area together with many others such as OpenORB [15, 31, 71], R-Rio [60, 61, 89], Coyote [44, 45, 33], and so on. In this section, we summarize because they result inadequate to carry out meta-computation on communications.

GARF

GARF moves in the meta-level all communication stuffs, decoupling such a code from the application code. Unfortunately, each communication it is reified and managed by two kind of objects: mailers and encapsulators. The former carries out computations on the trapped message, while the latter carries really out the interprocess communication. Such a decomposition doesn't permit to every of such entities to have a global view of the whole reified communication.

OpenCORBA

OpenCORBA adaptability is limited to the invocation mechanism and to the kind of type checking applied to the requested services. Two separate entities are charged to carry out such a customization and it is not foreseen their collaboration. The global view concept is not provided and several communication's aspects (such as brokers, name servers, and so on) are not exposed to the meta-program, but should be simple to extend the OpenCORBA system in order to also open up such aspects.

dynamicTAO

The DynamicTAO approach is quite different from the approach the other reflective middlewares adopt. Such an approach is very low-level and quite powerful to change the strategies the system adopts, but it is not so simple and powerful when programmers want to extend the communication behavior with features not directly related to strategies already involved in communication, e.g., to introduce from scratch a message checkpointing policy.

[4]Network Brokers and Dynamic Service Configurators are two kinds of DynamicTAO components.

CodA The CodA system provides a reflective framework really oriented to communications. Communications' aspects are easily reified in specific meta-entities. Several meta-entities cooperate to define the behavior to apply to messages they manage. The mechanism is quite near to solve each issues we detected, but it is not transparent the programmer knows the existence of such meta-entities and (s)he has to explicitly call them through the `meta` message. Besides it doesn't foresee a simple mechanism to vary the applied meta-behavior in dependence of the single exchanged message.

As just summed up, all the considered reflective middlewares, either presented or not, don't cover all the open issues in middlewares we detected. Usually they face well the lacks of flexibility and the separation of concerns, but continue to fails in managing each communication as a whole suffering hence of the global view lacking. Most of the reflective middlewares focus their efforts in render more adaptable communication mechanisms.

3.4 Conclusions

As stressed in this chapter several efforts has been done to cover some of the previously exposed open issues related to distributed middlewares. However, such efforts were done to render adaptable existing middlewares through reflection. This approach overcomes only part of the problems, improving middlewares flexibility and the separation of communication aspect from the applicative aspect. Unfortunately, that one is not enough to overcome all the detected issues and open a RMI mechanism. In most cases they don't consider the orthogonality between object-oriented encapsulation and the remote communications, thus they don't achieve a global view from the meta-level on the communications they manage. As we see in the next chapter the global view lacking could be overcome changing the reflective model they adopt from a object-based to an communication oriented one.

— Chapter 4

To Open Up the RMI Mechanism

This chapter can be considered as the main part of our project because it sums up the open issues about the existing approaches to communication oriented reflection. It also shows how we face such troubles in the specific case of opening up the RMI mechanism. In particular we consider the most general case consisting in a remote method invocation involving many servers, i.e., the same service is required to a group of servers all of them able to satisfy such a request. In order to manage this problem we introduce our reflective model, termed *multi-channel reification model*, which is especially designed to open up, i.e., reflectively extend, the multi-rmi mechanism. Finally, we look into the existing reflective models in order to determine their limits in manipulating communications.

4.1 Open Issues in Reflective Middlewares

As asserted in chapter 3, the existing approaches to carry meta-computations out on distributed communications, i.e., reflective middlewares designed to transparently extend communications among several distributed objects, are able to face the lacks of customizability and of transparency of classic middlewares (see chapter 2).

Unfortunately, they continue to model each communication in terms of its components and not as a whole entity. Hence, both sender, receivers, and exchanged message could be reified in separated and interacting meta-entities. Moreover all the reflective middlewares based on this approach suffer of the *lacks of global view* drawback. Usually, the absence of global view forces the meta-programmer to add code to the meta-programs for synchronizing and keeping each reified aspect of the communication in touch with the components reifying other communication aspects. Such an added code contributes to augment the complexity of the meta-level program and then to render more probable the possibility to introduce errors in it. Besides, to render explicit the communication interface of meta-entities reifying different parts of the same entity hinders the reuse of each single meta-entities separately from the others.

Most of such reflective middlewares allow only to carry out global changes to the mechanisms responsible for message dispatching, neglecting the management of each single message. Hence they fail to differentiate the meta-behavior of each exchanged message. In order to implement different meta-behavior for either a single or a group of messages the meta-programmer has to write the meta-program planning a specific meta-behavior for each kind of incoming message. Unfortunately, in this way the complexity and the size of the meta-program grows to the detriment of its readability, and of its maintenability.

4.2 How We Face Such Open Issues

As already stressed reflection is a good way to both separate the communication mechanism from the rest of the code, and open up such a mechanism, but it is our opinion that current reflective approaches don't take the possibility of reifying a communication and the objects which involves as a single meta-entity into consideration.

Hence, in order to overcome all the exposed troubles we have to design from ground a reflective approach oriented to reify and to reflect on communications. This means to design a reflective model whose domain is not a base-object but a communication, i.e., an exchanged message, the involved base-objects, and the communication's policies adopted by the system.

Extending the classic reflective approach to consider communications as their applicative domain allows to maintain all the typical reflective advantages, e.g., causal connection, transparency, separation of concerns and so on. Besides, in this way an abstract entity, as a communication, usually scattered among several entities can be reified into a single meta-entity which has a global view of the original communication and through the causal connection property can simply alter it.

Then, it is possible to use such an approach to open up any kind of remote communication (in our case the RMI mechanism), keeping all the advantages due to a reflective approach as in the analyzed reflective middlewares, and also achieving a global view on the reified communications.

4.3 (Multi-)Channel Reification Model

In this section we present the core of our work: the *multi-channel reification model*. This model describes a reflective architecture specially designed to extend the well-known client-server paradigm and in particular to extend the RMI mechanism. Our idea consists in describing an architecture suitable for reifying each step of the RMI (or multi-RMI) mechanism, freeing such a mechanism and its configuration from the base-level program, and overwhelming the previously described limits.

4.3.1 What a Multi-Channel Is

The *multi-channel reification model* is based on the idea of considering a method call as a message sent through a *logical channel* established among a set of objects requiring a service, and a set of objects providing such a service. We propose to

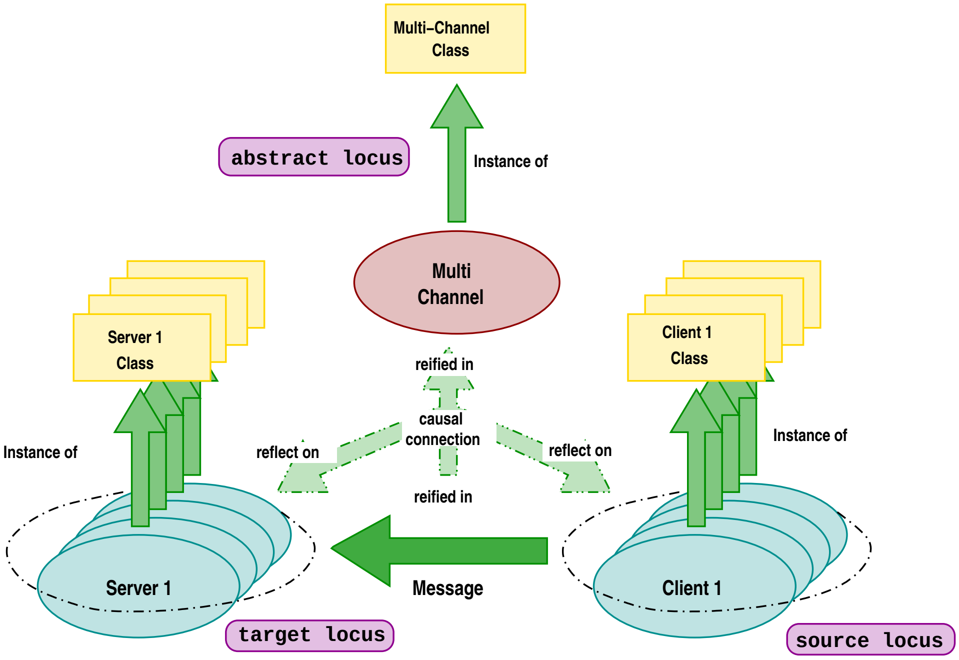

Figure 4.1: **Multi-Channel Reification Model**

reify this logical channel into a *logical object*, called *multi-channel*, which monitors message exchange and enriches the underlying communication semantics with new features used for the specific communication performed. We emphasize that we are talking about logical entities, the communication channel is embodied in a unique logical meta-entity, i.e., the multi-channel. At the model abstraction level we don't take care that this logical object might be implemented by several objects. Each multi-channel can be viewed as an interface established among the requiring objects (in the sequel termed as *senders*) and the objects satisfying such a request (in the sequel termed as *receivers*)[1]. Such an interface will hide intra-(logical-)objects communications in favor of an homogeneous management of the enrichment of the message exchanging semantics. Each method call is trapped by the multi-channel when originated, then it is imbued with the new semantics, and finally it is delivered to the appropriate destinations.

4.3.2 The Kind of the Meta-Entities Behavior

One of the open issues related to the communication oriented reflection is represented by the fact that to extend in a different way two different communications,

[1] In order to simply the exposition, we treat senders and receivers as separated entities, but this choice is not restrictive, a sender and a receiver can be identified without problems. In this case, the multi-channel will look like a meta-object able to reflect both on outgoing, and on ingoing messages.

i.e., two different remote method invocations, is a hard job and involves the writing of a lot of specific[2] code. In order to avoid this trouble we organize the meta-level in several (multi-)channels, which are characterized by *how* they extends the remote method invocations they will manage, i.e., which meta-behavior they will carry out on the trapped remote method invocation. Each multi-channel is devoted to enrich the semantics of the communication channel it embodies, and the kind of meta-behavior carried out is termed *kind*.

Hence, a message can be managed with a different semantics following the kind of the multi-channel entrusted to manage it. Each remote method invocation is associated with the kind of meta-behavior the programmer wants to carry out on such an invocation. Typically messages are classified thanks to the groups of senders and receivers exchanging them and the extra-behavior the programmer wants to add to such an exchange, i.e., the kind of the meta-behavior used to managed such a communication. Then, several multi-channels can be established among the same groups of senders and receivers, each of them implementing a different behavior and monitoring a different set of messages exchanged among the same referents. In the sequel multi-channels will be labelled through their kind, and we adopt the notation ≪`kind`≫.

An example of situation, which is useful to adopt many meta-behaviors in, is represented by a critical distributed system with methods able to alter the objects' state and methods only able to query the objects' state. In order to render safe a similar system, it is inevitable to add a checkpointing mechanism of exchanged messages. In this specific case it is convenient to save information only about methods whose execution causes state changes, i.e. to partition messages into two sets: non-modifying and modifying messages. Thus, two meta-behaviors (i.e., two kinds) will be defined: one managing non-modifying messages, even, simply delivering the message to the designed destination, i.e., kind ≪`normal`≫, and the other handling the modifying messages with an enriched semantics defining the expected checkpointing mechanism, i.e., kind ≪`checkpointing`≫. Of course this two kinds will be embodied by two multi-channels established between the same group of objects, and respectively, trapping the two sets of messages.

4.3.3 How Multi-Channels are Looked Up

We have to investigate about what univocally determines a multi-channel in order to understand how the multi-channels are looked up and entrusted to manage a remote method invocation.

A multi-channel, by definition, is established between two groups of objects: the objects belonging to the former group play the role of senders while the objects belonging to the latter group play the role of receivers. Looking into the main existing, both point-to-point and multi-point, communication statements, we noticed that such statements are asymmetric, i.e., each statement explicitly expresses

[2]We mean that the meta-programmer has to write code specially designed only for realizing a meta-behavior depending on the identity of the trapped method.

only which objects play the receiver role and not which are the objects invoking such a service[3]. Hence, we typically omit to specify the senders, this is compliant to the fact which each communication statement it is planned to be used only by sender at time, and that it can be determined by the context. However in different times a communication can involve more than one sender, e.g., in synchronization statement like `pvm_barrier`, but also in these cases the sender is determined from time to time by the context. Due to this behavior we state that the sender side dynamically changes.

Of course, each multi-channel is characterized by the communication semantics it defines, i.e., the kind its behavior realizes. A multi-channel is characterized by the set of referents playing the role of receivers. Because of the intrinsic dynamism of the sender side of a communication, only receivers are relevant to characterize a multi-channel, senders hook themselves on different moment and requests. Thus each multi-channel is represented by:

$$\textbf{multi-channel} \equiv \textbf{(kind, receiver}_i\textbf{, \ldots, receiver}_n\textbf{)}$$

Hence, in order to retrieve the right multi-channel each communication — in our context, each remote method invocation — has to specify both how the message it generates has to be managed (i.e., a kind), and, as usual, the designed destination of the messages. In this way through the logical identification of each multi-channel with a pair composed by its kind and the set of its referents on receiver side, it is possible, when starts the communication, to determine univocally towards which multi-channel the message has to be hijacked.

4.3.4 When Computations Switch Between Levels

Each reification is related to the communication channel established among the interacting objects, and it will take place every time a new channel is used, i.e., when a message, requiring a channel never used, is sent in the system. Each instantiated multi-channel is compliant with the requested features (its kind will satisfy the requested behavior and its referents will be the message receivers, and so on). Each message requiring an already used channel, is handled by the multi-channel which embodies that communication channel. When the multi-channel terminates its computation the execution control returns to the sender originating the meta-computation. On the basis of the implemented behavior, the multi-channel may also transfer the execution flow to one (or more) of its referents in order to execute some base-computations and achieve the expected result. Thus, when an object issues a request to another object, the exchanged message is transparently trapped and hijacked towards the multi-channel which reifies the communication channel it

[3] By example, in Java [9] we use the dot notation `o₁.methodName()` where o_1 is the receiver of the message and the sender is implicitly represented by `this`, analogously in C++PVM [43] we create a stream (an instance of `cppvmSendStream` class) connected to the receivers and each message reaches any receiver connected to the stream used.

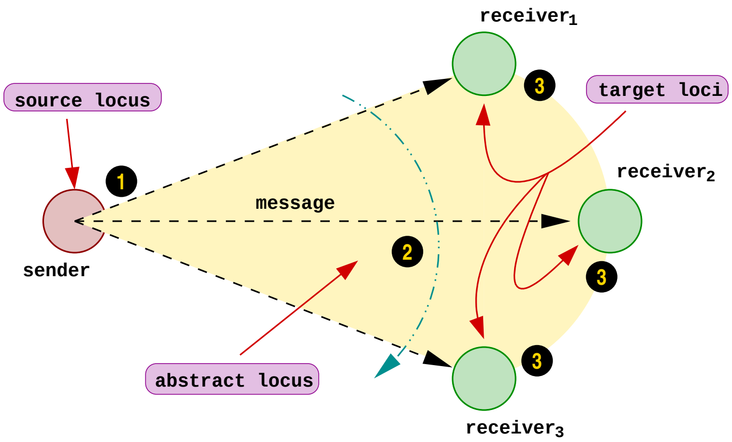

Figure 4.2: **Aspects and loci involved in a multi-communication**

has to use, then the multi-channel delivers the message, compliant to its behavior, to the designed receivers, and returns their results.

4.3.5 Communication Loci and Meta-Computations

As outlined in Fig. 4.2, a multi-point communication has a wide area of influence — the yellow cone in the picture — and involves several aspects of the system.

As explained in chapter 2 and in particular in session 2.2.1, each distributed communication involves many steps, e.g., message marshaling and unmarshaling, server localization, message dispatching, and so on. We term the overall of these steps as the *path of a message*, i.e., the set of steps each message has to carry out in order to be delivered. Each step involves a specific aspect involved in the communication, e.g., message marshaling takes place in the sender address space. We define as *communication locus* a point of the path of a message where are realized some important steps of the communication mechanism. The most relevant loci, we have identified (the numbers respect the situation depicted in Fig. 4.2) are:

❶ the sender and the beginning of the communication, termed *source locus*,

❷ the dispatching and the delivering of the message to the designed receivers, termed *abstract locus*, and

❸ the receivers and the carrying the computation really out, termed *target loci*.

These loci satisfy the property to be a partition of the overall path of a message. Moreover, the entities (hidden or not) which work in a locus are put in charge of managing the basic aspects which allow to a distributed communication to be correctly carried out, e.g., in the source locus messages are marshaled.

Each multi-channel incarnates these loci freeing the base-level from the communication mechanism moving it into the meta-level under the control of the meta-program. To allow meta-programs to customize the semantics of each communication, the multi-channel is able to perform a specific meta-computation parametrized on each locus substituting, hence, the locus standard behavior. Meta-computations performed on source and target loci guarantee that they are carried out on the site involved in the loci, e.g., on the sender site if the meta-program works on the source locus. This distinction derives from requirements of performance, reliability and availability. Working on the source and target loci is convenient for reducing communications, while working on the abstract locus allows to decentralize meta-computations and to improve the reliability and availability of the provided services.

4.4 Reflective Models

In [38], a first reflective model taxonomy has been pointed out. Ferber remarks the existence of two major reflective approaches: *meta-object* and *communication reification*.

The meta-object approach consists in linking each base-entity — also termed *referent* — with one or more meta-entities — also termed meta-objects — reifying it. The communication approach consists in reifying only the base-entities interactions into specific meta-entities.

4.4.1 Meta-Object Approaches

In the meta-object approaches the entities working in the meta-levels monitor and manipulate the entities working in the base-level, or more generally the entities working in a level monitor and manipulate the entities working in the underlying level. They can intercede and introspect the inner behavior of an entity, i.e., the intra-entity communications or the ingoing messages, but they don't consider the interactions that each entity has with other entities, i.e., the out-going messages. The models most representative of this approach are: the *meta-class model* [30], and the *meta-object model* [63, 51].

Meta-Class Model

A class describes both the structure and the behavior of its instances. The structure is the set of instance variables whose value will be associated within the instances. The behavior is the set of methods to which instances of the class will be able to respond [46].

The meta-class model [30, 18] is a variant of the meta-object approach, in which the reflective tower is realized by the instantiation link. The meta-object reifying a base-entity is its class, the meta-meta-object reifying a meta-object is its meta-class, and so on (see figure 4.3).

Classes fit perfectly the role of controller and modifier of structural information, because they keep such information hardwired in their nature. Their problem

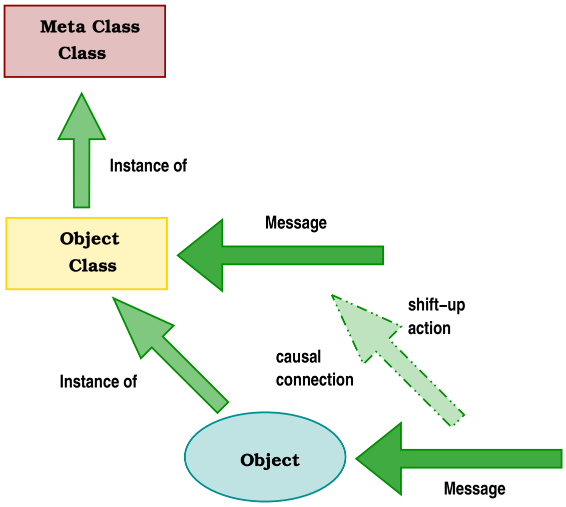

Figure 4.3: **Meta-Class Model Scheme**

is represented by the difficulty of specializing the meta-behavior of a single instance. Any instance of a class has the same meta-object; hence all instances share the same meta-behavior. To specialize the meta-behavior for each instance, it is possible to use the inheritance relation (building up a dummy derived class, differing from the original class only for the meta-behavior) or dictionaries keeping information and methods about/for each instance.

To dynamically change the behavior of an object it is necessary to substitute its meta-class, but not all languages allow the dynamic substitution of the class of an object; moreover changing the class of an object can lead to inconsistencies.

Of course, the meta-class model is directly implementable only in those languages handling classes as objects (e.g. SmallTalk, and CLOS) or simulating such a situation.

Meta-Object Model

The meta-object model [51] is a variation of the meta-object approach. But, instead of identifying the meta-object with the class of the base-entity, meta-objects are instances of a specific class *MetaObject* or of derived classes (see figure 4.4). The reflective tower is realized by the clientship relation. In this model, separate entities handle intercession and introspection on each base-entity. Each meta-object intercepts (shift-up action) the messages sent to its referent, and performs its meta-computation on such messages before actually delivering (shift-down action) them to its referent.

The meta-object model makes few assumptions about the relationships between base and meta-entities: in principle, each meta-object can be connected to many referents, and each referent can be linked to several meta-objects (one at a time) during its lifecycle. Usually, a meta-object is linked to an object through an

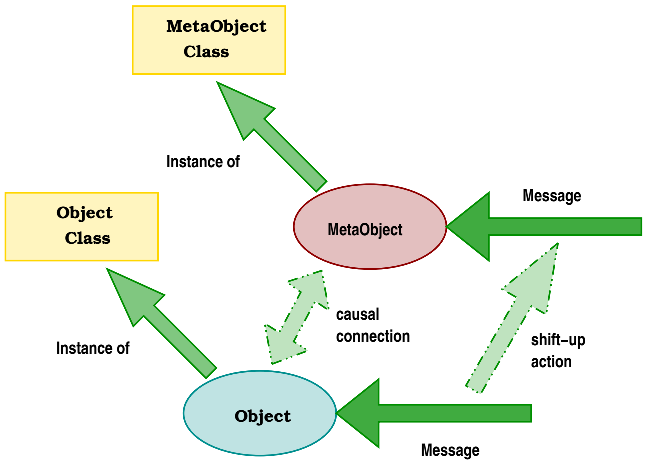

Figure 4.4: **Meta-Object Model Scheme**

instance variable, using SmallTalk parlance, or a C++ inherited field, so that is in order to change the meta-object it is possible to change the value associated to that slot.

However most implementations, for efficiency reasons, restrict this freedom: in CCEL [34], OpenC++ [26], Iguana [45] and ABCL-R [64] a meta-object is linked to one referent only, and each referent can have only one meta-object during its lifecycle.

It is simple to specialize the meta-behavior per object, developing a new class of meta-objects which refines the original one with the specialized meta-behavior. To specialize the meta-behavior per method or finer entities we need to develop the meta-object in case-style writing specific code that is able to check each possibility.

The major drawback of this model is that a meta-object can monitor a message only once it has been received by the referent. Thus the meta-object loses information about the sender and cannot perform actions related to the sender's identity.

The meta-object is the most used model, and its applications are not limited to programming languages, but they also involve operating system (e.g., AperTOS [96]), and graphic interfaces (e.g., Silica [74]).

4.4.2 Communications Reification Approaches

This kind of approach contrasting with the meta-object approaches. The meta-entities defined following this approach are bound to the interactions between the entities working in the underlying level, and unbound from such entities. The most representative model of this approach is the *message reification model* [38].

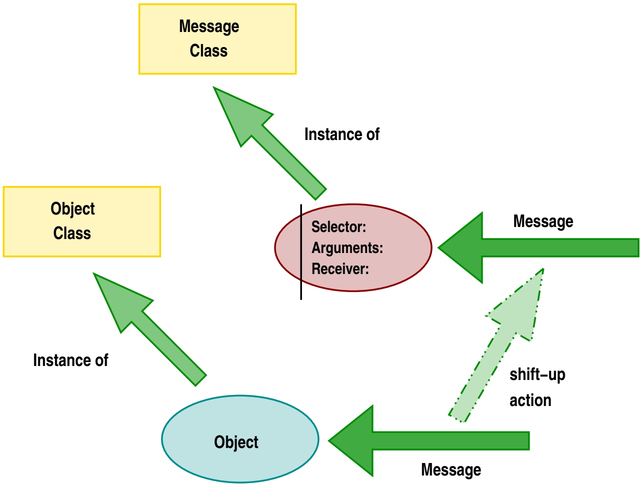

Figure 4.5: **Message Reification Model Scheme**

Message Reification Model

The message reification model [38] is a variant of the communication reification approach. In this model, meta-entities are special objects, called *messages*, which reify the actions that should be performed by the base-entities. The *kind* of a message defines the meta-behavior performed by the message; different messages may have different kinds. Every method call, is reified into an object — termed message — which is charged with its own management (e.g., delivery) in according to the kind of the meta-computation required, and when the meta-computation terminates, such a message is destroyed.

It is possible to define different behaviors for method calls performed by each object, specifying a different kind for each method call. Messages are not linked to the base-entity originating them and cannot access their structural information. The message object exists only for the duration of the action which it embodies. Thus it is impossible to store information among meta-computations (lack of information continuity). On the other hand, every method call creates and then destroys an object (the message). The approach as the drawback of introducing a big overhead due to the continuous creation and destruction of meta-entities. The reflective tower in this model consists only of two levels: the base and the meta-level.

4.5 Why We Need a New Reflective Model

We are interested in designing an architecture which renders possible to customize the remote method invocation mechanism, overcoming the open issues of the current reflective middlewares. In order to do that we need a reflective model which abstracts and supports the communication mechanism as a whole and encapsulated entity and allows to perform intercession and introspection about the exchanged messages.

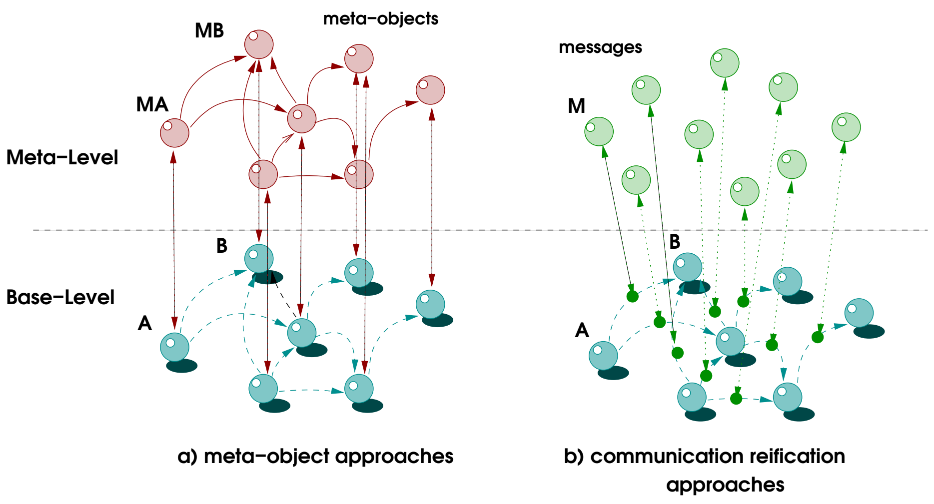

Figure 4.6: **Meta-Object vs Communication Reification**

Current reflective models only partially solve such issues, proving themselves inadequate to manage the customization of the communication mechanisms.

The models belonging to the meta-object approach (e.g., the meta-class and meta-object models) are too oriented to the base-entities, hence in order to customize a base-level communication the meta-programmer has to replicate such a communication in the meta-level emulating a communication reification approach. In other words, meta-objects may be used to monitor and to manipulate communication as well, but only by means of cooperative actions with other meta-objects. Figure 4.6.a shows a service request from referent `A` to referent `B`. The request is trapped by `A`'s meta-object (`MA`) and the reply is trapped by the server meta-object (`MB`). The two meta-objects have also to coordinate their actions before and after additional communication at meta-level.

On the other hands messages (or more in general the meta-entities of any model belonging to the communication reification approach) really embody the exchanged message, but they neglect some aspects of the communication. Messages are completely untied from the base-level entities involved in the communication (message's senders and receivers), this fact hinders the customization of the actions carried out in the loci directly tied to such entities (e.g., marshaling actions). Moreover, volatility characterizes messages, that is, messages are destroyed when the messages they embody are delivered, thus, it is a hard job to carry out customization based on previous communications. Figure 4.6.b shows how a base-level interaction is managed in the message reification approach: the service request is reified into message `M`, so only one meta-entity is responsible for that.

However, the main difference between meta-objects and communication reification approaches lies in their intended use, that is, a message reifies and monitors communications between two objects, while a meta-object (or a meta-class) controls the behavior of one specific object (or of a specific class), as shown in figure 4.6. As a side-effect of this difference, in the meta-object approach, in order to customize interactions among several base objects (and so the remote method invocation mechanism), we must reify as many meta-entities as there are base-entities. Such meta-entities interact via a communication graph which duplicates (or includes as subgraph) the communication graph at base-level (see figure 4.6.a). Instead, communication reification approaches result more suitable for customizing communication mechanisms, because they directly reify object interactions (instead of single objects) that don't need to communicate between themselves (see figure 4.6.b) encapsulating, hence, the adopted communication mechanism and simplifying its customization. At the meta-level, the communication graph, if any, is usually different from that at base-level.

From our analysis, current reflective models revealed not suitable for customizing the remote method invocation mechanism overcoming the troubles of the existing reflective middlewares. Then we need to design a new reflective model which merges the current approaches' features suitable for achieving our goals (e.g., meta-objects' persistence between meta-computation, and the kind concepts of the messages), and which introduces new features (e.g., the meta-entities' global view) in order to be able to completely customize the adopted communication mechanism.

4.6 Conclusions

This chapter introduce the reflective model, termed (multi-)channel reification model, especially designed for transparently extending and customizing the remote method invocation. Around the (multi-)channel reification model we built a usable framework, called mChaRM, which includes an enrichment of an existing object-oriented language (in the specific case an enrichment of Java), and an a simple API for helping the remote method invocation customization. In the following chapters we will describe the language enrichment, the APIs, how to use it and how we realized the framework.

— Chapter 5

mChaRM: a Framework Opening Up the Java RMI Mechanism

mChaRM is an acronym for multi-Channel Reification Model, and represents a programming framework for supporting the homonym model (see chapter 4). The framework consists in some language enrichment for opening up the RMI mechanism, and a set of APIs permitting the customization of the communications.

5.1 mChaRM: the Language Enrichment

In order to preserve transparency, typical reflective feature, we have slightly enriched the Java syntax introducing some new keywords which hide and support the key mechanisms introduced by the multi-channel reification model. We have chosen to extend Java language because it already has a remote method invocation, but the new keywords and concepts may be easily adapted to any other object-oriented language.

The new keywords can be grouped in extensions used in the program which will run in the base-level and in extensions aimed to help the meta-programmer to build new meta-entities (i.e., multi-channels).

5.1.1 Base-Level Language Enrichment

As stated in section 4.3, each communication statement has to provide information about both the communication channel (the receivers of the message) and the behavior it needs to be fulfilled (the kind of the communication channel they intend to use), in order to determine the right multi-channel to be used. To provide this information we have extended the base-level language with statements linking each method-call, and the receivers of such a call to the behavior that will be used for carrying such a call out. The introduction of such as information is coordinated by the new statement **kinds**.

kinds Statement

The **kinds** statement is used by the programmer to associate a *kind* and a set of receivers, in agreement with the characterization of multi-channel given in section 4.3, to a given method activation. Data provided by the **kinds** statement will be manipulated by the compiler|interpreter and translated into statements used to reify the related communication channel, to decouple the base-entities from the communication loci, and to route the method calls towards the right multi-channel. Of course, this translation is tied to the adopted multi-channel architecture, more on this will be explained in chapter 6. The **kinds** statement is described by the following syntax:

```
Kinds              ::=  kinds KindsList
KindsList          ::=  kind <kind-name> with ReceiverNameList to MethodNameList KindsList |
                        kind <kind-name> with ReceiverNameList to MethodNameList
ReceiverNameList   ::=  <receiver-name> , ReceiverNameList | <receiver-name>
MethodNameList     ::=  <method-name> , MethodNameList | <method-name>
```

where keywords are written in non-proportional font, and *<kind-name>*, *<receiver-name>*, and *<method-name>* strings representing, respectively, kinds, receivers, and methods identity.

The **kinds** statement has to be inserted in the class definition of each potential reflective sender. It identifies a section devoted to describe the interface of each instance towards the meta-level. A reflective sender, written in mChaRM, looks as:

```
class dummy
  kinds
    kind verbose with A, B, Z to dummy1
    kind normal with A, B, Z to dummy2, dummy3 {
      // usual class description
}
```

In the stub of code above we associate a multi-channel of kind ≪`verbose`≫ with the calls of the method `dummy`$_1$ carried out by instances of the class `dummy` to the receivers `A`, `B` and `Z`.

5.1.2 Meta-Level Language Enrichment

One of the targets we have set ourselves consists in supplying a mechanism which allows the meta-programmer to work on a reification of the communication as a whole and not on the reification of each single aspect involved in the communication. To consider the communication as embodied by a unique object, renders more natural to hide the intra-object communications[1] allowing the meta-programmer to focus his|her efforts on the enrichment of the original communication.

[1] We classify as intra-object communications, the communications necessary to carry the message from the sender to the receivers, passing through the meta-entities which embody the communication loci, i.e., the components of the multi-channel.

Channel Description

To this aim we have enriched the class definition of the Java language with a new section, introduced by the keyword **kind**, describing how the class the meta-programmer is writing has to be bring back to its components.

In our idea the meta-programmer will write only a class representing a multi-channels' kind. Besides, such a class, through its declaration section, instructs the system about which classes will represent the reification of its communication loci. In particular each multi-channel reifies and represents three communication loci (see section 4.3.5), hence the class representing a kind of multi-channels contains all the information needed to automatically build the classes of the instances embodying such loci. Substantially, the class written by the meta-programmer describes a multi-channel masking its real structure and the entities embodying its loci.

Each class, describing a multi-channels' kind, has a section in which it has to specify the name of the kind and, if necessary, its sub-components. The name of the kind is the only information mandatory. Data related to the classes whose instances embody abstract, source, and target loci are respectively introduced by the **core**, **senders**, and **receivers** keywords. Each keyword introduces an optional session which provides: the name of the class which represents the related locus, and also the name of the class it extends and the list of interfaces it implements. Parent classes and interfaces are respectively introduced by the keywords **enriches**, and **provides** instead of the Java keywords **extends**, and **implements** in order to avoid confusion with the standard clauses. Default meta-behaviors has been provided per each locus, thus each of these sections or part of them can be omitted.

This syntax enhancement is described by the grammar below, such a statement has to be put in each class which describes multi-channels juxtaposed after the class name.

ChannelComponents	::=	`kind` <*kind's name*> AbstractLocusSection SourceLocusSection TargetLocusSection
AbstractLocusSection	::=	`core` <*class used as abstract locus*> EnrichesSection | ε
SourceLocusSection	::=	`senders` <*class used as source locus*> EnrichesSection | ε
TargetLocusSection	::=	`receivers` <*class used as target locus*> EnrichesSection | ε
EnrichesSection	::=	`enriches` <*parent class name*> ProvidesSection | ε
ProvidesSection	::=	`provides` ProvidesList
ProvidesList	::=	<*interface name*> | <*interface name*>, ProvidesList

Note that, in the above grammar, all the elements between brackets describe strings.

Hence, when the meta-programmer writes a class describing a multi-channel, (s)he will have to adopt the just described statement to specify the multi-channel's components, and such a class will look like the following stub of code:

```
public class normal instantiates PREmCharM_MOP
  kind normal
  core example enriches channelCore
  senders exampleSenderStub enriches senderStub
  receivers exampleReceiverStub enriches receiverStub {
                 ...
}
```

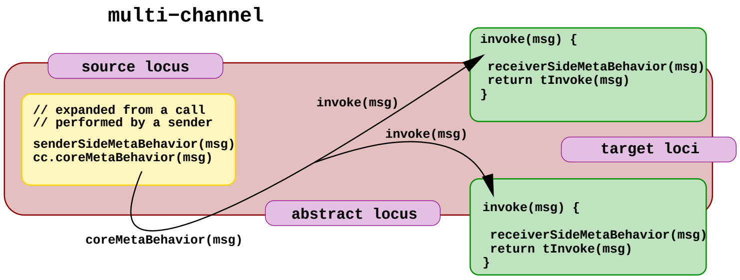

Figure 5.1: **Following a method call through the meta-level.**

Such a class describe a multi-channel whose kind is ≪`normal`≫. It also instructs the framework that its (sub-)components (i.e., the multi-channel's parts embodying respectively the source, abstract and target loci) have to be described by, respectively, the classes `exampleSenderStub`, `example`, and `exampleReceiverStub`. These classes are automatically generated from the multi-channel description (more about such a translation in chapter 6).

5.2 Meta-Behaviors, and Support APIs

We propose a simple API, sufficient for supporting all multi-channel uses. Our API does not claim to be exhaustive, we kept it simple for presentation purposes. Despite its simplicity our API is powerful enough to be used for implementing sophisticated communication mechanisms and for showing the capabilities of the multi-channel approach. The prototype, which can be downloaded from:

`http://www.disi.unige.it/person/CazzolaW/mChaRM_webpage.html`

implements a more extended API that could be widened in the next future.

Methods supplied by the API are classifiable, thanks to their purpose and to the involved aspect, into three categories: <u>multi-channel's meta-behavior</u>, <u>introspection and intercession on messages</u>, and <u>introspection and intercession on referents</u>. Methods belonging to the first category realize the behavior defined by the kind of the channel and cannot be directly used by the meta-programmer, i.e., they are automatically activated by the system, and don't have to be directly invoked in the meta-program. Methods belonging to the other two categories are predefined and cannot be altered, they are used both by the methods belonging to the first category and by the new multi-channel's methods.

In the sequel we will describe in details how to write the multi-channel's meta-behavior and the supplied API.

5.2.1 Multi-Channel's Meta-Behavior

Following the description of the multi-channel reification model, see chapter 4, the behavior of each multi-channel consists in actions performed on the message they manage, inside the communication loci it goes through. To this aim the system automatically performs a predefined method on a message when it goes through each locus. As depicted in figure 5.1, when the message goes through the source locus, the system applies the `senderSideMetaBehavior` method on it; when the message goes through the abstract locus the system applies the `coreMetaBehavior` method on it; and then when the message goes through the target locus the system applies the `receiverSideMetaBehavior` method on it.

Hence, to program a multi-channel means to write such methods which the multi-channel uses during the management of the trapped message. Methods belonging to this category implement the kind of multi-channel defining how multi-channels have to behave. The meta-programmer must override these methods in order to build new kinds of multi-channels which carry the meta-programmer's purposes out.

Meta-Behavior API
`Object coreMetaBehavior(mChaRMMethodCall msg)` *embodies the reflective behavior carried out by the multi-channel, when the message goes through the abstract locus.*
`void senderSideMetaBehavior(mChaRMMethodCall msg)` *embodies the meta-behavior carried out by the multi-channel, when the message goes through the source locus.*
`void receiverSideMetaBehavior(mChaRMMethodCall msg)` *embodies the meta-behavior carried out by the multi-channel, when the message goes through the target locus.*

Table 5.1: mChaRM's APIs for carrying meta-computation out.

Methods in table 5.1 define the meta-computation carried out by the multi-channel on the messages when they transit through each of the previously exposed loci. As said above, each of these methods work on the message passing through the multi-channel, thus, their arguments represent the exchanged message. Each message is represented, or using a more appropriate parlance reified by an instance of the class `mChaRMMethodCall`. This class encapsulates each piece of information related to the message (e.g., the sender, the method it represents, and so on), and can be manipulated through its methods (more about this class can be read in section 5.2.2).

The methods `senderSideMetaBehavior`, and `receiverSideMetaBehavior` are respectively performed on the source locus (i.e., when the message is trapped and hijacked to the meta-level) and on the target loci (i.e., it is activated every time that the meta-program calls the method `invoke`). They do not return values, however these methods allow the multi-channel to modify the exchanged message via side-effects performed on their actual arguments in agreement with the multi-channel behavior (see section 5.2.2 about actions that can be performed on messages). Their default behavior is empty, — i.e., they do nothing.

The method `coreMetaBehavior` implements the core of the multi-channel computation. It coordinates the result of the computations performed into the target loci and decides how to handle the received messages and to who demand their computation. As default behavior it forwards the messages to the specified receivers and returns the last value received to the caller.

```
public Object coreMetaBehavior(mChaRMMethodCall msg) throws MethodDoesNotExistException {
  Object[ ] result = new Object[(msg.receivers()).length];
  for(int i=0; i<(msg.receivers()).length; i++)
    result[i] = ((receiverStubInterface)receiverStub((msg.receivers())[i])).invoke(msg);
  return result[0];
}
```

Note that, we do not make supposition on where the `coreMetaBehavior` is performed, it should containing only statements whose execution is location-independent. For statements location-dependent the meta-programmer has to override either the `senderSideMetaBehavior` or the `receiverSideMetaBehavior` routines (or both).

5.2.2 Introspection and Intercession on Messages

Introspection and Intercession on Messages
sender and receivers identification
`String sender()` *returns the name of the sender which has sent the reified message.*
`String[] receivers()` *returns the names of the designed receivers of the reified message.*
introspection
`String getMethodName()` *retrieves the name of the called method.*
`boolean hasArgs()` *checks if the reified method call has some arguments.*
`Object[] actualArguments()` *returns the actual arguments of the reified method call.*
`Object inspectArgument(int position)` *retrieves the value of a specified actual argument.*
intercession
`String setMethodName(String methodName)` *through the name changes the method which will be really activated, and returns the original name of the called method.*
`Object modifyArgument(int position, Object newValue)` *changes the values of a specified actual argument, and returns the old value.*
`void insertArgument(int position, Object value)` *inserts a new argument in the call.*
`Object removeArgument(int position)` *removes from the call a specified argument.*

Table 5.2: A selection of the API provided by mChaRM for carrying introspection and intercession out on messages.

Most of the manipulation a meta-programmer could need to carry out on a communication are related to the exchanged message, thus the part of the API devoted to carry intercession and introspection out on messages dispatched through a multi-channel is quite detailed.

These methods, table 5.2, together with the methods reported in table 5.1, represent the core of the whole mechanism allowing the meta-program to alter the messages which pass through the multi-channel. The API supplies to the meta-programmer methods for looking into the contents of the actual parameters, for modifying their contents, for enriching the method call we are handling with new parameters, and so on.

We can classify, thanks to their behavior, the reflective actions on messages in three categories: *identification of sender and receivers*, *introspection*, and *intercession* on the reified message.

Identification of Sender and Receivers.

Each reified message is also characterized by the sender which has started the communication and by the designed receivers of the message. In the API, in order to query such information, we have introduced the methods:

```
String sender()
String[] receivers()
```

that respectively allow the meta-program to query for the name of the sender and of the designed receivers of the message. These methods combined with methods of section 5.2.3 permit to carry introspection and intercession out on messages' referents.

Introspection on Reified Message.

Introspective actions on the the reified messages consists in a set of methods which allow the meta-program to question the reified message about its characteristics (e.g., method name, contents of the actual arguments, and so on).

Each message is identified by the name of the called method, such a piece of information can be retrieved using the method:

```
String getMethodName()
```

the name of the called method is represented by a string.

Other basic information characterizing the reified message are their arguments, in order to look at them the API supplies two methods:

```
Object[] actualArguments()
Object inspectArgument(int position)
       throws ReferenceToNonExistentArgumentException
```

the former method retrieves all the arguments used in the call, while the latter inspects a single argument identified through its *position* in the reified call, if the specified position exceed the number of arguments passed to the call the method raises a `ReferenceToNonExistentArgumentException` exception.

Intercession on Reified Message.

To carry out intercession on the reified message means both to alter its identity and to modify its arguments both their contents and their number.

Regarding to modify the message's identity, the API permits to substitute the reified message, thanks to the fact that in Java a remotely invoked method is looked up via its name, changing its name.

```
String setMethodName(String methodName)
```

the `setMethodName` method doesn't check that the new name is a correct name for the designed receiver, i.e., if such a name correspond to a public method in the class of the designed receiver, such a check has to be explicitly performed when the method is really invoked.

Instead, regarding to alter the message actual parameters, the API supplies the method:

```
Object modifyArgument(int position, Object newValue)
       throws ReferenceToNonExistentArgumentException
```

which permit to substitute the contents of a single parameter with a new value. Moreover the API supplies a couple of dual methods:

```
void insertArgument(int position, Object value)
       throws ReferenceToNonExistentArgumentException
Object removeArgument(int position)
       throws ReferenceToNonExistentArgumentException
```

which permits to add and to remove arguments to the original method call. Both these methods identify the message's arguments through the position they have in the call, and raise a `ReferenceToNonExistentArgumentException` exception when the specified position fall outside the range of the possible position of the reified method call.

Such methods manage the messages as unrelated to the receivers' class, hence, they don't check the changes congruence in the designed receiver's interface. In this way, we can go around the type checking and manage situation, like information piggybacking, which foresee temporary situation unsound with the final result.

In chapter 7 we show how to use these methods to implement some communication protocols and to extend the communication semantics with new features.

Introspection on Senders and Receivers
Abstract Locus
`senderStubInterface senderStub(String name)` *accessor to a representative of the specified sender.*
`receiverStubInterface receiverStub(String name)` *accessor to a representative of the specified receiver.*
`Object retrieveReceiverFieldValue(String receiverName, String fieldName)` *retrieves the content of a field of the specified receiver.*
`Object retrieveSenderFieldValue(String senderName, String fieldName)` *retrieves the content of a field of the specified sender.*
Source and Target Loci
`Object referent()` *accessor to the referent of the stub.*
`Object retrieveField(String fieldName)` *queries for the contents of a specified field of the referent.*
`Object invoke(mChaRMMethodCall msg)` *invokes a specified method of the referent with the specified arguments.*

Table 5.3: A selection of the API provided by mChaRM for carrying introspection and intercession out on senders and receivers.

5.2.3 Introspection and Intercession on Senders and Receivers

The proposed model want to be a generic purpose reflective model, due to this reason the API also provides some methods for introspection and intercession of the multi-channel's referents, but is specially designed to support the enrichment of communication semantics, not for managing the base-object structure or semantics; thus, the part of the API devoted to intercession on multi-channel's referents is kept simple and consists of few methods, as shown in table 5.3. Besides, due to the fact that many multi-channels could be attached to the same referent, haphazard intercession might steer to an inconsistent state of the involved referents. Hence the current API allows a multi-channel to only look into the state of its referents and to invoke methods of its referents.

Such API's methods can be classified, based on where they are designed to be used, as: usable on the abstract locus, and usable on the other loci. However, in despite of where they are used their designed behavior is quite similar, thus

```
Object referent()
```

method permits to directly access to the aspect reified either in the source locus or in one of the target loci, and it can be used only in the reification of such loci, while

```
senderStubInterface senderStub(String name)
receiverStubInterface receiverStub(String name)
```

methods wrap the `referent()`, and are used in the abstract locus and permits to access to one of the multi-channel's referents, the one specified in the call through its name.

The API also supplies some methods for inspecting the state of the multi-channel's referents, i.e., to inspect the content of an attribute of a the base-level entities involved in the communication. Thus

```
Object retrieveField(String fieldName)
```

method permits to directly access to the state of aspect reified either in the source locus or in one of the target loci, and it can be used only in the reification of such loci, while

```
Object retrieveReceiverFieldValue(
        String receiverName, String fieldName)
Object retrieveSenderFieldValue(
        String senderName, String fieldName)
```

methods wrap the `retrieveField()`, and are used in the abstract locus and allows the meta-program running in the abstract locus to retrieve the content of the attributes of each base-level entity involved in the reified communication, respectively playing the role of receiver and sender.

The only exception to this behavior is represented by the

```
Object invoke(mChaRMMethodCall msg)
        throws MethodDoesNotExistException
```

method, which allows the meta-program running in the abstract locus to invoke public methods defined in the interface of the designed receiver. This method is the only way to carry intercession out on base-level entities, i.e., the API entrusts intercession to the methods defined in the base-level methods. This strategy limits the danger of inconsistent changes to the base-level state, because their commitment undergo to the base-level rules for changing.

5.3 Verbose Channel: a Simple Example

In this simple example we would like to show how to use the just exposed API in order to describe a kind of multi-channels and their behavior, i.e., the meta-behavior applied to the communications they will reify.

The class below describes a multi-channel of kind ≪`verbose`≫. A multi-channel of kind ≪`verbose`≫ is designed to output some notes while the message travels through the meta-level loci, i.e., the multi-channel's components reifying such loci notify, printing a string, when a message passes through the loci they are reifying (e.g., when the message leaves the sender, when it goes through the multi-channel computation and when it gets to the designed receivers).

public class verbose **instantiates** PREmCharM_MOP
  kind verbose

```
core verboseChannel enriches channelCore
senders verboseSenderStub enriches senderStub
receivers verboseReceiverStub enriches receiverStub {

  public Object coreMetaBehavior(mChaRMMethodCall msg) throws MethodDoesNotExistException {
    System.out.print("*** ");
    msg.printmChaRMMethodCall();
    return super.coreMetaBehavior(msg);
  }

  public void receiverSideMetaBehavior(mChaRMMethodCall msg) {
    System.out.println("*** I'm in the stub of the receiver ["+retrieveField("whoAmI")+"],");
    System.out.print("*** the call trapped is: "+msg.getMethodName()+"( ");
    if (msg.hasArgs())
      for(int i=0;i<(msg.actualArguments()).length;i++)
        System.out.print((msg.actualArguments())[i]+" ");
    System.out.println(").");
  }

  public void senderSideMetaBehavior(mChaRMMethodCall msg) {
    System.out.println("*** I'm inside senderSideMetaBehavior ["+retrieveField("whoAmI")+"],");
    System.out.print("*** I'm working for "+msg.getMethodName()+"( ");
    if (msg.hasArgs())
      for(int i=0;i<(msg.actualArguments()).length;i++)
        System.out.print((msg.actualArguments())[i]+" ");
    System.out.print(")\n*** when I exit I'll forward the call to: "+(msg.receivers())[0]);
    for(int i=1;i<(msg.receivers()).length;i++)
      System.out.print(", "+(msg.receivers())[i]);
    System.out.println(".");
  }

}
```

Such a class is basically written in Java. The only exception is represented by the **kind** section which fixes the dependences (a sort of inheritance relation) between this class and the previously written classes. It defines which classes the multi-channel's components we are describing want to inherit from.

We only describe the basic behavior the multi-channels have to carry out on each possible locus, i.e., in such a class we override the API's methods carrying out the meta-computations (`senderSideMetaBehavior`, `receiverSideMetaBehavior`, and `coreMetaBehavior`).

As we see more in details in chapter 6, our framework is based on OpenJava [91, 28]. The **instantiates** is part of the OpenJava language and defines which meta-object will preprocess (at compile-time) the current class. The class `PREmCharM_MOP` manages the extension we have introduced to the Java language expanding the logical multi-channel class in the classes which really compose it.

5.4 Conclusions

At the moment we have presented both the model and the language intended to be used to open up the remote method invocation mechanism, in the sequel we have to describe the framework architecture, or better the one we have chosen, adopted to support the multi-channel reification model and the preprocessor which translate mChaRM code to pure Java code. We also present some applications which illustrate how to use the framework and why you have to use it.

— Chapter 6

mChaRM: Framework Realization

In this chapter we describe the mChaRM working prototype which provides a programming and run-time framework to open up the Java RMI mechanism. We describe the architecture that we chose for the multi-channels and how it has been realized. We also briefly describe OpenJava, and the meta-objects we developed in order to translate the mChaRM code into pure Java, and how the translation takes place.

6.1 mChaRM: A Brief Framework Overview

The multi-channel reification approach and the language extension described in the previous chapters have been realized by the mChaRM framework. Its main goals consist in providing a development tool and a reflective middleware which opens up the Java RMI mechanism. mChaRM is developed in Java and consists of three aspects:

❶ a preprocessor which manages the extensions to the Java syntax,

❷ a Java package (`mChaRM.multichannel`) containing some basic classes from which develop new multi-channels, and

❸ a collection of multi-channels (package `mChaRM.mChaRMCollection`) implementing some kinds of multi-channels.

The preprocessor has been realized using OpenJava [28, 91]. We wrote two meta-objects (`PREmChaRM_MOP`, and `mChaRM_MOP`) which steer the OpenJava compiler (`ojc`) during the translation of the mChaRM code into Java. They respectively translate the **kinds** in the multi-channel's code, and in the base-level's code, adapting the generated code to support the reflective approach described. The compiling procedure of mChaRM source code looks as in Fig. 6.1.

As already explained the multi-channel approach enriches multi-point communications with new features. Java doesn't support multi-point communications, such as broadcast, multicast, or multi-rmi. Hence mChaRM tool adopts its own

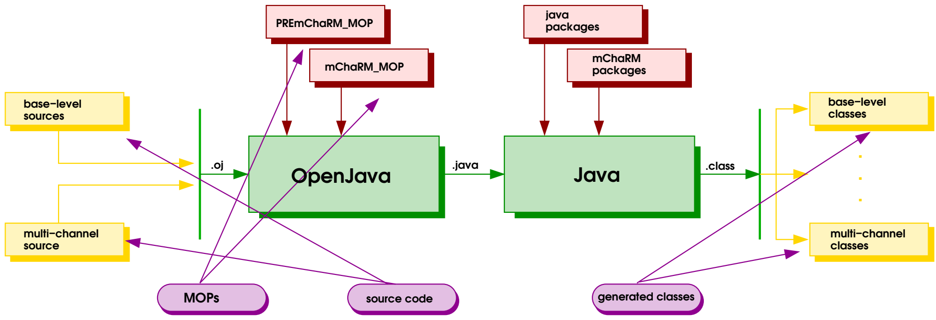

Figure 6.1: **mChaRM's Architecture**

multi-point communication mechanism. The current version uses a naive realization of multi-point RMI based on Java RMI [88], which broadcasts a given method call to a set of servers and then gathers the results and returns the first value received to the caller. A multi-point communication can be raised using the method `multiRMI`:

```
Object multiRMI(String methodName, String[] recsName, Object[] args)
```

At the moment this method is introduced in the client code by the mChaRM preprocessor and hides the reification|reflection mechanism provided by mChaRM, however we are already working on decoupling it from the mChaRM tool improving the transparency of the reflective mechanism. We have also kept its implementation simple because the main purpose of mChaRM consists of rendering the multi-communication protocol customizable by the (meta-)programmer. For exposition reasons we use the `multiRMI` also for the normal point-to-point RMI.

6.2 OpenJava

OpenJava [28, 91] is a compile-time MOP for Java that inherits most of the design philosophy of its direct ancestor OpenC++ Version 2 [26]. It can be seen as an "advanced macro processor" [28] that performs a source-to-source translation of a set of classes written in a possibly extended version of Java into a set of classes written in standard Java.

OpenJava produces an object representing a logical structure of class definition for each class in the source code. This object is called a class meta-object. A class meta-object also manages macro expansion related to the class it represents. Programmers customize the definition of the class meta-objects for describing macro expansion. In OpenJava, the meta-program of a macro is described as a meta-class. Macro expansion by OpenJava is divided into two: the first one is macro expansion of class declarations (callee-side), and the second one is that of expressions accessing classes (caller-side).

The programmer, through a declaration `instantiates M` put just after the class name, specify that the class meta-object for the class is an instance of the meta-class `M`. This meta-object controls macro expansion involved with the class the programmer is writing. The declaration of `M` is described in regular Java.

Every meta-class must inherit from the meta-class `OJClass`, which is a built-in class of OpenJava. The `translateDefinition()` is a method inherited from `OJClass`, which is invoked by the system to make macro expansion. If an `instantiates` clause in a class declaration is found, OpenJava creates an instance of the meta-class indicated by that `instantiates` clause, and assigns this instance to the class meta-object representing that declared class. Then OpenJava invokes `translateDefinition()` on the created class meta-object for macro expansion on the class declaration later. Since the `translateDefinition()` declared in `OJClass` does not perform any translation, a subclass of `OJClass` must override this method for the desired macro expansion. Modifications are reflected on the source program at the final stage of the macro processing.

As a class is represented by a class meta-objects, a member method is also represented by a method meta-objects. In OpenJava, classes, member methods, member fields, and constructors are represented by instances of the class `OJClass`, `OJMethod`, `OJField`, and `OJConstructor`, respectively.

Macro expansion in OpenJava is managed by meta-objects corresponding to each class (type), this translation is said to be type-driven. Callee-side translation of class declarations is lead by `translateDefinition()` of the meta-objects associated to such class declarations, through an `instantiates` clause.

In addition to the callee-side translation, `OJClass` provides a framework to translate the code related to the corresponding class spread over whole program selectively (caller-side translation). The parts related to a certain class is, for example, instance creation expressions or field access expressions.

Translation Mechanism

Given a source program, the processor of OpenJava:

❶ analyzes the source program to generate a class meta-object for each class;

❷ invokes the member methods of class meta-objects to perform macro expansion;

❸ generates the regular Java source program reflecting the modification made by the class meta-objects, and

❹ executes the regular Java compiler to generate the corresponding byte code.

Those methods of `OJClass` whose name start from `expand` performs caller-side translation, and they affect expressions in source program declaring another class `C`. Such expressions may also be translated by `translateDefinition()` of the class meta-object of `C` as callee-side translation. Thus different class meta-objects affect the same part of source program. To solve such an ambiguity, callee side translation is always applied before caller side translation.

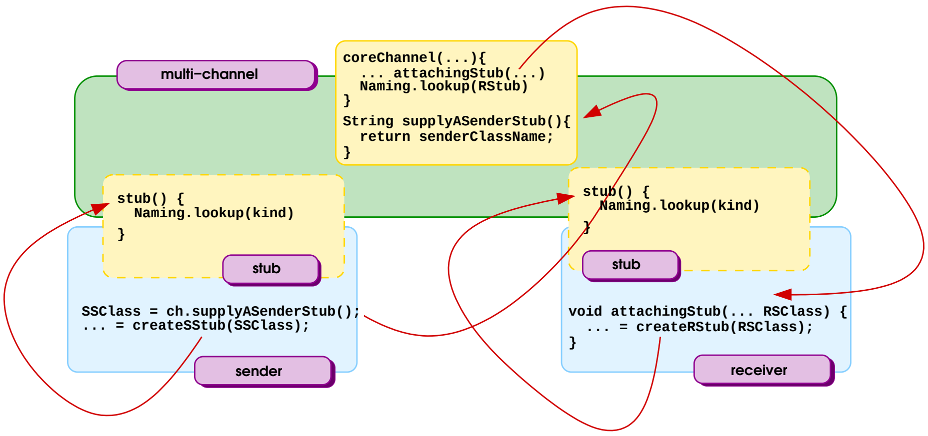

Figure 6.2: **How the mChaRM framework establishes a multi-channel among its referents.**

Syntax Extension

With OpenJava macros, a meta-class can introduce new class|member modifiers and clauses starting with the special word at some limited positions of the regular Java grammar. The newly introduced clauses are valid only in the parts related to instances of the meta-class. In a class declaration (callee-side), the positions allowed to introduce new clauses are:

- before the block of member declarations,
- before the block of method body in each method declaration,
- after the field variable in each field declaration.

Whereas in other class declarations (caller-side), the only allowed position is after the name of the class.

The member method `getDeclSuffix()` is invoked by the system when needed, and returns a `SyntaxRule` object representing the syntax grammar beginning with the given special word. An instance of the class `SyntaxRule` implements a recursive descendant parser of LL(k), and analyzes a given token series to generate an appropriate abstract syntax tree. The system uses `SyntaxRule` objects obtained by invoking `getDeclSuffix()` to complete the parsing.

6.3 Multi-Channel's Chosen Architecture

Both to achieve a good ratio among performance, transparency, reliability, and availability and to comply with the model requirements, we have chosen to design and implement multi-channels as composed of several entities, i.e., a central kernel (termed *core*) and as many stubs as its referents.

Each stub is part of a multi-channel and it is strictly coupled to one of its base-level referents. It runs on the same site and in the same addressing space of such a base-object. It is designed to interface that object to the multi-channel core, and vice versa. A sender stub transparently traps each call appointed to the multi-channel, which it represents. Sender stubs reify, and reflect on, source loci, receiver stubs reify, and reflect on, target loci, while the core encapsulates the abstract locus functionality. Receiver stubs' main goal consists in applying the `invokes` issued by the core and appointed to their receivers. Many stubs can be attached to each base-level object, a stub for each multi-channel connected to that object, they will be recognized by the kind of their multi-channel. Stubs also carry out the part of the multi-channel's meta-behavior related to source and target loci. The related stub classes also define the corresponding API methods (see table 5.1), i.e., `senderSideMetaBehavior`, and `receiverSideMetaBehavior`.

Both sender and receiver stubs attach themselves dynamically to the multi-channel's core. Figure 6.2 sketches the algorithm for bootstrapping a multi-channel and for connecting together its components. Following its characterization (see the multi-channel reification model definition in chapter 4), the multi-channel knows which of its referents are playing the role of receiver. Hence, during its initialization the core notifies, calling the method `attachingStub`, to its receivers of creating its receiver stub, then it will be attached to the core.

The **kinds** declaration in the sender class, provides to the preprocessor all information about which multi-channels the sender could use during its lifecycle. The preprocessor expands such a declaration into the code which asks a stub to the core of the multi-channel (a remote method invocation of the core method `supplyASenderStub`) and which really creates the stub. In order to minimize the amount of exchanged data the core of the multi-channel only supplies the name of the stub's class to senders and receivers. Then they use the methods offered by the Java core reflection library [87] and such a piece of information to really create the requested stub.

Each stub can be viewed as an extension of its own referent, it communicates with its referent via a local method call, avoiding problems of communication reliability, and network unreachability. It looks as if sending and receiving action was moved to the meta-level layer where they are played with a new semantics, the one described by the multi-channel. In this way every communication algorithm can be developed without bothering the system functional behavior.

The multi-channel core does not reside in a specific location with respect to its referents, it is independent from its execution site which usually depends on the implemented behavior (e.g., if it implements a fault tolerant behavior it will be located on a faultless machine or replicated on several machines). It communicates with the stubs via remote method calls. All messages exchanged among multi-channel components are encapsulated in the API supplied to the meta-programmer.

This architecture guarantees a complete encapsulation of each aspect and locus involved by the communication. Thus, every communication protocol can be replaced by a new one encoded into a multi-channel. Each message communication is handled as follows[1]:

❶ the sender stub traps each method call appointed to a multi-channel of its kind;

❷ the sender stub carries out the `senderSideMetaBehavior` method, then

❸ it calls the `coreMetaBehavior` of the core which it is attached to (this call is a remote call);

❹ after its computation, which can involve zero or more `invoke`s, the core returns the result to the sender stub

❺ the sender stub returns the result to its referent, as the required call should have done.

Note that `invoke` calls the `receiverSideMetaBehavior`, before really applying the passed message.

Our framework provides some basic classes: `senderStub`, `receiverStub`, and `channelCore` (package `mChaRM.multichannel`). Such classes represent basic components from which it is possible to derive other stub's and core's classes and therefore new kinds of multi-channels. In the sequel we examine their implementation.

Stubs

Sender and receiver stubs inherit by a common class `stub` which keeps information about multi-channel identity (i.e., the core identity, the stub name, the name of its referent, and so on), and methods for handling such information. Most of the task is carried out by the constructors of the stubs. In both cases, stubs communicate with the core through the Java RMI mechanism. They play the server role because they will receive messages by the core (e.g., for retrieving the value of its referent fields, or for activating the `invoke`). Sender stubs also play the client role because their main behavior consists in asking the core of the multi-channel to perform its part of the meta-computation. As shown by the following code both kinds of stubs register themselves as servers and sender stubs also require an interface in order to communicate with the core of the multi-channel.

```
public senderStub(Object myReferent, String myKind, String myReferentName) {
    setReferent(myReferent); setWhoIsMyCore(myKind);
    Naming.rebind("__"+myReferentName+myKind, this);
    WhoIsMyCore().senderStubHasBeenCreated(myReferentName);
}

public receiverStub(Object myReferent, String myKind, String myReferentName) {
    setReferent(myReferent); setWhoIsMyCore(myKind);
    Naming.rebind(myKind+"__"+myReferentName, this);
}
```

[1]This Algorithm fills the gaps left in Fig. 5.1 and in the description given in the previous section explaining how the multi-channel hooks to the base-level.

The main task performed by a sender stub consists in forwarding the (reworking of the) original call to the core of the multi-channel. The caller implicitly invokes the `stubBehavior` method of its sender stub which is part of the designed multi-channel. `stubBehavior` performs a call to `senderSideMetaBehavior` and then it passes the reworked information about the original call to the core.

```
final public Object stubBehavior(mChaRMMethodCall msg) {
  senderSideMetaBehavior(msg);
  return WhoIsMyCore().coreMetaBehavior(msg);
}
```

Both `stubBehavior` and `senderSideMetaBehavior` cannot be called directly from the meta-program. The `senderSideMetaBehavior` can be overridden in descendant classes of class `senderStub` in order to build new kind of multi-channels.

Instead the main service offered by each receiver stub is the method `invoke`. Its behavior consists in forcing the receiver to activate the message filtered by the multi-channel. Implicitly `invoke` carries out the `receiverSideMetaBehavior`, and then it calls `tInvoke` which really activates the reworked message.

```
final public Object invoke(mChaRMMethodCall msg) {
  receiverSideMetaBehavior(msg);
  return tInvoke(msg);
}
```

The method `tInvoke` uses the Java core reflection library [87]. The algorithm looks into the method table of the receiver for a method matching the received message and, if such a method is found then it is activated with the passed arguments. Finally, it returns to the caller the achieved result. Method `tInvoke` cannot be overridden nor directly activated from the code of the meta-program.

```
final private Class[ ] extractArgsClasses(Object[ ] args) {
  if (args == null) return null;
  Class[ ] c = new Class[args.length];
  for (int i=0; i<args.length; i++) c[i] = args[i].getClass();
  return c;
}

final protected Object tInvoke(mChaRMMethodCall msg) {
  Class[ ] c = extractArgsClasses(msg.actualArguments());
  Method method = (referent().getClass()).getDeclaredMethod(msg.getMethodName(), c);
  return method.invoke(referent(), msg.actualArguments());
}
```

Core

The core of the multi-channel is described by a class named: `channelCore`. The class constructor requires the kind of the multi-channel, a string identifying the behavior that its instances realize, the identifiers of its referents (following the model only the receivers can characterize the channel), and the identifier of the stub classes it uses. The constructor hooks the just created instance with the stubs of the specified referents, as well as to initialize the multi-channel structures.

```
public channelCore(String kind, String[] RsName, String SSClassName, String RSClassName) {
  receivers = new Hashtable(); senders = new Hashtable();
  setReceiverStubClassName(RSClassName); setSenderStubClassName(SSClassName);
  setRsName(RsName); setKind(kind); sendersName = new Vector(10, 1);
  serverName = "__"+kind()+"{ ";
  for (int i=0; i<howManyReceivers(); i++) serverName += receiverName(i) + " ";
  serverName += "}";
  Naming.rebind(serverName, this);
  reflectiveReceiverInterface RSI;
  for (int i=0; i<howManyReceivers(); i++) {
    RSI = (reflectiveReceiverInterface)Naming.lookup(receiverName(i));
    RSI.attachingStub(serverName, receiverStubClassName(), receiverName(i));
    setReceiverStub(receiverName(i), (receiverStubInterface)Naming.lookup(serverName+"__"+receiverName(i)));
  }
}
```

The multi-channel's behavior is determined by the method `coreMetaBehavior`. A new kind of multi-channels can be created by writing a subclass of `channelCore` which overrides such a method. The default behavior supplied by `coreMetaBehavior` in the framework, simply, consists in forwarding the trapped message to the specified receiver stubs for its activation, and in dispatching the result of the computation of the last receiver back to the caller. The multi-channel provided as default by the framework realizes a context switch between base- and meta-level and vice versa, and a multi-point RMI as really expected by using the `multiRMI` without enhancing its behavior. By the way it is a good starting point for deriving new communication behaviors (as explained).

```
public Object coreMetaBehavior(mChaRMMethodCall msg) {
  Object[] result = new Object[(msg.receivers()).length];
  for(int i=0; i<(msg.receivers()).length; i++)
    result[i] = ((receiverStubInterface)receiverStub((msg.receivers())[i])).invoke(msg);
  return result[0];
}
```

6.4 MOP to Handle the kinds Statement

As shown in Fig. 6.1 the code written in mChaRM is translated into standard Java through a preprocessing phase lead by an OpenJava meta-object instance of the class: `mChaRM_MOP`. This meta-object is charged both of expanding the new keyword, and of adding all the necessary stuff for supporting the approach (the method `multiRMI`, the hooks for the stubs and so on). As shown by the code below, this meta-object, during the parsing of the source code, will add information for keeping trace of the object stubs, and the code used for creating and hooking the stubs. It has also to expand the **kinds** to the code for routing the messages through the right multi-channel.

```
public void translateDefinition() throws MOPException, CannotAlterException {
  ParseTree suffix = this.getSuffix("kinds");
  if (suffix != null) {
    addField(createAFieldForAnHashTableOfStubs("HashTableForSStub"));
    addCodeForSendersStuff(suffix);
    addMethodMultiRMI();
```

```
  }
  addField(createAFieldForAnHashTableOfStubs("HashTableForRStub"));
  addMethodAttachingStub();
  addInterface(OJClass.forName("mChaRM.multichannel.reflectiveReceiverInterface"));
}
```

When the meta-object, parsing the base-level code, detects the keyword **kinds** it knows that instances of such a class will play also the role of sender. Each sender has to be able to implicitly choose and retrieve the right multi-channel. Hence, the meta-object will add to the class which is parsing a static method, `initialize`, which fills an hashtable (`__kinds`), indexed on the messages and its potential receivers, with the kind of the multi-channels which have to be used for handling such messages. Information kept by such a hashtable are used by the method `multiRMI` for picking the right multi-channel up. The `initialize` method is executed during the object initialization phase.

```
private void addCodeForSendersStuff(ParseTree pt) {
  String initializeCode ="";
  OJMethod initialize = new OJMethod(this, OJModifier.forModifier(PRIVATE+STATIC),
                          OJClass.forName("void"), "__initialize", null, null, null, null);
    // add a static hashtable for handling the dynamic binding with channels
  TypeName tn = new TypeName("java.util.Hashtable");
  VariableInitializer vi = new AllocationExpression(tn, new ExpressionList());
  FieldDeclaration fd = new FieldDeclaration(new ModifierList(PRIVATE+STATIC), tn, "__kinds", vi);
  addField(new OJField(getEnvironment(), this, fd));

  Enumeration e = ((openjava.ptree.List)pt).elements();
  while (e.hasMoreElements()) {
    // parsing of each row of the section introduced by kinds
    // building a mapping receivers-method and multi-channel's kind
    // used for routing a message through the designed multi-channel.
                    ...

  }
  initialize.setBody(makeStatementList(initialize.getEnvironment(), initializeCode));
  addMethod(initialize);              // initialize represents the map built above
                    ...
}
```

When the meta-objects is parsing a potential sender class, it also adds the method `multiRMI` to such a class. The method `multiRMI` has to deliver the message it takes to the designed receivers. It starts verifying in the previously introduced maps (`__kinds`) if a multi-channel has been foreseen to reify such a message. If yes, it demands the delivery of the message to the multi-channel activating the related stub which will starts the meta-computation.

```
private void addMethodMultiRMI() throws CannotAlterException, MOPException {
 OJMethod MRMI = new OJMethod(this, OJModifier.forModifier(PRIVATE), OJClass.forName("Object"),
        "multiRMI", new OJClass[]{OJClass.forName("String"), OJClass.forName("String[]"),
        OJClass.forName("Object[]")}, new String[]{"methodName", "RsName", "args"}, null, null);
 String MRMIcode = "String servers = \"{ \";";
    // builds the body of the method multiRMI and stores it into MRMIcode
                    ...
 MRMIcode += "return null;}";
 MRMI.setBody(makeStatementList(MRMI.getEnvironment(), MRMIcode));
 addMethod(MRMI);
```

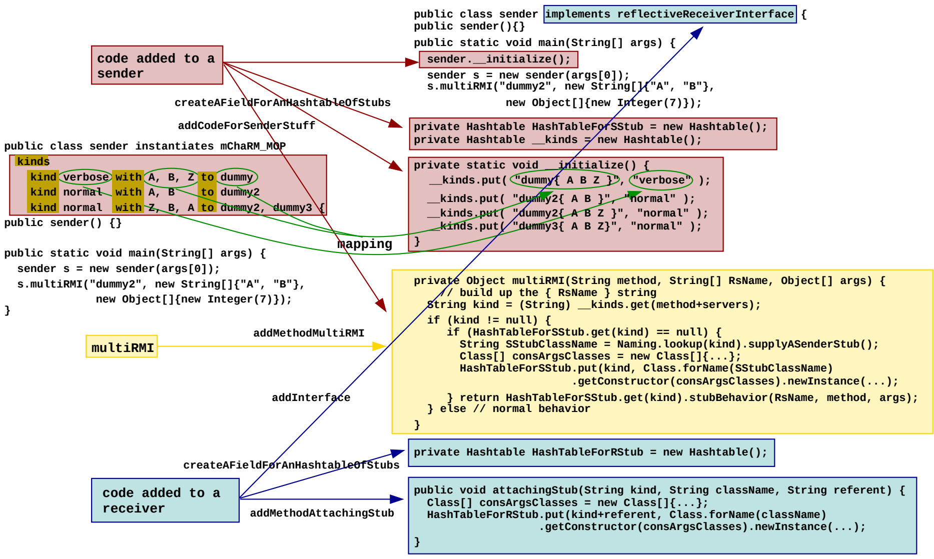

Figure 6.3: **A trivial example of translation carried out by the preprocessor**

}

Moreover, the preprocessor will add all the stuff for managing the case in which the instances of such a class will play the role of receiver. It basically adds a method which, when triggered, hooks a receiver stub instantiated from a designed class to the current object. As outlined in Fig. 6.2 this method will be triggered by the core of the multi-channel.

```
private void addMethodAttachingStub() throws MOPException, CannotAlterException {
  String ASCode; OJMethod AS;
  OJClass stringClass = OJClass.forName("java.lang.String");
  AS = new OJMethod(this, OJModifier.forModifier(PUBLIC), OJClass.forName("void"),
         "attachingStub", new OJClass[]{stringClass, stringClass, stringClass},
         new String[]{"kind", "className", "referentName"}, null, null);
     // puts in ASCode the body of a method which when called will attach
     // a receiver stub to the current object
  AS.setBody(makeStatementList(AS.getEnvironment(), ASCode));
  addMethod(AS);
}
```

6.4.1 How the Keyword kinds Is Translated

Figure 6.3 shows how the **kinds** declaration is translated by the just described OpenJava's meta-object.

The class in figure is very trivial, but it illustrates well how a class written using mChaRM is transformed in a class written in standard Java. The code originally written by the programmer is depicted on the left side of the picture, whereas on the right side of the picture there is its translation.

As described, the preprocessor adds specific code for senders and specific code for receivers. In the figure, we emphasize such a difference using red rectangles for code specially added to a sender, and cyan rectangles for code added to both senders and receivers. Green arrows emphasize how the mapping between the source code which specifies when and which channel to use and the generated code which really realizes such a binding.

We can also note on the left side of the picture, in the yellow rectangle, the added `multiRMI` method.

6.5 MOP to Handle Multi-Channel Description

As previously explained, every kind of multi-channels is described by a single class description, rather than to be described in terms of its components. By the way a multi-channel is intrinsically composed by several objects (see section 6.3). The MOP `PREmChaRM_MOP` steers such decomposition.

The meta-programmer writes a single class describing the multi-channel. Such a description contains all data related to the classes that the MOP has to generate. The **core**, **senders**, and **receivers** clauses introduce information (i.e., the class name, the name of the interface it implements, and the name of its parent class) about the classes which respectively describe the multi-channel's core, sender stubs, and receiver stubs.

Our MOP with respect to the multi-channel description has to create three classes and fill them in with attributes and methods related to the corresponding locus they describe. The most interesting and delicate part of the MOP is represented by the algorithm which redistributes fields and methods of the multi-channel description in the generated classes.

In order to determine the right membership of each attribute and method, the MOP will build a *dependency graph* of methods and attributes. We are sure of the membership of three specific methods, i.e., `senderSideMetaBehavior`, `receiverSideMetaBehavior`, and `coreMetaBehavior`. With such a piece of information is possible to build a dependency graph analyzing the code of these methods. Hence, methods and attributes used in the code of `senderSideMetaBehavior` and so on for their closure belong to the dependency graph related to the sender stub. All methods and attributes belonging to such a dependency graph will be part of the class of the sender stub. Such approach has some limits:

- in the multi-channel class we cannot put methods and attributes not used, because we are not able to determine where they have to be added, besides

- we duplicate code when a method or an attribute belong to two dependency graph, with obvious troubles.

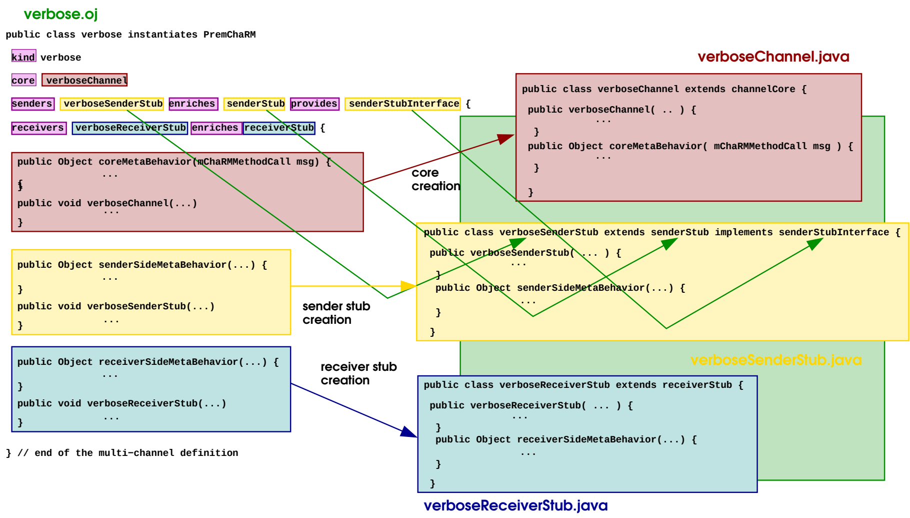

Figure 6.4: **How from a multi-channel description its components are generated.**

By the way, these troubles can be easily overcome with some extra coding, and fixing some programming rules. In next release, we want to introduce some keywords, such as **abstract-locus**, **source-locus** and **target-locus**, which will help in recognizing where to put each element.

In general, the MOP carries out the following algorithms:

❶ it analyzes the added syntax determining the data about classes it has to create;

❷ it creates the three `OJObjects` representing classes it has to compile;

❸ it builds the three dependency graphs using the algorithm just described;

❹ it fills the representative of the three classes;

❺ it writes such classes on files;

❻ finally, it creates a dummy class used by OpenJava to carry out the creation of the `.class` file from the translated code.

6.5.1 How Multi-Channels are Translated

Figure 6.4 shows how the description of the ≪`verbose`≫ multi-channel (section 5.3) is used to generate the Java code of its components.

The meta-programmer writes the code depicted on the left side of the picture, whereas on the right side of the picture there are the classes generated by the `PREmChaRM_MOP` MOP.

core, **senders**, and **receivers** clauses are used to originate new classes representing the multi-channel's loci. Each clause, implicitly or explicitly, provides their class name, the name of their parents, and the name of the interface they implement. In the picture, such information are emphasized using rectangles of the same color of the rectangles adopted to emphasize the code they contribute to generate, i.e., red for the core, yellow for the sender stub, and cyan for the receiver stub. Green arrows explain the use of such information in the case of the **senders** clause.

6.6 Implementation Choices and Conclusions

As concluding remarks of this section we would like to argue about some implementation choices and about some potential enhancement to the current framework implementation.

Java and OpenJava

We had to choose which programming language to adopt, when we decided to implement our ideas. We considered several object-oriented languages such as C++, SmallTalk, and Java. We chose Java for the following reasons:

- the Java environment provides a built-in RMI mechanism, whereas the other languages rely on ancillary frameworks and libraries;

- Java provides APIs (the Java core reflection class library) for carrying out meta-computations on the structure of its programs (structural reflection); that is the basic mechanism to manage methods as first class citizens in Java, and to realize the causal connection relation with little efforts;

- Java popularity in the last few years is grown a lot, hence adopting Java as programming language is a good way to render our work highly topical.

Starting from the choice of Java, it has been natural to adopt OpenJava as the tool for developing the mChaRM preprocessor.

Proxy vs Preprocessing

To support our model we chose to extend the Java language instead of adopting a proxy approach like in Jedi [3]. The absence of a preprocessor is the main advantage of proxies with respect to preprocessing. The code developed by the programmer would has been system independent still keeping its flexibility, but we would have lost in transparency. The programmer should explicitly use proxies, we could keep hidden the meaning of proxies, but the shift up action still remains intertwined to the rest of the application code. Of course, preprocessing adds a step in the development, and often the generated code can have useless and redundant stub of code which ad hoc solutions wouldn't have taken in consideration.

Possible enhancements to the current implementation of the mChaRM framework are related to the API it provides and to the meta-level languages. The MOP related to the channel syntax (see section 6.5) is quite experimental. Our MOP doesn't translate several specific Java features such as inner classes, anonymous class declarations, and so on. At the current stage of the work we preferred to focus our attention on supporting our ideas, rather than on providing a MOP that translates our syntax in conjunction with every Java features. An improvement currently under development is represented by the separation of the multi-rmi statement from the mChaRM framework designing a specific class which our framework will use.

— Chapter 7

Applications

In this chapter we show two working examples which explain how the mChaRM framework can be used to adapt the Java RMI mechanism improving its flexibility. Such example can be seen both as programming examples using mChaRM and as killer examples of situation which cannot or are not so easy to be managed with the classical approaches to middlewares.

7.1 SecurityChannel: Filter Outgoing Messages

Security implies not only protection from external intrusions but also controlling the actions of internally-executing entities and the operations of the whole software system. In this case, the interleaving between operations and data protection may become very complicated and often intractable. For this reason, security must be specified and designed in a system from its early design steps [39].

From another point of view

- it is very important that the security mechanisms of the application be correct and stable;
- the security code should not be mixed with the application code, otherwise it should be very hard to reuse well-proven implementations of the security model.

If this is not done, when a new secure application is developed the designer|implementer wastes time to re-implement and to test the security modules of the application. Moreover, security is related to: “who is allowed to do what, where and when”; so security is not functionally part of the solution of the application problem, but an added feature defining constraints on object interactions [86, 50]. From this last remark we can think of security as a feature operating at a different computational level and we can separate its implementation from the application implementation.

Thus, it is possible to exploit some typical reflection features, like *separation of concerns* and *transparency*, to split a secure system into two levels: at the first level there are (distributed) objects cooperating to solve the system application; at the second one, rights and authorizations for such entities are identified, specified and mapped onto reflective entities which transparently monitor the objects of the first level and authorize the allowed access to other objects, services, or information.

Reflection allows the separation of nonfunctional from functional code, organizing the access control policy in a specific meta-level connected to the rest of the application by the causal connection relation. In general, access control is enforced at the object communication level; for this reason communications-oriented reflective models (such as the multi-channel reification approach) are more suitable to model them than meta-object oriented models [21]. The main advantages of using a communications-oriented reflective approach is represented by the fact that a malicious request cannot reach the server, because it is trapped by the meta-entity, which validates the request.

Working in this way it is possible to develop stable and reliable entities for handling security. It is also possible to reuse them during system development, thus reducing development time and costs, and increasing application level assurance.

In the last few years a lot of work has been done in this direction [84, 75, 93, 92], but only in [6, 8, 7] has been emphasized that this kind of security check is a typical nonfunctional requirements bound to the communications. In fact it consists in intercepting and filtering the ingoing messages before that they really reach the designed receivers.

In this section we use the mChaRM framework to enrich the Java remote method invocation with a security check. We will show the multi-channel for a generic approach [6, 8] to access control trouble, and a more specific one implementing a history-based protocol for filtering the ingoing messages [7].

7.1.1 Multi-Channel Approach To Security

One of the possible communication extensions consists in validating if a message can be delivered or not. We can consider each message exchanged to a sender for a receiver as a service request by the sender to the receiver. Thus multi-channel established between two communicating objects, plays the judge role, verifying if the sender object has the permission for requesting or less that service. As stated in [8] associating the verification phase to the communication whereas to the receiver, hinders malicious attacks. The code reassembles the ATM example in [8], with customers and ATMs and customers can deposit in and withdraw from ATMs.

```
public class client instantiates mChaRM_MOP
  kinds
    kind validation with ATM1 to withdraw {

  public static void main(String[] args) {
    client c = new client("Walter", "#2588");
    ATM1 = (ATMInterface)Naming.lookup("ATM1");
    ATM1.deposit(c, 1000);
    ATM1.withdraw(c, 500);
  }
}
```

```
public class ATM instantiates mChaRM_MOP {

  public void deposit(client id, int sum) {
    // deposits «sum» on «id»'s account
  }

  public void withdraw(client id, int sum) {
    // withdraws «sum» from «id»'s account
  }
}
```

This trivial example is interesting because it shows how it is possible to use the kind mechanism in order to monitor only some requests. In our example a deposit can be done by everyone, whereas a withdrawal can be performed only by the account owner. Hence a multi-channel, which filters only the `withdraw` method, is enough for achieving the desired behavior. Such a behavior is imposed signing, in the `client` class, that its invocation has to be managed by a multi-channel of kind ≪`validation`≫, i.e., as specified in the **kinds** section of the `client` class above.

The structure of multi-channels which filter out-going requests on a matrix access basis could look as the class below. The basic idea consists in establishing a multi-channel among a server providing some services and its potential client. As can be guessed from its code, the multi-channel simply initializes itself loading the information related to the constraints on the services of its receiver. Then, it traps each message forwarded to its receiver, verifies if the request can be or cannot be carried out. Whenever a request passes such a check and is accepted is delivered to the designed receiver for the management.

```
public class ValidationChannel instantiates PremChaRM
  kind validation
  core validationChannelCore
    enriches channelCore provides channelInterface {
  private Hashtable right;

  public ValidationChannel(String kind, String[] RsName, String SSClassName, String RSClassName) {
    super(kind,RsName,SSClassName,RSClassName);
      // initializes the right table
  }

  public Object coreMetaBehavior(mChaRMMethodCall msg) {
    Integer permission = right.get(msg.inspectArgument(1));
    if (permission != null)
      if (permission > (Integer)msg.inspectArgument(2)) // yes! I've enough money
        return super.coreMetaBehavior(msg);
    System.out.printl("*** withdrawal cannot be allowed");
  }
}
```

This simple example only shows the how is possible to use our framework in order to filter messages before they will be really delivered to the designed receiver. Instead, next example will display a more complicated security policy which involve meta-level communications, i.e., communication among multi-channels.

7.1.2 A History-Based Security Policy

Modeling a history-based control policy [17, 77] using the multi-channel reification model is based on associating a multi-channel to each communication between two objects. Multi-channels trap each request before it reaches the called object. Once the action is trapped, the multi-channel looks up the system constraints and the history in order to validate the request. When the request is validated, the multi-channel either signals to the caller that the service is forbidden or lets the callee perform the request.

Supply on Request

By an example, we consider the ATM-Bank system described in the previous section. In such a scenario we have the constraint that: *only bank account holders can withdraw from the bank ATM*. This fact can be expressed as: the client can obtain money only if is an account holder in that bank and asks to draw money from the ATM, i.e., more formally:

$$\circ_p \langle \lambda x.x = \mathsf{Client} \blacktriangleright \mathsf{ATM}[\,\mathsf{drawing}(\,\mathsf{sum}\,)\,] \wedge \mathsf{isAccountHolder}\,(\,\mathsf{Bank},\,\mathsf{Client}\,)\,\rangle$$

$$\Rightarrow \mathsf{ATM} \blacktriangleright \mathsf{Bank}[\,\mathsf{drawing}(\,\mathsf{Client},\,\mathsf{sum}\,)\,]$$

for all clients, and amounts. Of course we also have to check if his account has money before allowing the drawing; in this example we do not consider this aspect because it does not add new elements to the explanation.

This constraint involves several entities in the base-level: a client, an ATM, and a bank. The relevant part of the history consists in the list of drawing requests carried out by such a client from the bank's ATM. The evaluation of such a constraint will involve at least two multi-channels: the one established between the ATM and the bank, and the one established between the client and such an ATM. Then, such an evaluation, whenever the withdrawal request will be allowed, will take place following the sequence diagram of Fig. 7.1.

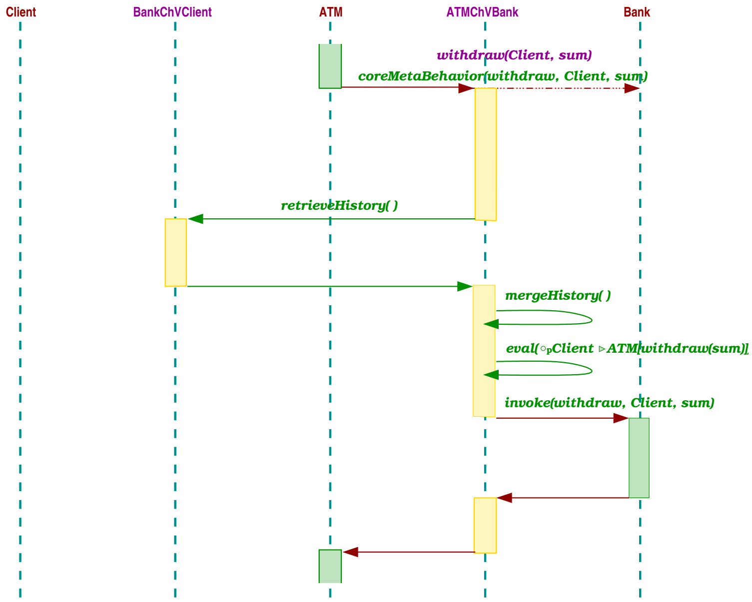

Figure 7.1: **Sequence diagram for the validation mechanism**

The history is built starting from allowed requests, and each multi-channel contributes to its construction with their set of allowed requests. We can keep system history in two ways: centralized or decentralized. In a large distributed system it is hard and inefficient to keep the history centralized, because it is necessary to charge an entity (*history holder*) with this task. The history holder has to communicate with all multi-channels in order to collect all history fragments and in order to supply them histories, that they will use to evaluate the constraints before allowing a service to be accessed. This behavior increases the overhead due to the objects' communication, but it guarantees history consistence. History decentralization is achieved by charging each multi-channel with the task of holding the part of history managed by it. In this way the bottleneck represented by a centralized history holder is avoided, but when a multi-channel has to evaluate a history-based constraint, it has to complete the information it holds with the part of history, necessary in the evaluation, held by other multi-channels, so there are communications only when needed and rarely with the same entity.

Implementation In a similar scenario, we need only multi-channels of kind ≪`history-based validation`≫ in order to realize the described history-based access validation policy. Such multi-channels will be established between two objects and they will check if any request exchanged between their referents can be allowed. Two phases can be identified in the execution of a multi-channel of kind ≪`history-based validation`≫:

❶ *initialization phase*
at multi-channel bootstrap, the multi-channel loads the constraints related to the services offered by its referent (the one playing the role of server), merging together the constraints related to a unique request[1];

❷ *validation phase*
each multi-channel spends most of its lifecycle waiting for a request and validating it; each request is trapped by a multi-channel when is going to leave the sender (shift-up action), and it is processed by the `coreMetaBehavior` method which handles the constraints validation related to such a message;

The main part of the multi-channel behavior is performed on the abstract locus by the `coreMetaBehavior`. Such a method performs per each filtered message the following steps:

- the first step consists of reassembling the historical environment in which the constraint will be evaluated:
  - it consults its constraint database (built during the initialization phase) looking for the constraints related to the request it must validate;

[1] A service can be allowed only if all its constraints are satisfied, and if $(\Gamma_1 \Rightarrow \delta) \wedge (\Gamma_2 \Rightarrow \delta) \equiv \Gamma_1 \wedge \Gamma_2 \Rightarrow \delta$ we substitute each group of constraints related to a unique request with a formula which joins all the premises

- once it retrieved the constraints to evaluate, it scans them in order to determine the previously requests because it must know if they are allowed or not;
- from the previously retrieved information it goes back to those multi-channels that have the missing part of the history, needed for the evaluation;
- it asks the involved multi-channels the missing part of the history, and it reassembles all histories into a unique history;

• when it has rebuilt the historical environment it begins the real constraints' evaluation;

• when it has evaluated the constraints, if the request must be rejected, the multi-channel notifies that to the client, otherwise it stores the request in the history as *allowed*.

The multi-channel class realizing the described (or similar) history-based access control policy could be outlined by the code below. As in the general case most of the operations are carried out in the abstract locus, i.e., by the redefinition of the `coreMetaBehavior` method.

```
public class History-Based_Security instantiates PremChaRM
  kind history-based_validation
  core HBValidationChannelCore
    enriches channelCore provides HBVChannelInterface {

  public History-Based_Security(String kind, String[] RsName, String SSClassName, String RSClassName) {
    super(kind, RsName, SSClassName, RSClassName);
  }

  public Object coreMetaBehavior(mChaRMMethodCall msg) {
    bintree constraint = retrieveConstraint(msg); // retrieves the related constraints
    interface_time rTime = (interface_time)lookupEveryWhere.lookup("tm");
    int actime = rTime.updateTime();
    if (eval(constraint, actime, msg.args())) {
          // Yes! the service can be carried out, I store it as allowed at the time ≪actime≫
      request r = new request(msg.sender(), (msg.receivers()), msg.getMethodName());
      elhistory el = new elhistory(r, actime);
      updateHistory(el);
      return super.coreMetaBehavior(msg);
    } else { // unfortunately, the service cannot be supplied
      System.out.println("It isn't possible to satisfy the request!");
      System.out.println("The security constraints are not respected!!");
      return null;
    }
  }

  public boolean eval(bintree tree, int t, Object[] par) {
    // evaluates the constraint ≪tree≫ at time ≪t≫
                                    ...
  }

  public bintree retrieveConstraint(mChaRMMethodCall msg) {
    // retrieve the constraint related to the reified message
                                    ...
```

```
  }

  public partialHistory retrieveHistory() {
    // on demand supplies its part of the history to another channel
                              ...
  }
}
```

Of course the complete class is more complex than such a stub of code. The complete class includes a lot of methods like `eval`, `retrieveConstraint`, and `retrieveHistory`, strictly coupled to the specific problem and not related to its realization using multi-channels approach. We omit the details about such a methods because not relevant to the explanation.

```
public interface channelInterface {
  Object coreMetaBehavior(mChaRMMethodCall msg) throws MethodDoesNotExistException;
}

public interface HBVChannelInterface extends channelInterface {
  partialHistory retrieveHistory() throws java.rmi.RemoteException;
}
```

A significant part of the proposed solution is represented by the fact that the history-based access control policy foresees that the system's history is reified by several multi-channels rather than by only a meta-entities. In this case multi-channels have to communicate in order to correctly evaluate a contraint. Multi-channels are just like classical objects. They provide operations and reply to interfaces. Thus, in order to communicate, as usual, they have to implement a new interface which defines the methods remotely invoked by the other multi-channels, e.g., in our example, such multi-channels will implement the interface `HBVChannelInterface` showed above, which defines the method `retrieveHistory`. The newly defined interface has to extend the predefined interface `channelInterface` which is implemented by any kind of user-defined multi-channels.

7.2 RMPChannel: Change Communication Semantics

As explained, to transparently extend the communication protocols with sophisticate and complex algorithms represents the main task for which the multi-channel reification approach has been designed. As an example of this features we describe the code of a multi-channel realizing the *reliable multicast protocol* [67, 95].

7.2.1 Reliable Multicast Protocol

The Reliable Multicast Protocol (RMP) provides a totally ordered, reliable, atomic multicast service on top of an unreliable multicast datagram service. RMP is a reliable multicast transport protocol offering a variety of services of different quality. The basic RMP protocol provides what can be thought of as N-way virtual circuits, called groups, between sets of processes connected by a multicast medium. It is fully distributed, so that all processes play the same role in communication.

While primarily using *negative acknowledgments* (nACKs) for error detection and retransmission, RMP provides true reliability and limits the necessary buffer space by passing a token around the members of a group. Each sender can sequence their packets separately, or RMP can sequence all the packets sent to group. RMP offers also a feature called total ordering, which means that RMP makes sure that every member of the group has received the packets before passing them to application.

RMP does not require a special multicast server or group manager. It is based on a token rotating scheme and a mix of negative and positive acknowledgements. The token rotates in the whole recipient group and the token owner acknowledges every packet it has received while it holds the token. The token passes and acknowledges are sent usually as multicasts. When other group members discover that they are missing some packets, they can send negative acknowledges to the original sender.

7.2.2 Basic Delivery Algorithm

The basic algorithm primarily uses nACKs for reliable delivery, and uses a single ACK per packet or set of packets to provide total ordering and stability of packets. In RMP, all data packets are stamped with a tuple which uniquely identifies each data packet. Data packets are multicast to the members of the group and are handled by a primary receiver called the token site. When the token site receives one or more data packets, it multicasts a *positive acknowledgment* (pACK) out to the members of the group. Each ACK contains zero or more identifying tuples for Data packets, along with a global sequence number, called a timestamp, which serializes all of the ACK, Data, and New List packets in a group. For a given ACK, the ACK is given the value of the timestamp it contains, and each data packet ordered by the ACK is given a consecutive timestamp.

Each ACK performs a number of functions:

- It lets the sender know that the current token site has received the packet. In this way it functions as a traditional positive acknowledgment to the sender.
- The timestamps in the ACKs provide a total and causal ordering on messages.
- The timestamps also provide a global basis for the detection of dropped packets. The receivers can detect any missed packets through these global sequence numbers.

To detect when a message has been received by all members of the group, each ACK also passes the token site to the next process in the group. Before it can accept the token, each member is required to have all of the packets with timestamps less than that of the ACK naming it the token site. If it doesn't have some packets, it requests them before accepting the token.

Ordering of packets, detection of missing packets, and buffering of packets for retransmission is all handled with the use of two lists, termed *DataList*, and *OrderingQ*.

When a message is received, it is placed into the DataList. When an acknowledge is received, it is placed in the OrderingQ, creating one or more slots on the end of the queue if necessary. Each packet occupies exactly one slot in the OrderingQ.

Whenever an acknowledge message is received, the OrderingQ is scanned through once to match up Data packets in the DataList with empty slots that have been created by an ACK. When a slot is found that has the same identifying tuple as a Data packet in the DataList, the packet is moved from the DataList to that slot. When holes occur in the OrderingQ, nACK packets are sent out, requesting retransmission of these packets.

A site is not allowed to accept the token until there are no empty slots in the OrderingQ up to the last data packet ordered by the ACK naming this site as the new token site.

In addition to bounding the buffer space needed in a ring, passing the token guarantees that site failures and dropped messages are detected within N[2] messages.

7.2.3 RMP Multi-Channel Description

The RMP algorithm is shattered among the multi-channel components. There are two relevant typology of components involved in the RMP algorithm: the sender of the message and its receivers. Of course some changes to the presented algorithm have to be done. The presented algorithm is specially designed for message passing environments whereas the mChaRM framework supports the client|server model. Thus, instead of considering messages and message passing in the classical sense we will adopt methods, and method calls.

The receiver stubs, implementd by the class `RMPReceiverStub`, will play the role, as described in the previous section, played by designed receivers of the message, i.e., the processes belonging to the group. They will acknowledge each other the receipt of a message, and will wait for the acknowledges from every of their siblings, before really carrying the message out.

Each message is a method, so the described acknowledging is carried out calling two methods: `pACK` for positive acknowledges, and `nACK` for negative acknoledges. In the RMP algorithm, the positive acknowledges are intended to the designed receivers of the broadcast message, so the method `pACK` is a method of the receiver stubs. Whereas the negative acknowledges are intended to the sender of the broadcast message, so the method `nACK` is a method of the core of the multi-channel, because, as we see in the sequel, the core plays the role of the sender of the broadcast message.

[2]Where N is the number of site in the group.

The class `RMPReceiverStub` which implements the RMP protocol on the target loci looks like the following stub of code:

```
public class RMPReceiverStub extends receiverStub implements RMPReceiverStubInterface {

  public String pACK (int n) throws RemoteException {
    synchronized (otherAcks) {
      if (n+1 > otherAcks.size()) otherAcks.setSize(n+1);
      if (otherAcks.elementAt(n) == null) otherAcks.setElementAt(new Integer(1), n);
      else otherAcks.setElementAt(new Integer(((Integer)otherAcks.get(n)).intValue()+1), n);
    }
    if (!myAcks.get(n)) new sendNAck(1000, n).start();
    return new String("Ok");
  }

  private void ackTheMsg (int n, String[] receivers) throws NotBoundException {
    for (int i = 0; i < receivers.length; i++)
      if (receivers[i].compareTo(whoAmI) != 0)
        new sendPAck(1000, n, Naming.lookup(serverName + "__" + receivers[i])).start();
  }

  public void receiverSideMetaBehavior(mChaRMMethodCall msg) {
    boolean newMsg = false;
    int msgNumber = ((Integer)msg.removeArgument(0)).intValue();
    String[] receivers = msg.receivers();
    synchronized (otherAcks) {
      if (msgNumber+1 > otherAcks.size()) otherAcks.setSize(msgNumber+1);
    }
    synchronized (myAcks) {
      if (!myAcks.get(msgNumber)) {
        newMsg = true;
        myAcks.set(msgNumber);
      }
    }
    if (newMsg)
      ackTheMsg(msgNumber, receivers);
    else return;
    for (int i = 0; i <= msgNumber; i++)
      while ( (otherAcks.elementAt(i) == null) || !myAcks.get(i) ||
          (((Integer)otherAcks.elementAt(i)).intValue() < receivers.length - 1));
  }
}
```

In order to avoid deadlocks, due to circular waiting of acknowledge messages, the acknowledging phase is entrusted to a different thread.

Such threads are managed by instances of the classes `sendNAck`, and `sendPAck`, which, respectively, send a negative acknowledge to the multi-channel's core asking to rentransmit a missing message, and a positive acknowledge to the other receiver stubs belonging to the same multi-channel, i.e., whose referents belong to the same group.

Acknowledges are automatically sent by the standard method of the Java class `Thread`, named `run`, which has been overrode in the described classes, showed below:

```
  private class sendNAck extends Thread {
   int delay, msgNumber;
   sendNAck (int delay, int msgNumber) {
```

```
      this.delay = delay;
      this.msgNumber = msgNumber;
    }

    public void run () {
      while (true)
        if ((cc.nACK(msgNumber, whoAmI)).compareTo("Ok") != 0) this.sleep(delay);
        else break;
    }
  }

  private class sendPAck extends Thread {
    int delay, msgNumber;
    RMPReceiverStubInterface rec;

    sendPAck(int delay, int msgNumber, RMPReceiverStubInterface rec) {
      this.delay = delay;
      this.msgNumber = msgNumber;
      this.rec = rec;
    }

    public void run () {
      while (true) {
        if ((rec.pACK(msgNumber)).compareTo("Ok") != 0) this.sleep(delay);
        else break;
    }
  }
```

Whereas, the multi-channel core replaces the real sender in that role, it ignites the message propagation and when a stub loses a message the core provides to send the missing message again to the interested stub.

The multi-channel's core also manages the returned results, deciding by majority, which is the right value to be returned to the sender, i.e., the method `voting` applies the majority algorithm on the returned values. We don't explicitely discuss and show such a method, because it is irrelevant for the explanation.

The class `RMPChannelCore` describes how the core of the multi-channel might look like:

```
public class RMPChannelCore extends channelCore implements RMPChannelInterface {

  public RMPChannelCore(String[ ] RsName) throws ReceiverStubNotFoundException {
    super("RMP", RsName, "mChaRM.multichannel.senderStub", "RMPReceiverStub");
  }

  public String nACK (int n, String name) throws MethodDoesNotExistException {
    ((RMPReceiverStubInterface)receiverStub(name)).invoke((mChaRMMethodCall)messages.elementAt(n));
    return new String("Ok");
  }

  private synchronized int newMsg () {
    messages.setSize(++msgNumber+1);
    return msgNumber;
  }

  public Object coreMetaBehavior(mChaRMMethodCall msg) throws MethodDoesNotExistException {
    int myMsg;
    sendMsg[ ] t = new sendMsg[(msg.receivers()).length];
    pair[ ] keys = new pair[(msg.receivers()).length];
```

```
    myMsg = newMsg();
    msg.insertArgument(0, new Integer(myMsg));
    messages.setElementAt(msg, myMsg);
    for (int i=0; i<(msg.receivers()).length i++) {
      keys[i] = new pair(myMsg, i);
      t[i] = new sendMsg((RMPReceiverStubInterface)receiverStub((msg.receivers())[i]), msg, keys[i]);
      t[i].start();
    }
    for (int i=0; i<(msg.receivers()).length; i++) t[i].join();
    return voting(keys);
  }
}
```

The multi-channel's core has to broadcast the message to the designed receivers, i.e., to the members of the group. In order to avoid deadlocks in the RMP algorithm the multi-channel's core is multi-threaded. Each thread is responsible for the message delivery to one of the designed receivers.

Each instance of the class `sendMsg` realizes a thread of the multi-channel's core and their method `run` dispatches the message to the designed receivers.

```
  private class sendMsg extends Thread {
    private RMPReceiverStubInterface rec;
    private mChaRMMethodCall msg;
    private pair key;

    public sendMsg(RMPReceiverStubInterface rec, mChaRMMethodCall msg, pair key) {
      this.rec = rec;
      this.msg = msg;
      this.key = key;
    }

    public void run() {
      Object res = rec.invoke(msg);
      synchronized (result) {
        result.put(key, res);
      }
    }
  }
```

Due to the multi-threaded nature of this multi-channel, all the shared data structures have to be synchronized and the access to them has to be rendered mutually exclusive to the multi-channel's threads, in order to avoid data corruption and race conditions.

Hence we have implemented in a simple way a tight collaboration among core and receivers stubs, and we synchronize the final message execution. This example shows several potentiality of the proposed approach, e.g., how manipulate the messages — each message is piggybacked with a number in order to distinguish them —, how the multi-channel components cooperate, how to reflect on ingoing messages, and so on. This example well characterizes many problematic tied to reflect on ingoing messages, and that are hard to be handled by the traditional reflective approaches.

7.3 Conclusions

In this chapter, we showed only some of the possible applications of a reflective approach which opens up the RMI mechanism. Several others applications can be developed, and some of them have already been developed and inserted in the distributed framework available at:

`http://www.disi.unige.it/person/CazzolaW/mChaRM_webpage.html`

or will be developed in the next future.

Of course, all feasible and interesting examples are related to communications and to extend communications with extra behavior. Hence, we could consider to develop multi-channels which checkpoint each exchanged message, and rollback the system to a stable state when the system or part of the system crashes. Another possibility consists in realizing transactional behavior through multi-channels, i.e., each multi-channel represents a transaction and manages the service commitment, communications among multi-channels permit to manage nested transactions. Multi-channels could also be used to implement load-balancing, performance tuning, auditing and many other communication oriented policies.

— Chapter 8

Conclusions and Future Works

In this work we explored the middleware field looking for limits and for the motivation of such limits. Our research detected three main troubles related to the current middlewares:

- interaction policies are hardwired into the middleware limiting the middleware adaptability (lacking of flexibility);
- distributed algorithms are intertwined to the applicative code breaking the separation of concerns requirement (lacking of separation of concerns);
- algorithms originally designed as a whole, have to be scattered among several entities and no one of these entities directly knows the whole algorithm (lacking of global view).

All these drawbacks impact on the software development hindering the reuse of software modules which implement features strictly related to communications such as message checkpointing, control access policies and so on.

First two troubles (lacking of flexibility, and separation of concerns) have already been faced and partially solved by the new branch of the reflection research area, termed reflective middleware. Reflection can be considered as the natural solution to improve flexibility and separation of concerns. Reflective middlewares are middleware systems that provide inspection and adaptation of their behavior through an appropriate causally connected self representation. A lot of reflective middlewares have been developed in the last few years, they usually provide well adaptable communication framework, which separate quite well the communications management from the rest of the application.

Unfortunately, they continue to fail in considering each remote invocation in terms of the entity involved in the communication (i.e., the client, the server, the message and so on) and not as a single entity. Hence, the global view requirement is not achieved. This is due to the fact that most of the meta-models that have been presented so far and used to design the existing reflective middlewares are oriented to monitor objects and not their interactions. In these models, every object is associated to a meta-object, which traps the messages sent to the object and implements the behavior of that invocation. Such a meta-models inherit the global view lacking from the object-oriented methodology which encapsulates the computation orthogonally to the communication.

Once we determined because the current middlewares are inadequate to satisfy the incoming requirements, the first attained result is represented by the design of a new reflective model, named *multi-channel reification model*, which considers communications as first-class citizens, reifying them into meta-entities and overcoming the lacking of global view while keeping all the typical benefits of reflective approaches.

To fulfill this commitment we designed a new model, called *multi-channel reification model*. The multi-channel reification model is based on the idea of considering a method call as a message sent through a logical channel established among a set of objects requiring a service, and a set of objects providing such a service. This logical channel is reified into a logical object called *multi-channel*, which monitors message exchange and enriches the underlying communication semantics with new features used for the performed communication.

In our ideas, a similar approach provides a basic tool for developing a reflective middleware which overcomes all the detected drawbacks. In fact, another attained result is represented by the mChaRM reflective middleware. mChaRM is a framework based on the multi-channel reification model, which opens up the Java RMI mechanism. Basic characteristics of mChaRM are:

- each aspect of a remote method invocation can be adapted through a meta-program;
- remote communications are reified into specific meta-entities which guarantee the global view requirement;
- the adaptation granularity is at method invocation level, i.e., it is possible to specify a different behavior for every method invocation;
- remote method invocation can be easily and transparently extended with new features;
- inter-object communications are hidden to the base-level programmer.

Hence, we analyzed middlewares and reflective middlewares in general and applied our ideas to open up a specific kind of middleware, middlewares based on RMI, and specially the Java RMI mechanism. However, the proposed solution and its basic ideas are so general that it is possible to smoothly apply them to any other kind of middleware (e.g., based on message passing).

In the next future we are planning to explore two different paths: move the exposed ideas from distributed to wireless environment, and use the mChaRM to explore the benefits that some applications, such as fault tolerance and security, can achieve from the approach.

Wireless and distributed systems are strictly related and several typical problems of distributed systems are put forward in wireless systems again. For example reliability, reachability, object migration, synchronization and so on. Moreover, some completely new problems have been introduced for wireless, such as location awareness, smart caching and code mobility. We think that a communication oriented reflective approach like the one presented in this work could be a good solution to face in a clean and smart way such problems, and we would like to verify the soundness of this statement.

On the other hands, we are also interested in exploring benefits which our multi-channel approach could bring to software development, in particular to the development of stand alone meta-levels implementing properties, basic to, but separable from the other entities' behavior, e.g., synchronization, or access control policies. A similar achievement, with the development of a specific methodology could provide, as explained in [23, 24], a tool for developing, testing, and reusing software systems which cross-fertilize orthogonally the rest of the system.

— Appendix A

mChaRM Code

A.1 Basic Multi-Channel Components

A.1.1 Multi-Channel Core

```
package mChaRM.multichannel;

import java.io.*;
import java.rmi.*;
import java.lang.*;
import java.rmi.server.UnicastRemoteObject;
import java.util.Hashtable;
import java.util.Arrays;
import java.util.Vector;
import java.net.MalformedURLException;
import mChaRM.RMI.*;

 /* channelCore

  This class defines the multi-channels' core default behaviour. Any kind of multi-channel can be
  defined extending this class. The key issue consists in overriding the coreMetaBehavior method.

  Remember each multi-channel is identified by its kind, ie., the reflective behaviour it realizes, and by
  the set of objects – playing the role of receivers – which it is connected to.

  */

public class channelCore extends UnicastRemoteObject implements channelInterface {
 /*
  String representing the reflective behavior (kind) realized by multi-channels
  instanciated from this class.
  Remember each multi-channel is identified by its kind and by the set of
  objects – playing the role of receivers – which it is connected to.
  */
 private String kind;
 /*
  Strings representing the objects which the multi-channel is connected to.
  Remember each multi-channel is identified by its kind and by the set of
  objects – playing the role of receivers – which it is connected to.
  */
 private String[] receiversName;
 /*
  Hashtable indexed on receivers' name; each entry contains the interface of the stub connected
  to the receiver. These interfaces are gates used by the core for coordinating the whole multi-
  channel behavior.
  */
```

```
private Hashtable receivers;
private Vector sendersName;
/*
 Hashtable indexed on senders' name; each entry contains the interface of the stub connected
 to the sender. These interfaces are gates used by the core for coordinating the whole multi-channel
 behavior.
 */
private Hashtable senders;
/*
 The name of the class this kind of multi-channel uses as receiverStub. This approach allows us
 to define multi-channel which redefine the stubs behavior.
 */
private String receiverStubClassName;
/*
 The name of the class that this kind of multi-channel uses as senderStub. This approach allows us
 to define multi-channel which redefine the stubs behavior.
 */
private String senderStubClassName;
private String serverName;

public String kind() {return kind;}
final public void setKind(String k) {kind=k;}
  /* receiverName
  ----------------------------------------------------------------------
  Returns the name of the receiver whose position is specified by indx.
  The position is related to how the names are stored. At the moment this method can be used
  consistently only in conjunction with loop statements.
  ----------------------------------------------------------------------
  arguments: indx the index of the position.
  returns: the name of the receiver.
  ----------------------------------------------------------------------
  */
final public String receiverName(int indx) {return receiversName[indx];}
  /* setReceiverName
  ----------------------------------------------------------------------
  inserts the name of a receiver at the position specified by indx.
  ----------------------------------------------------------------------
  The method doesn't check if the slot is already used.
  Note that you have to keep the set of the name alphabetically sorted in order to allow the
  core to connect to its stubs and vice versa.
  ----------------------------------------------------------------------
  arguments: indx the index of the position.
             name the name of the receiver.
  ----------------------------------------------------------------------
  */
final public void setReceiverName(int indx, String name) {receiversName[indx]=name;}
final public void setRsName(String[ ] names) {
  Arrays.sort(names, String.CASE_INSENSITIVE_ORDER);
  receiversName = names;
}
  /* howManyReceivers
  ----------------------------------------------------------------------
  returns the number of receivers bound to this multi-channel.
  ----------------------------------------------------------------------
  The number of receivers is a quantities fixed at compile-time and depends on the kinds
  statement.
  ----------------------------------------------------------------------
  returns: the number of receivers.
  ----------------------------------------------------------------------
  */
final public int howManyReceivers() {return receiversName.length;}
  /* senderName
```

returns the name of the sender whose position is specified by indx.

The position is related to how the names are stored. At the moment this method can be used consistently only in conjunction with loop statements.

arguments: indx the index of the position.
returns: the name of the sender.
exceptions: ArrayIndexOutOfBoundsException thrown when the index passed is too big with respect to the number of the senders.

```
  */
final public String senderName(int indx) throws ArrayIndexOutOfBoundsException {
  return (sendersName.elementAt(indx)).toString();
}
/* setSenderName
```

inserts the name of a sender at the position specified by indx.

The method doesn't check if the slot is already used. The method provides to append a new sender to the tail when the current number of sender is passed as index.

arguments: indx the index of the position.
name the name of the receiver.

```
 */
final public void setSenderName(int indx, String name) throws ArrayIndexOutOfBoundsException {
  if (indx == sendersName.size()) sendersName.insertElementAt(name, indx);
  else sendersName.setElementAt(name, indx);
}
/* howManySenders
```

returns the number of senders bound to this multi-channel, at the moment of the call.

The number of senders is zero at the core creation and increase during the execution with the calls the core has to process.

returns: the number of senders.

```
 */
final public int howManySenders() {return sendersName.size();}
final public receiverStubInterface receiverStub(String name)
        {return (receiverStubInterface)(receivers.get(name));}
final public receiverStubInterface receiverStubByPosition(int indx)
        {return (receiverStubInterface)(receivers.get(receiverName(indx)));}
final public void setReceiverStub(String name, receiverStubInterface RSI){receivers.put(name, RSI);}
final public senderStubInterface senderStub(String name)
        {return (senderStubInterface)(senders.get(name));}
final public senderStubInterface senderStubByPosition(int indx)
        {return (senderStubInterface)(senders.get(senderName(indx)));}
final public void setSenderStub(String name, senderStubInterface SSI){senders.put(name, SSI);}

final public String receiverStubClassName() {return receiverStubClassName;}
final public void setReceiverStubClassName(String name) {receiverStubClassName=name;}
final public String senderStubClassName() {return senderStubClassName;}
final public void setSenderStubClassName(String name) {senderStubClassName=name;}

/* constructor of multi-channels.
```

This constructor creates and registers as server the core of a multi-channel of kind: kind. The core is registered using the following string:

“_”+*kind*+“{ ”rec_1+“ ”+rec_2 … +rec_n+“ }”

then, it instructs each receivers to create and to register as server their stubs.
The stub of the receiver rec_i is registered as:

“__”+*kind*+“{ ”rec_1+“ ”+rec_2 … +rec_n+“ }”+“__”+rec_i

finally the core connect itself with its stubs on the receivers site.

---

arguments: kind a String representing the multi-channel's behavior.
RsName an Array of Strings representing the name of the receivers, which it is connected to.
SSClassName a String specifying the class of the stubs that will be attached to the multi-channel senders.
RSClassName a String specifying the class of the stubs that will be attached to the multi-channel receivers.
exceptions: ReceiverStubNotFoundException thrown when one of the specified receiver wasn't started before the core.

```
*/

public channelCore(String kind, String[] RsName, String SSClassName, String RSClassName)
      throws RemoteException, ReceiverStubNotFoundException {
  int i;
  receivers = new Hashtable(); senders = new Hashtable();
  setReceiverStubClassName(RSClassName); setSenderStubClassName(SSClassName);
  setRsName(RsName); setKind(kind); sendersName = new Vector(10, 1);
                 // I'm building up the name which the channel use to register itself as server
  serverName = "__"+kind()+"{ ";
  for (i=0; i<howManyReceivers(); i++) serverName += receiverName(i) + " ";
  serverName += "}";
                                   // I'm registering myself as server
  try {
    Naming.rebind(serverName, this);
  } catch(MalformedURLException e) {
    System.out.println("*** Troubles in registering the core as a RMI server");
    e.printStackTrace();
  }
                                   // I'm instructing the receivers about creating their stubs
  try {
    reflectiveReceiverInterface RSI;
    for (i=0; i<howManyReceivers(); i++) {
      RSI = (reflectiveReceiverInterface)lookupEveryWhere.lookup(receiverName(i));
      RSI.attachingStub(serverName, receiverStubClassName(), receiverName(i));
      setReceiverStub(receiverName(i),
           (receiverStubInterface)lookupEveryWhere.lookup(serverName+"__"+receiverName(i)));
    }
  } catch(Exception e) {
      throw new ReceiverStubNotFoundException("*** Troubles in detecting remote "+
                 +"receiver stubs: "+receiverName(i)+" I look for: "+
                 +serverName+"__"+receiverName(i));
  }
}

/* coreMetaBehavior
```

This method embodies the reflective behaviour realized by the multi-channel.

This method performs meta-computations on the reified method call:

multiRMI(RsName, methodName, args)

the computations performed can't make assumption about where their are performed.
This method must be overrode in order to implement new kind of multi-channels.
as default behavior, it delivers the message to the specified receivers, collects the results and returns the result of the computation of the first receiver to the sender.

Note that receivers can be a subset of the multi-channel's receivers

```
  arguments: msg the hijacked method call.
  returns: the result of the method call.
  exceptions: MethodDoesNotExistException thrown when the method reified doesn't exist in the
              referent class.

  */

public Object coreMetaBehavior(mChaRMMethodCall msg)
          throws MethodDoesNotExistException, RemoteException {
  Object[ ] result = new Object[(msg.receivers()).length];
  for(int i=0; i<(msg.receivers()).length; i++)
    result[i] = ((receiverStubInterface)receiverStub((msg.receivers())[i])).invoke(msg);
  return result[0];
}

 /* retrieveSenderFieldValue

  retrieves the content of a field of the specified sender.

  arguments: senderName the name of the sender the core questions about the content of its field
             fieldName the name of the field we want to inspect.
  returns: the content of the inspected field.
  exceptions: SenderStubNotFoundException thrown when the sender is unreacheable.
              FieldDoesNotExistException thrown when the class of the sender doesn't define the
              field passed.

  */
public Object retrieveSenderFieldValue(String senderName, String fieldName)
       throws SenderStubNotFoundException, FieldDoesNotExistException, RemoteException {
  senderStubInterface SSInterface = (senderStubInterface)senderStub(senderName);
  if (SSInterface == null) throw new SenderStubNotFoundException();
  return SSInterface.retrieveField(fieldName);
}
 /* retrieveReceiverFieldValue

  retrieves the content of a field of the specified receiver.

  arguments: receiverName the name of the receiver the core questions about the content of its
             field fieldName the name of the field we want to inspect.
  returns: the content of the inspected field.
  exceptions: ReceiverStubNotFoundException thrown when the receiver is unreacheable.
              FieldDoesNotExistException thrown when the class of the sender doesn't define the
              field passed.

  */
public Object retrieveReceiverFieldValue(String receiverName, String fieldName)
       throws ReceiverStubNotFoundException, FieldDoesNotExistException, RemoteException {
  receiverStubInterface RSInterface = (receiverStubInterface)receiverStub(receiverName);
  if (RSInterface == null) throw new ReceiverStubNotFoundException();
  return RSInterface.retrieveField(fieldName);
}

 /* supplyASenderStub()

  on request, it supplies the class name of its senderStub to the requiring client.

  when a sender needs that one of its computation be reified by a multi-channel
  it looks for a sender stub of such multi-channel.

  This method cannot be overrode nor used directly by users.
```

```
    returns: the class name of its senderStub.
    ---------------------------------------------------------------
    */
  final public String supplyASenderStub() throws java.rmi.RemoteException {
    return senderStubClassName();
  }

  /* the invocation of this method allows to connect the multi-channel to the new stub.
    ---------------------------------------------------------------
    This method cannot be overrode nor used directly by users.
    ---------------------------------------------------------------
    arguments: senderStubName the name of the stub, who the multi-channel have to connect to.
    exceptions: SenderStubNotFoundException thrown when the senderStub is not ready to be linked
    to the core.
    ---------------------------------------------------------------
    */
  final public void senderStubHasBeenCreated(String senderStubName)
                  throws SenderStubNotFoundException {
    try {
      setSenderName(howManySenders(), senderStubName);
      setSenderStub(senderStubName, (senderStubInterface)
                          lookupEveryWhere.lookup("__"+senderStubName+serverName));
    } catch(Exception e) { throw new SenderStubNotFoundException(
        "*** Troubles in detecting remote sender's stubs: "+"__"+
        +senderStubName+serverName); }
  }

}
```

A.1.2 Stubs

```
package mChaRM.multichannel;

import java.io.*;
import java.rmi.*;
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;
import java.lang.reflect.*;
import mChaRM.RMI.*;

  /* stub
    ---------------------------------------------------------------
    This class represents the common root for both sender and receiver stub classes. Both sender and
    receiver stubs have common features and behaviors. Both of them have to be connected to the core of
    the multi-channel, and offer it some services. Besides both of them allow the multi-channel to access
    the state of its referents.
    ---------------------------------------------------------------
    */

class stub extends UnicastRemoteObject {
  private channelInterface channelCore;    // the hook towards channel's core

    /* WhoIsMyCore
      ---------------------------------------------------------------
      It supplies a pointer to the core of the multi-channel.
      ---------------------------------------------------------------
      return: an instance of the interface of the multi-channel.
      ---------------------------------------------------------------
      */
  final public channelInterface WhoIsMyCore() {return channelCore;}
```

```
  /* setWhoIsMyCore
  ---------------------------------------------------------------------------
  links the stub to the core of the multi-channel which it is part of.
  ---------------------------------------------------------------------------
  arguments: kind the kind of the core of the multi-channel which it is part of.
  exceptions: CoreNotFoundException thrown when the core doesn't exist or isn't registered as a
              server.
  ---------------------------------------------------------------------------
  */
final public void setWhoIsMyCore(String kind) throws CoreNotFoundException {
  try {
    channelCore = (channelInterface)lookupEveryWhere.lookup(kind);
  } catch(java.rmi.NotBoundException e) {
      throw new CoreNotFoundException(
          "*** It is impossible to connect the stub to the multi-channel core ["+
          +kind+"], because it isn't registered as server!");
  }
}

private Object referent;                    // the hook towards the stub's referent

  /* setReferent
  ---------------------------------------------------------------------------
  sets the referent of this stub.
  ---------------------------------------------------------------------------
  arguments: referent the referent.
  ---------------------------------------------------------------------------
  */
final public void setReferent(Object r) {referent = r;}

  /* referent
  ---------------------------------------------------------------------------
  accessor to the referent of the stub.
  ---------------------------------------------------------------------------
  return: a pointer to the referent of this stub.
  ---------------------------------------------------------------------------
  */
final public Object referent() {return referent;}

public stub() throws RemoteException {}

  /* retrieveField
  ---------------------------------------------------------------------------
  queries for the contents of a specified field of the referent.
  ---------------------------------------------------------------------------
  exceptions: FieldDoesNotExistException thrown when the specified field isn't declared in the
              class of the referent.
  arguments: fieldName the name of the field whose content is retrieved.
  return: an object representing the contents of the specified field.
  ---------------------------------------------------------------------------
  */
public Object retrieveField(String fieldName) throws FieldDoesNotExistException {
  try {
    Field f = (referent().getClass()).getDeclaredField(fieldName);
    return f.get(referent());
  } catch(NoSuchFieldException e) {
     throw new FieldDoesNotExistException("*** The requested field ["+fieldName+
        +"] is not declared in the class of the referent ["+referent().getClass().getName()+"]");
  } catch(IllegalAccessException e){
     throw new FieldDoesNotExistException("*** The requested field ["+fieldName+
        +"] cannot be accessed"); }
}
```

```
}


package mChaRM.multichannel;

import java.rmi.server.UnicastRemoteObject;
import java.net.MalformedURLException;
import java.rmi.RemoteException;
import java.rmi.*;
import java.io.*;

 /* senderStub
  ----------------------------------------------------------------------
  Instances of this class are used as stubs of the multi-channel on the sender side. Each sender which
  asks a reflective computation to a multi-channel will be extended by a stub of such a multi-channel.
  This class defines the meta-computations that must be performed on the sender side. Each new meta-
  behavior to be performed on the sender side can be defined extending this class and overrinding the
  method senderSideMetaBehavior.
  ----------------------------------------------------------------------
  */

public class senderStub extends stub implements senderStubInterface {

  /* senderStub constructor.
   ----------------------------------------------------------------------
   it initialize the stub, registers it as a remote server, and connect it to the core of the multi-
   channel, and vice versa.
   ----------------------------------------------------------------------
   arguments:  myReferent a representant of the stub referent.
               myKind the multi-channel's name.
               myReferentName the referent's name.
   exceptions: CoreNotFoundException thrown when the core of the multi-channel, which the stub
                 is part of, doesn't exist or isn't correctly registered as a server.
               SenderStubCannotBeRegisteredAsAServerException thrown when the stub we are
                 creating cannot be registerd as a server.
               SenderStubNotFoundException thrown when the core can't link to this sender stub.
   ----------------------------------------------------------------------
   */
 public senderStub(Object myReferent, String myKind, String myReferentName)
        throws SenderStubNotFoundException, CoreNotFoundException,
               SenderStubCannotBeRegisteredAsAServerException, RemoteException {
   setReferent(myReferent);
   try {
     Naming.rebind("__"+myReferentName+myKind, this); // register myself
     setWhoIsMyCore(myKind);
     WhoIsMyCore().senderStubHasBeenCreated(myReferentName);
   } catch(RemoteException e) {
        throw new SenderStubCannotBeRegisteredAsAServerException(
          "*** Troubles in registering the sender's stub "+myReferentName+" as a RMI server");
   } catch(MalformedURLException e) {
        throw new SenderStubCannotBeRegisteredAsAServerException(
          "*** Troubles in registering the sender's stub "+myReferentName+" as a RMI server");}
 }

  /* stubBehavior
   ----------------------------------------------------------------------
   This method embodies the reflective behaviour performed on the sender side, and also forwards
   the request trapped to the core of the multi-channel.
   ----------------------------------------------------------------------
   This method starts on the sender side the meta-computations related to the reified method call:
                              multiRMI(RsName, methodName, args)
```

```
   then it passes all the stuff to the core of the multi-channel.
   the computations performed can make assumption about the fact it will be performed on the
   sender site. This method can't be neither overrode nor directly called by the programmer.
   ---------------------------------------------------------------------------------------
   Note that receivers can be a subset of the multi-channel's receivers, and as side-effect the
   senderSideMetaBehavior method can change the arguments passed to the core of channel
   ---------------------------------------------------------------------------------------
   arguments: msg the hijacked method call.
   return: the result of the method call.
   ---------------------------------------------------------------------------------------
  */
  final public Object stubBehavior(mChaRMMethodCall msg) {
    try {
      senderSideMetaBehavior(msg);
      return WhoIsMyCore().coreMetaBehavior(msg);
    } catch(Exception e) {
      System.out.println("*** Troubles in contacting the channel core");
      e.printStackTrace();
    }
    return null;
  }

  /* senderSideMetaBehavior
   ---------------------------------------------------------------------------------------
   This method performs the sender side meta-behavior.
   ---------------------------------------------------------------------------------------
   It is called before passing all the staff to the core and it permits to perform meta-computations
   on sender side. As default behavior, nothing it is done, it is possible to define new behavior
   defing a new subclass overriding this method.
   ---------------------------------------------------------------------------------------
   arguments: msg the hijacked method call.
              args the actual arguments of the call.
   ---------------------------------------------------------------------------------------
   */
  public void senderSideMetaBehavior(mChaRMMethodCall msg) {}

}


package mChaRM.multichannel;

import java.rmi.*;
import java.io.*;
import java.lang.Class;
import java.lang.reflect.*;
import java.net.MalformedURLException;

  /* receiverStub
  ---------------------------------------------------------------------------------------
  Instances of this class are used as stubs of the multi-channel on the receiver site. Each receiver
  connected to a multi-channel will be extended by a stub of such a multi-channel. This class defines
  the meta-computations that have to be performed on the receiver side. Each new meta-behavior to be
  performed on the receiver side can be defined extending this class and overrinding the method
  receiverSideMetaBehavior.
  ---------------------------------------------------------------------------------------
  */

public class receiverStub extends stub implements receiverStubInterface {

   /* receiverStub constructor.
   ---------------------------------------------------------------------------------------
    it initialize the stub, registers it as a remote server, and connect it to the core of the multi-
```

channel.

---

arguments: myReferent a representant of the stub referent.
myKind the multi-channel's name.
myReferentName the referent's name.
exceptions: ReceiverStubCannotBeRegisteredAsAServerException thrown when the stub cannot be registered as a server.
CoreNotFoundException thrown when the core of the multi-channel, which the stub is part of, doesn't exist or isn't correctly registered as a server.

**/*

```
public receiverStub(Object myReferent, String myKind, String myReferentName)
      throws RemoteException, CoreNotFoundException,
            ReceiverStubCannotBeRegisteredAsAServerException {
  setReferent(myReferent);
  try {
    setWhoIsMyCore(myKind);
    Naming.rebind(myKind+"__"+myReferentName, this); // register myself
  } catch(MalformedURLException e) {
      throw new ReceiverStubCannotBeRegisteredAsAServerException(
          "*** Troubles in registering the receiver's stub "+myReferentName+" as a RMI server");
  }
}
```

/ invoke*

---

it performs the meta-computation on the receiver side – calling the receiverSideMetaBehavior –, then really invoke the method reified by the meta-computationi – invoking the tInvoke method.

It is called by the coreMetaBehavior method of the multi-channel core. It cannot be overrode. Note that as side-effect the receiverSideMetaBehavior method can change the value of the arguments used by tInvoke.

---

arguments: msg the hijacked method call.
return: the result of the method invocation
exceptions: MethodDoesNotExistException thrown when the method reified doesn't exist in the referent class.

**/*

```
final public Object invoke(mChaRMMethodCall msg) throws MethodDoesNotExistException {
  receiverSideMetaBehavior(msg);
  return tInvoke(msg);
}

final private Class[ ] extractArgsClasses(Object[ ] args) {
  if (args == null) return null;
  Class[ ] c = new Class[args.length];
  for (int i=0; i<args.length; i++) c[i] = args[i].getClass();
  return c;
}
```

/ tInvoke*

---

it invokes a method with the actual arguments on the receiver side.

---

it uses the reflection mechanism offered by the Java API Core Reflection. It is able to handle overloaded method. It cannot neither be overrode nor directly invoked by user.

---

arguments: msg the hijacked method call.
return: the result of the method invocation.

```
    exceptions: MethodDoesNotExistException thrown when the method reified doesn't exist in the
                referent class.
    ----------------------------------------------------------------------
    */
  final protected Object tInvoke(mChaRMMethodCall msg) throws MethodDoesNotExistException {
    Class[] c = extractArgsClasses(msg.actualArguments());
    try {
      Method method = (referent().getClass()).getDeclaredMethod(msg.getMethodName(), c);
      return method.invoke(referent(), msg.actualArguments());
    } catch(Exception e) {
      String error = "*** The method: "+msg.getMethodName()+"( ";
      for(int i=0;i<c.length;i++) error += c[i].getName()+" ";
      error += ") wasn't declared in the referent class ["+referent().getClass().getName()+"]";
      throw new MethodDoesNotExistException(error);
    }
  }

   /* receiverSideMetaBehavior
    ----------------------------------------------------------------------
    it is called before the true invokation and permits to perform the last meta-computation on the
    receiver side.
    ----------------------------------------------------------------------
    as default nothing it is done, it is possible to define new
    behavior defing a new subclass overriding this method.
    ----------------------------------------------------------------------
    arguments: msg the hijacked method call.
    ----------------------------------------------------------------------
    */
  public void receiverSideMetaBehavior(mChaRMMethodCall msg) {}

}
```

A.1.3 Message Representative

```
package mChaRM.multichannel;

import java.util.Vector;
import java.io.Serializable;

  /* mChaRMMethodCall
  ----------------------------------------------------------------------
  This class embodies the method call filtered by the multi-channel. This class also supplies
  some methods for manipulate method calls.
  ----------------------------------------------------------------------
  */

public class mChaRMMethodCall implements Serializable {

 private class mChaRMArgs implements Serializable {
    private Vector args;
    private mChaRMArgs(Object[] a) {
      args = new Vector(10, 2);
      if (a != null) for(int i=0;i<a.length;i++) args.add(i, a[i]);
    }
    private Object[] args() { return args.toArray(); }
    private void printmChaRMArgs() {
      if (args.size() == 0) System.out.print("()");
      else {
        System.out.print("("+args.get(0));
        for(int i=1;i<args.size();i++) System.out.print(", "+args.get(i));
        System.out.print(")");
```

```
      }
    }
    private int numberOfArgs() {return args.size();}
    private Object killArg(int pos) {return args.remove(pos);}
    private void addArg(int pos, Object val) {args.add(pos, val);}
    private Object getArg(int pos) {return args.get(pos);}
    private Object setArg(int pos, Object o) {
      Object old = args.get(pos);
      args.set(pos, o);
      return old;
    }
  }

  private String sender;
  private String[] receivers;
  private String methodName;
  private mChaRMArgs actualArguments;

  public mChaRMMethodCall(String s, String[] rs, String m, Object[] args) {
    sender = s;
    receivers = rs;
    methodName = m;
    actualArguments = new mChaRMArgs(args);
  }

    /* sender
     ---------------------------------------------------------------
     returns the name of the sender of the message.
     ---------------------------------------------------------------
     It only supplies a String representative of the sender but it is simple to retrieve a reference of
     the sender, using the methods supplied by the multi-channel.
     ---------------------------------------------------------------
     return: the name of the sender.
     ---------------------------------------------------------------
     */
  public String sender() {return sender;}

    /* receivers
     ---------------------------------------------------------------
     returns an array containing the name of the reecivers involved in the call.
     ---------------------------------------------------------------
     It only supplies a String representative of each receiver but it is simple to retrieve a reference
     of the real receiver, using the methods supplied by the multi-channel.
     ---------------------------------------------------------------
     return: an array of receiver's names.
     ---------------------------------------------------------------
     */
  public String[] receivers() {return receivers;}

  public String getMethodName() {return methodName;} // returns the name of the method called.

    /* setMethodName
     ---------------------------------------------------------------
     changes the name of the called method.
     ---------------------------------------------------------------
     arguments: m the name of the method to be used.
     return: the name of the method originally called.
     ---------------------------------------------------------------
     */
  public String setMethodName(String m) {
    String oldMethod = methodName;
    methodName = m;
```

```
  return oldMethod;
}

  /* actualArguments
  ---------------------------------------------------------------
  returns the actual arguments used in the trapped call.
  ---------------------------------------------------------------
  return: an array of Object representing the arguments of the call.
  ---------------------------------------------------------------
  */
public Object[] actualArguments() {return actualArguments.args();}

  /* inspectArgument
  ---------------------------------------------------------------
  returns the actual value of a specified argument.
  ---------------------------------------------------------------
  arguments:  position the position in the call of the argument we want to inspect.
  return:     an Object containing the value of the inspected argument.
  exceptions: ReferenceToNonExistentArgumentException thrown when position exceeds (both the
              lower and the upper bound) the number of arguments of the call.
  ---------------------------------------------------------------
  */
public Object inspectArgument(int position) throws ReferenceToNonExistentArgumentException {
  if (position <0 | position > actualArguments.numberOfArgs())
    throw new ReferenceToNonExistentArgumentException("the call doesn't have "+position+" arguments");
  else return actualArguments.getArg(position);
}

  /* modifyArgument
  ---------------------------------------------------------------
  changes the value of a specified argument and returns the original value of such an argument.
  ---------------------------------------------------------------
  arguments:  position the position in the call of the argument we want to modify.
              newValue the new value for the specified argument.
  return:     an Object containing the original value of the modified argument.
  exceptions: ReferenceToNonExistentArgumentException thrown when position exceeds (both the
              lower and the upper bound) the number of arguments of the call.
  ---------------------------------------------------------------
  */
public Object modifyArgument(int position, Object newValue)
      throws ReferenceToNonExistentArgumentException {
  if (position <0 | position > actualArguments.numberOfArgs())
     throw new ReferenceToNonExistentArgumentException("the call doesn't have "+position+" arguments");
  else return actualArguments.setArg(position, newValue);
}

  /* insertArgument
  ---------------------------------------------------------------
  inserts a new argument at the specified position in the call.
  ---------------------------------------------------------------
  arguments: position the position in the call that will have the argument we want to insert.
             value the value of the new argument.
  ---------------------------------------------------------------
  */
public void insertArgument(int position, Object value) {actualArguments.addArg(position, value);}

  /* removeArgument
  ---------------------------------------------------------------
  removes the specified argument from the embodied call.
  ---------------------------------------------------------------
  arguments:  position the position in the call of the argument we want to remove.
```

```
    return:      an Object containing the value of the removed argument.
    exceptions:  ReferenceToNonExistentArgumentException thrown when position exceeds (both the
                 lower and the upper bound) the number of arguments of the call.
    ------------------------------------------------------------------------
    */
  public Object removeArgument(int position) throws ReferenceToNonExistentArgumentException {
    if (position <0 | position > actualArguments.numberOfArgs())
       throw new ReferenceToNonExistentArgumentException("the call doesn't have "+position+" arguments");
    else return actualArguments.killArg(position);
  }

    // prints the embodied method call.
  public void printmChaRMMethodCall() {
    System.out.print(sender+" --- "+methodName);
    actualArguments.printmChaRMArgs();
    System.out.print(" --->"+" ["+receivers[0]);
    for(int i=1;i<receivers.length;i++) System.out.print(","+receivers[i]);
    System.out.println("]");
  }

   /* hasArgs
    ------------------------------------------------------------------------
    wonders about wheather the embodied call has or not arguments.
    ------------------------------------------------------------------------
    return: true if has arguments false otherwise.
    ------------------------------------------------------------------------
    */
  public boolean hasArgs() {return (actualArguments.numberOfArgs() != 0);}
}
```

A.2 MOP Managing Base-Level Extensions

```
package mChaRM.MOP;

import openjava.mop.*;
import openjava.ptree.*;
import openjava.syntax.*;
import java.util.*;

/* mChaRM_MOP
   ------------------------------------------------------------------------
   meta-object handling the syntax enrichment and expanding it in order to support the multi-channel
   reification approach.

   it implements two syntax extensions:
     - one for the sender-side:
          kinds {kind <kind> <rec>{and <rec>} apply <method>{, <method>}}
   of course, each base object can use both.
   ------------------------------------------------------------------------
   extends:         OJClass
   instantiates:    Metaclass
   public methods:  createAFieldForAnHashTableOfStubs, addMultiRMIMethods, isRegisteredKeyword,
                    addCodeForSendersStuff, addCodeForCreatingReceiverStubs, getDeclSuffixRule,
                    getMethodByName, translateDefinition
   ------------------------------------------------------------------------
*/

public class mChaRM_MOP instantiates Metaclass extends OJClass {

   /* createAFieldForAnHashTableOfStubs
```

```
    arguments: name - the name of the new field
    returns: the just created field

    it creates a field declaration for a hashtable, which will keep the hooks for the stubs
  */

private OJField createAFieldForAnHashTableOfStubs(String name) {
  TypeName tn = new TypeName("java.util.Hashtable");
  VariableInitializer vi = new AllocationExpression(tn, new ExpressionList());
  FieldDeclaration fd = new FieldDeclaration(new ModifierList(ModifierList.PRIVATE), tn, name, vi);
  return new OJField(getEnvironment(), this, fd);
}

 /* addMultiRMIMethods

    arguments: none

    Java doesn't has a multi-communication mechanism, thus I supposed the existence of a method
    called multiRMI, which implements this kind of communication. The meta-object puts a method
    multiRMI in the classes needing it. I encapsulated in it the reification mechanism, I didn't defi-
    ne a default behavior, but it is simple do it.

    this method adds two multiRMI method one with 3 arguments calculates the kind of the call and
    calls the other multiRMI method, which starts the meta-computation
  */

private void addMethodMultiRMI()
    throws CannotAlterException, MOPException {
  OJMethod MRMI = new OJMethod( this, OJModifier.forModifier( OJModifier.PRIVATE ),
                OJClass.forName( "java.lang.Object" ), "multiRMI",
                new OJClass[]{ OJClass.forName( "java.lang.String" ),
                OJClass.forName( "java.lang.String[]" ), OJClass.forName( "java.lang.Object[]" ) },
                new java.lang.String[]{ "methodName", "RsName", "args" }, null, null );
  String MRMIcode = "java.lang.String servers = \"{ \";";
  MRMIcode += "java.util.Arrays.sort(RsName,i";
  MRMIcode += "java.lang.String.CASE_INSENSITIVE_ORDER);";
  MRMIcode += "for (int i=0; i<RsName.length; i++) servers += RsName[i] +";
  MRMIcode += "\" \"; servers += \"}\";";
  MRMIcode += "java.lang.String kind = (java.lang.String)__kinds.";
  MRMIcode += "get(methodName+servers); if (kind !=null) {";
  MRMIcode += "kind = \"__\"+kind+servers;";
  MRMIcode += "if (HashTableForSStub.get(kind) == null) {";
  MRMIcode += "try {";
  MRMIcode += "java.lang.String senderStubClassName = ((channelInterface)";
  MRMIcode += "mChaRM.RMI.lookupEveryWhere.lookup(kind)).supplyASenderStub();";
  MRMIcode += "Class[] consArgsClasses = new Class[]{ Class.forName(\"java.lang.Object\"),";
  MRMIcode += "Class.forName(\"java.lang.String\"), Class.forName(\"java.lang.String\") };";
  MRMIcode += "HashTableForSStub.put(kind, ((Class.forName(senderStubClassName)).getConstructor(";
  MRMIcode += "consArgsClasses).newInstance(new Object[]{this, kind, whoAmI()})));";
  MRMIcode += "} catch ( Exception e ) {";
  MRMIcode += "System.out.println( \"*** Trouble in detecting the coreChannel: \" + kind );";
  MRMIcode += "e.printStackTrace(); } } try {";
  MRMIcode += "mChaRMMethodCall msg = new mChaRMMethodCall(whoAmI, RsName, methodName, args);";
  MRMIcode += "return ((senderStub)HashTableForSStub.get(kind)).stubBehavior(msg);";
  MRMIcode += "} catch (Exception e) {System.out.println( \"*** Trouble in dialoguing with ";
  MRMIcode += "coreChannel: \" + kind ); e.printStackTrace(); } }";
  MRMIcode += "return null;}";
  MRMI.setBody( makeStatementList( MRMI.getEnvironment(), MRMIcode ) );
  addMethod( MRMI );
```

```
}

  /* isRegisteredKeyword
     ________________________________________________________________
     arguments: keyword - the keyword to check
     ________________________________________________________________
     is used by openjava to check the existence of a keyword met during the parsing.
     ________________________________________________________________
   */

public static boolean isRegisteredKeyword(String keyword) {
  if (keyword.equals("kinds")) return true;
  return OJClass.isRegisteredKeyword( keyword );
}

  /* getDeclSuffixRule
     ________________________________________________________________
     arguments: keyword - the keyword to parse
     returns: the parse tree of the code tied to the keyword
     ________________________________________________________________
     is used by the openjava interpreter to parse new keywords, it is the core of the detecting
     phase of the new extensions.
     ________________________________________________________________
   */

public static SyntaxRule getDeclSuffixRule(String keyword) {
  if (keyword.equals("kinds"))
    return new IterationRule(new PrepPhraseRule("kind",
          new CompositeRule(new NameRule(), new PrepPhraseRule("with",
          new CompositeRule(new NameRule(),
                new IterationRule(new PrepPhraseRule("and", new NameRule()), true))),
          new PrepPhraseRule("apply",
                new DefaultListRule(new NameRule(), TokenID.COMMA, false)))));
  return OJClass.getDeclSuffixRule( keyword );
}

  /* addCodeForSendersStuff
     ________________________________________________________________
     arguments: pt - the parse tree of the keyword 'kinds'
     ________________________________________________________________
     it adds a structure in the base-class, keeping the information for calling the multiRMI
     with the right kind, and thus using the right multi-channel.

     it takes the related information by the parse tree associated to the keyword 'kinds'.
     ________________________________________________________________
   */

private void addCodeForSendersStuff(ParseTree pt) {
  String initializeCode ="";
  OJMethod initialize=null;
  try {
    initialize = new OJMethod(this, OJModifier.forModifier(PRIVATE+STATIC),
          OJClass.forName("void"), "__initialize", null, null, null, null);
  } catch(OJClassNotFoundException e0) {
    System.out.println("*** Trouble in creating the initialize method");
    e0.printStackTrace();
  }
    // add a static hashtable for handling the dynamic binding with channels
  try {
    TypeName tn = new TypeName("java.util.Hashtable");
    VariableInitializer vi = new AllocationExpression(tn, new ExpressionList());
    FieldDeclaration fd = new FieldDeclaration(
```

```
            new ModifierList(ModifierList.PRIVATE+ModifierList.STATIC), tn, "__kinds", vi);
    addField(new OJField(getEnvironment(), this, fd));
  } catch(CannotAlterException e) {
    System.out.println("*** Troubles in adding the __kinds structure!");
    e.printStackTrace();
  }

  Enumeration e = ((openjava.ptree.List)pt).elements(), aKind;
  while (e.hasMoreElements()) {
    aKind = ((openjava.ptree.List)e.nextElement()).elements();
    String theKind = (aKind.nextElement()).toString();
    Enumeration anEnum = ((openjava.ptree.List)aKind.nextElement()).elements();
    Vector ancillary = new Vector();
    ancillary.addElement(anEnum.nextElement().toString());
    anEnum = ((openjava.ptree.List)(anEnum.nextElement())).elements();
    while (anEnum.hasMoreElements()) ancillary.addElement(anEnum.nextElement().toString());
    Object[ ] aux = new String[1];
    aux = ancillary.toArray(aux);
    Arrays.sort(aux, String.CASE_INSENSITIVE_ORDER);
    String receivers = "{ "; for(int i=0;i<aux.length;i++) receivers += aux[i]+" "; receivers += "}";
    anEnum = ((openjava.ptree.List)aKind.nextElement()).elements();
    while (anEnum.hasMoreElements())
      initializeCode += "__kinds.put(\""+anEnum.nextElement().toString()+
              +receivers+"\",\""+theKind+"\");\n";
  }
  try {
    initialize.setBody(makeStatementList(initialize.getEnvironment(), initializeCode));
    addMethod(initialize);
  } catch(Exception e1) {
    System.out.println("*** Trouble in adding the initialize method");
    e1.printStackTrace();
  }
  OJMethod mainMethod = getMethodByName("main");
  try {
    mainMethod.setBody(makeStatementList(mainMethod.getEnvironment(),
          (mainMethod.getEnvironment().currentClassName()+".__initialize();"+
          +mainMethod.getBody()).toString()));
  } catch(Exception e2) {
    System.out.println("*** Trouble in modifying the main");
    e2.printStackTrace();
  }
}

  /* addCodeForCreatingReceiverStubs
     ------------------------------------------------------------
     arguments: pt - parse tree associates to the keyword 'hooks'
     ------------------------------------------------------------
     it adds the code to create and attach the receiver's stubs.

     the information are taken from the parse tree associated to
     the keyword 'hooks'.
     ------------------------------------------------------------
  */

private void addMethodAttachingStub()
      throws openjava.mop.MOPException, openjava.mop.CannotAlterException {
  String ASCode; OJMethod AS;
  OJClass stringClass = OJClass.forName( "java.lang.String" );
  AS = new OJMethod( this, OJModifier.forModifier( OJModifier.PUBLIC ),
          OJClass.forName( "void" ), "attachingStub",
          new OJClass[ ]{ stringClass, stringClass, stringClass },
          new String[ ]{ "kind", "className", "referentName" }, null, null );
```

```
    ASCode = "try { Class[] consArgsClasses = new Class[]{";
    ASCode += "Class.forName(\"java.lang.Object\"),";
    ASCode += "Class.forName(\"java.lang.String\"),";
    ASCode += " Class.forName(\"java.lang.String\")};";
    ASCode += "HashTableForRStub.put(kind+\"__\"+referentName,";
    ASCode += "(Class.forName(className)).getConstructor(consArgsClasses).";
    ASCode += "newInstance(new Object[]{this, kind, referentName}));";
    ASCode += "} catch(java.lang.ClassNotFoundException e) {System.out.println(";
    ASCode += "\"*** Trouble: one of the parameters class is not a class!\");";
    ASCode += "e.printStackTrace();";
    ASCode += "} catch(java.lang.NoSuchMethodException e) {System.out.println(";
    ASCode += "\"*** Trouble: this class doesn't have a constructor matching";
    ASCode += "such an interface!\");";
    ASCode += " e.printStackTrace();} catch(java.lang.Exception e) {System.out.println(";
    ASCode += "\"*** Trouble: it is impossible instantiate a new stub from\"";
    ASCode += "+className+\"!\"); e.printStackTrace();}";
    AS.setBody( makeStatementList( AS.getEnvironment(), ASCode ) );
    addMethod( AS );
  }

  /* getMethodByName
     ------------------------------------------------------------
     arguments: name - the name of the method we want retrieve
     returns: the retrieved method, if exists
     ------------------------------------------------------------
     it retrieves the method whose name is passed as arguments
     —
     Note: it works because I pass it only methods I know to be unambiguosus.
     ------------------------------------------------------------
   */

  private OJMethod getMethodByName(String name) {
    OJMethod[] ms = getAllMethods();
    for(int i=0; i<ms.length;i++)
      if ((ms[i].getName()).equals(name))
        return ms[i];
    return null;
  }

  /* translateDefinition
     ------------------------------------------------------------
     arguments: none
     ------------------------------------------------------------
     this method is used by the openjava interpreter, it translates the base-classes code, adding
     some fields, methods, and translating the new syntax.
     ------------------------------------------------------------
   */

  public void translateDefinition() throws MOPException, CannotAlterException {
    ParseTree suffix = this.getSuffix("kinds");
    if (suffix != null) {
      System.out.println( this.toString() + " involves a suffix " + " ( kinds ): "+suffix);
      addField(createAFieldForAnHashTableOfStubs("HashTableForSStub"));
      addCodeForSendersStuff(suffix);
      addMethodMultiRMI();
    }
    addField(createAFieldForAnHashTableOfStubs("HashTableForRStub"));
    addMethodAttachingStub();
    addInterface(OJClass.forName("mChaRM.multichannel.reflectiveReceiverInterface"));
  }

}
```

A.3 MOP Managing Meta-Level Language

```
package PremChaRM;

import openjava.mop.*;
import openjava.syntax.*;
import openjava.ptree.*;
import openjava.ptree.util.PartialParser;
import mChaRM.multichannel.*;
import java.io.*;
import java.util.*;

 /*
  This class represents the OpenJava Preprocessor for a class containing the mChaRM mechanism.
  Its role is to translate the OpenJava class who instantiates it (which is a class written in an extended
  syntax of java) into three pure-java classes. The whole class contains methods and objects representing
  the mChaRM objects Core Channel, Sender and Receiver. The classes created by PremChaRM represent
  these objects and they are obtained sharing out methods and fields according the use they did. A method
  or a field is put into a class only if it is really used by respective mChaRM object.
 */

public class PremChaRM extends OJClass {
 private final static int KIND = 0;
 private final static int CORE = 1;
 private final static int SENDER = 2;
 private final static int RECEIVER = 3;
 private final static int CLASSNAME = 0;
 private final static int ENRICHNAME = 1;
 private final static int PROVIDESNAME = 2;

   // This array of constant contains the names of the base methods.
 private final static String[] NAME_START_METHODS = {"", "coreMetaBehavior",
        "senderSideMetaBehavior", "receiverSideMetaBehavior"};

   // This array of constant contains the names of base classes.
 private final static String[] NAME_BASE_CLASSES = {"", "mChaRM.multichannel.channelCore",
        "mChaRM.multichannel.senderStub", "mChaRM.multichannel.receiverStub" };

 private OJSubClass ojsCore;      // This OJSubClass contains, if present, the new class CORE
 private OJSubClass ojsSender;    // This OJSubClass contains, if present, the new class SENDER
 private OJSubClass ojsReceiver; // This OJSubClass contains, if present, the new class RECEIVER

 private Vector ojMethods = new Vector();  // It contains all the method declared in base class
 private Vector ojMethodsCore = new Vector(); // It contains the method of hierarchy of the Core.
 private Vector ojMethodsSender = new Vector(); // It contains the method of hierarchy of the Sender.
 private Vector ojMethodsReceiver = new Vector(); // It contains the method of hierarchy of the Receiver.

   // This array contains all methods of created Core class.
 private OJMethod[] coreMethods;

 /*
  This array contains name of output class create during a translating process.
  It use during a translating of main class, to find a name of new class and
  change the allocatition and invocation of static methods inside the main class.
  */
 private String[] OJMIclassNames = new String[4];

 /*
  Instance of Graph, is used to check the consistence of an access to fields.
  The prepocessor inserts the information into the graph using some utils
  and check the consistence of an access to fields using member function of PreGraph.
  */
```

```
private PreGraph ojGraph = new PreGraph();

private ObjectList ExtendedSyntax; // it contains the AST of the extedend syntax.
private OJField[] ojFields; // it contains all fields of starting class.
private boolean[] isDefined = {false, false, false};

/*
 During a process that translate class that contains main, is possible
 that in several situation the programm need information contained in
 OJMI, this flag used in combination of function HandleFirstTime(String)
 permit to access to information only one time.
 */
private boolean firstTime = true;

/* isRegisteredKeyword
 ----------------------------------------------------------------
 Function predefined in OJClass, is used to check if the
 keyword present in added syntax regard current OJClass
 ----------------------------------------------------------------
 arguments: keyword String that must be check
 return: true if the keyword is correct
 ----------------------------------------------------------------
*/
public static boolean isRegisteredKeyword(String keyword) {
 if (keyword.equals("kind")) return true;
   return OJClass.isRegisteredKeyword( keyword );
 }

/* getDeclSuffixRule
 ----------------------------------------------------------------
 Function predefined in OJClass, is used to get a parse tree of
 added syntax.
 ----------------------------------------------------------------
 arguments: keyword start String of added syntax
 return: the parse tree contains all added syntax
 ----------------------------------------------------------------
 */
public static SyntaxRule getDeclSuffixRule(String keyword) {
  if (keyword.equals("kind")) {
    SyntaxRule syntax[] = new SyntaxRule[4];
    syntax[0] = new NameRule();
    syntax[1] =
      new IterationRule(
          new SelectionRule(
              new CompositeRule(
                  new PrepPhraseRule( "core", new NameRule() ),
                  new PrepPhraseRule( "enriches", new TypeNameRule() ),
                  new PrepPhraseRule( "provides", new DefaultListRule(
                      new TypeNameRule(), TokenID.COMMA) ) ),
              new PrepPhraseRule( "core", new NameRule() ) ), true );
    syntax[2] =
      new IterationRule(
          new SelectionRule(
              new CompositeRule(
                  new PrepPhraseRule( "senders", new NameRule() ),
                  new PrepPhraseRule( "enriches", new TypeNameRule() ),
                  new PrepPhraseRule( "provides", new DefaultListRule(
                      new TypeNameRule(), TokenID.COMMA) ) ),
              new PrepPhraseRule( "senders", new NameRule() ) ), true );
    syntax[3] =
      new IterationRule(
           new SelectionRule(
```

```
                new CompositeRule(
                    new PrepPhraseRule( "receivers", new NameRule() ),
                    new PrepPhraseRule( "enriches", new TypeNameRule() ),
                    new PrepPhraseRule( "provides", new DefaultListRule(
                        new TypeNameRule(), TokenID.COMMA) ) ),
                new PrepPhraseRule( "receivers", new NameRule() ) ), true );
        return new CompositeRule( syntax );
    } return OJClass.getDeclSuffixRule( keyword );
}
```

/ openClasses*

This method create the output class. It starting after that the added syntax is parsing, it create and initialize the output class.

The new class is insert ipn instance of OJSubClass

**/*

```
public void openClasses() {
    String strClassName, strSuperClassName;

    PreUtils.setName( getName() );
    try {
        OJClass ojcSuperClass;
        ObjectList olSyntax, olInterfaces;
        String strImports[ ], strImplements[ ]=null;

        ParseTreeObject compUnit = getSourceCode();
        do {
            compUnit = compUnit.getParent();
        } while( !( compUnit instanceof CompilationUnit ) );
        strImports = ((CompilationUnit)compUnit).getDeclaredImports();
        putMetaInfo( "Kind", ExtendedSyntax.get(KIND).toString() );

/* Core Handling */

        olSyntax = (ObjectList)ExtendedSyntax.get(CORE);
        if (olSyntax.size() == 0) {
            ojsCore = null;
            putMetaInfo( "Core", NAME_BASE_CLASSES[CORE] );
        } else if (olSyntax.size() == 1) {
            if (olSyntax.get(0) instanceof ObjectList) {
                strClassName = ((ObjectList)olSyntax.get(0)).get(CLASSNAME).toString();
                strSuperClassName = ((ObjectList)olSyntax.get(0)).get(ENRICHNAME).toString();
                olInterfaces = (ObjectList)((ObjectList)olSyntax.get(0)).get(PROVIDESNAME);
                strImplements = new String[olInterfaces.size()+1];
                for( int i=0; i<strImplements.length-1; i++ )
                    strImplements[i] = olInterfaces.get(i).toString();
                strImplements[olInterfaces.size()] = "mChaRM.multichannel.channelInterface";
            } else {
                strClassName = olSyntax.get(0).toString();
                strSuperClassName = NAME_BASE_CLASSES[CORE];
            }

            ojsCore = new OJSubClass(strClassName, strSuperClassName, strImplements, strImports );
            putMetaInfo( "Core", strClassName );
        } else {
            System.err.println( "FATAL ERROR: too many 'core' clauses " );
            System.exit( -1 );
        }

/* Sender Handling */
```

```
      olSyntax = (ObjectList)ExtendedSyntax.get(SENDER);
      strImplements = null;
      if (olSyntax.size() == 0) {
        ojsSender = null;
        putMetaInfo( "Sender", NAME_BASE_CLASSES[SENDER] );
      } else if(olSyntax.size() == 1) {
        if (olSyntax.get(0) instanceof ObjectList) {
          strClassName = ((ObjectList)olSyntax.get(0)).get(CLASSNAME).toString();
          strSuperClassName = ((ObjectList)olSyntax.get(0)).get(ENRICHNAME).toString();
          olInterfaces = (ObjectList)((ObjectList)olSyntax.get(0)).get(PROVIDESNAME);
          strImplements = new String[olInterfaces.size()];
          for( int i=0; i<strImplements.length; i++ )
            strImplements[i] = olInterfaces.get(i).toString();
        } else {
          strClassName = olSyntax.get(0).toString();
          strSuperClassName = NAME_BASE_CLASSES[SENDER];
        }
        ojsSender = new OJSubClass(strClassName, strSuperClassName, strImplements, strImports);
        putMetaInfo( "Sender", strClassName );
      } else {
        System.err.println( "FATAL ERROR: too many 'sender' clauses " );
        System.exit( -1 );
      }

/* Receiver Handling */

      olSyntax = (ObjectList)ExtendedSyntax.get(RECEIVER);
      strImplements = null;
      if(olSyntax.size() == 0) {
        ojsReceiver = null;
        putMetaInfo( "Receiver", NAME_BASE_CLASSES[RECEIVER] );
      } else if(olSyntax.size() == 1) {
        if( olSyntax.get(0) instanceof ObjectList ) {
          strClassName = ((ObjectList)olSyntax.get(0)).get(CLASSNAME).toString();
          strSuperClassName = ((ObjectList)olSyntax.get(0)).get(ENRICHNAME).toString();
          olInterfaces = (ObjectList)((ObjectList)olSyntax.get(0)).get(PROVIDESNAME);
          strImplements = new String[olInterfaces.size()];
          for( int i=0; i<strImplements.length; i++ )
            strImplements[i] = olInterfaces.get(i).toString();
        } else {
          strClassName = olSyntax.get(0).toString();
          strSuperClassName = NAME_BASE_CLASSES[RECEIVER];
        }
        ojsReceiver = new OJSubClass(strClassName, strSuperClassName, strImplements, strImports );
        putMetaInfo( "Receiver", strClassName );
      } else {
        System.err.println( "FATAL ERROR: too many 'receiver' clauses " );
        System.exit( -1 );
      }
    } catch( Exception e ) {
      PreUtils.printExceptionText( "FATAL ERROR: Unable to create File ", e );
      System.exit( -1 );
    }
  }

  /* prepareClasses
```

This method makes two different works: first, it initializes the three output classes inserting import clauses and the relative constructor; then it creates graphs for methods and fields, finally inserts right fields and methods in relative class.
In the first phase it finds all method belong to the superclass of the three output class calling cascade-

Methods, finally it create the three constructor.
In the last phase it divides all member of start class, to do that first find late Binding methods and then for each method found search all method called in. At this time for each method founded search all fields used in. If there are multi-referred fields it indicate a Fatal Error and terminate the esecution otherwise it splits the whole classe in three classes.

```
 */
private void prepareClasses() {
 ojMethods = PreUtils.array2Vector( getDeclaredMethods() );
 ParameterList plParams;

 if( ojsCore != null ) {
   ojMethodsCore = cascadeMethods( ojsCore.getSuperClass() );
   isDefined[CORE-1] = true;
 }

 if( ojsSender != null ) {
   ojMethodsSender = cascadeMethods( ojsSender.getSuperClass() );
   isDefined[SENDER-1] = true;
 }

 if( ojsReceiver != null ) {
   ojMethodsReceiver = cascadeMethods( ojsReceiver.getSuperClass() );
   isDefined[RECEIVER-1] = true;
 }

 ojFields = getDeclaredFields();
 for( int i=1; i<=RECEIVER; i++ )
   if( isDefined[i-1] ) insertConstructorMethods(i);
 for( int i=1; i<=RECEIVER; i++ )
   if( isDefined[i-1] ) insertLateBindingMethods(i);

 Vector constr=null;
 int[] className=null;
 for( int j=CORE; j<=RECEIVER; j++ ) {
   switch( j ) {
     case CORE:
       if( !isDefined[j-1] ) continue;
       constr = ojsCore.constructors;
       className = new int[] {1,0,0};
       break;
     case SENDER:
       if( !isDefined[j-1] ) continue;
       constr = ojsSender.constructors;
       className = new int[] {0,1,0};
       break;
     case RECEIVER:
       if( !isDefined[j-1] ) continue;
       constr = ojsReceiver.constructors;
       className = new int[] {0,0,1};
       break;
     }
   for( int i=0; i<constr.size(); i++ ) {
     OJMethod tempMethod = (OJMethod)constr.elementAt(i);
     StatementList tempBody;
     String[] parameters;
     try {
       tempBody = tempMethod.getBody();
       parameters = tempMethod.getParameters();
       Vector multiRefered = PreUtils.fieldHandling( tempBody,
            PreUtils.array2Vector(parameters), PreUtils.array2Vector( ojFields ) );
       for( int z=0; z<multiRefered.size(); z++ )
```

```
          ojGraph.insertFields((OJField)multiRefered.elementAt(z), className );
        } catch( CannotAlterException e ) {
          PreUtils.printExceptionText( "FATAL ERROR: Unexpected Error, in prepareClasses", e );
          System.exit(-1);
        }
      }
    }

    for( int i=0; i<ojGraph.numberOfMethods(); i++ ) {
      OJMethod tempMethod = (ojGraph.methodAt(i));
      StatementList tempBody;
      String[] parameters;
      try {
        tempBody = tempMethod.getBody();
        parameters = tempMethod.getParameters();
        Vector multiRefered = PreUtils.fieldHandling( tempBody,
              PreUtils.array2Vector(parameters), PreUtils.array2Vector( ojFields ) );
        for( int j=0; j<multiRefered.size(); j++ )
          ojGraph.insertFields((OJField)multiRefered.elementAt(j), ojGraph.methodUsedBy(i) );
      } catch( CannotAlterException e ) {
        PreUtils.printExceptionText( "FATAL ERROR: Unexpected Error, in prepareClasses", e );
        System.exit(-1);
      }
    }

    Vector ErrorField = ojGraph.DangerousFields();
    if( ErrorField.size() > 0 ) {
      System.err.println("FATAL ERROR. Double referencing Fields not Permitted");
      System.exit(-1);
    }

    int[] usedIn;
    for( int i=0; i<ojGraph.numberOfFields(); i++ ) {
      if(ojGraph.fieldUsedBy(i, CORE)) ojsCore.addNewFields(ojGraph.fieldAt(i));
      if(ojGraph.fieldUsedBy(i, SENDER)) ojsSender.addNewFields(ojGraph.fieldAt(i));
      if(ojGraph.fieldUsedBy(i, RECEIVER)) ojsReceiver.addNewFields(ojGraph.fieldAt(i));
    }
    for( int i=0; i<ojGraph.numberOfMethods(); i++ ) {
      if(ojGraph.methodUsedBy(i, CORE)) ojsCore.addNewMethods(ojGraph.methodAt(i));
      if(ojGraph.methodUsedBy(i, SENDER)) ojsSender.addNewMethods(ojGraph.methodAt(i));
      if(ojGraph.methodUsedBy(i, RECEIVER)) ojsReceiver.addNewMethods(ojGraph.methodAt(i));
    }
  }

  /* cascadeMethods
  ---------------------------------------------------------------
    This method return all method declared in a class hierarchy.
    To do that it call itself recursively, stops when find a Base class.
  ---------------------------------------------------------------
    arguments: ojClass Class whom find methods
    return: a vector which contains all declared methods
  ---------------------------------------------------------------
  */
  private Vector cascadeMethods( OJClass ojClass ) {
    Vector methods = PreUtils.array2Vector( (Object[])ojClass.getDeclaredMethods() );

    if (!ojClass.getName().equals( NAME_BASE_CLASSES[CORE] ) &&
        !ojClass.getName().equals( NAME_BASE_CLASSES[SENDER]) &&
        !ojClass.getName().equals( NAME_BASE_CLASSES[RECEIVER] ))

      methods.addAll( cascadeMethods( ojClass.getSuperclass() ) );
    return methods;
```

```
}

/* insertConstructorMethods
```

This method finds particulary methods representing constructors.
arguments: className name of class where it search the overridden methods

```
*/
private void insertConstructorMethods(int className) {
 OJSubClass myClass = null;
 String[ ] ConstructorParameters = new String[0], ConstructorParameters1 = new String[0];
 OJMethod method = null;
 OJSubClass ojClass = null;
 int[ ] vUsed = null;
 boolean baseCostructorFound = false;

 Vector myMethods=null;
 switch( className ) {
   case CORE:
     myClass = ojsCore;
     ConstructorParameters = new String[ ] {
              "java.lang.String", "java.lang.String[ ]",
              "java.lang.String", "java.lang.String"
            };
     vUsed = new int[ ] {1,0,0};
     ojClass = ojsCore;
     break;
   case SENDER:
     myClass = ojsSender;
     ConstructorParameters = new String[ ] {
              "java.lang.Object", "java.lang.String",
              "java.lang.String"
            };
     vUsed = new int[ ] {0,1,0};
     ojClass = ojsSender;
     break;
   case RECEIVER:
     myClass = ojsReceiver;
     ConstructorParameters = new String[ ] {
              "java.lang.Object", "java.lang.String",
              "java.lang.String"
            };
     ConstructorParameters1 = new String[ ] {
              "java.rmi.server.UnicastRemoteObject",
              "java.lang.String","java.lang.String"
            };
     vUsed = new int[ ] {0,0,1};
     ojClass = ojsReceiver;
     break;
   }
 for( int i=0; i<ojMethods.size(); i++ ) {
   method = (OJMethod)ojMethods.elementAt(i);
   if( method.getName().equals( myClass.getName() ) ) {
     OJClass[ ] cTipi = method.getParameterTypes();
     String[ ] strTipi = new String[cTipi.length];
     for( int j=0; j<strTipi.length; j++ )
       strTipi[j] = cTipi[j].getName();
     if( !baseCostructorFound &&
         ( PreUtils.equalsParameterType( ConstructorParameters, strTipi ) ||
           PreUtils.equalsParameterType( ConstructorParameters1, strTipi ) ) )
       baseCostructorFound = true;
```

```
      ojClass.addNewConstructor(method);
    }
  }

  if( !baseCostructorFound ) {
    System.err.println("FATAL ERROR: Default constructor not found");
    System.exit(-1);
  }

  try {
    Vector constr = myClass.constructors;
    Vector fstLine = myClass.fstLineConstr;
    for( int i=0; i<constr.size(); i++ ) {
      insertMethodsFrom(method, className );
      method = (OJMethod)constr.elementAt(i);
      String fLine = (String)fstLine.elementAt(i);
      if( !fLine.equals( "" ) ) {
        String[ ] exprList = parseConstr( fLine );

        StatementList stmListTest = new StatementList();
        for( int j=0; j<exprList.length; j++) {
          Expression expr = PartialParser.makeExpression(method.getEnvironment(), exprList[j]);
          stmListTest.add( new VariableDeclaration( new TypeName(
              expr.getType( method.getEnvironment() ).getName() ), "__constr_variable_name"+j, expr ) );
        }
        OJMethod methodTest = new OJMethod(method.getDeclaringClass(), method.getModifiers(),
                    method.getReturnType(), "__constr_method", method.getParameterTypes(),
                    method.getParameters(), method.getExceptionTypes(), stmListTest);
        insertMethodsFrom( methodTest, className );
        try {
          Vector multiRefered = PreUtils.fieldHandling( stmListTest,
              PreUtils.array2Vector(methodTest.getParameters()), PreUtils.array2Vector( ojFields ) );
          for( int j=0; j<multiRefered.size(); j++ )
            ojGraph.insertFields((OJField)multiRefered.elementAt(j), vUsed );
        } catch( CannotAlterException e ) {
          PreUtils.printExceptionText( "FATAL ERROR: Unexpected Error, in prepare constructor", e );
          System.exit(-1);
        }
      }
    }
  } catch( CannotAlterException e ) {
    PreUtils.printExceptionText("FATAL ERROR: Unexpected error in constructor handling",e);
    System.exit(-1);
  } catch( MOPException e ) {
    PreUtils.printExceptionText("FATAL ERROR: Constructor incorrect ",e);
    System.exit(-1);
  } catch( Exception e ) {
    PreUtils.printExceptionText("FATAL ERROR: Unespected Error Constructor incorrect ",e);
    System.exit(-1);
  }
}

/* insertLateBindingMethods
```

This method find methods, in start class, that override someone else in hierarchy class specified by className. In particulary for each method, in hierarchy class specified by className, it searches once in start class that is equals to him (same parameters, same name).

arguments: className name of class where it search the overridden methods

```
*/
private void insertLateBindingMethods(int className) {
```

```
 Vector myMethods=null;
 String[ ] interfaces=null;
 switch( className ) {
   case CORE:
     myMethods=ojMethodsCore;
     interfaces = ojsCore.getInterfacesStr();
     break;
   case SENDER:
     myMethods=ojMethodsSender;
     interfaces = ojsSender.getInterfacesStr();
     break;
   case RECEIVER:
     myMethods=ojMethodsReceiver;
     interfaces = ojsReceiver.getInterfacesStr();
     break;
 }
 for( int i=0; i<myMethods.size(); i++ )
   for( int j=0; j<ojMethods.size(); j++ )
     if(PreUtils.overriddenMethod((myMethods.elementAt(i)), (ojMethods.elementAt(j))))
       if( ojGraph.insertMethod( (OJMethod)ojMethods.elementAt(j), className ) )
         insertMethodsFrom((OJMethod)ojMethods.elementAt(j), className);
 try {
   for( int i=0; interfaces!=null && i<interfaces.length; i++ ) {
     OJMethod[ ] interMethod = OJClass.forName( interfaces[i] ).getAllMethods();
     for( int j=0; j<interMethod.length; j++ )
       for( int z=0; z<ojMethods.size(); z++ )
         if(PreUtils.overriddenMethod(((OJMethod)ojMethods.elementAt(z)), interMethod[j]))
           if(ojGraph.insertMethod((OJMethod)ojMethods.elementAt(z), className))
             insertMethodsFrom((OJMethod)ojMethods.elementAt(z), className);
   }
 } catch( OJClassNotFoundException e ) {
   PreUtils.printExceptionText("FATAL ERROR: implemented interface not Found",e);
   System.exit(−1);
 } catch( NullPointerException e ) {
   PreUtils.printExceptionText("FATAL ERROR: unexpected error",e);
   System.exit(−1);
 }
}


/* insertMethodsFrom
 ---------------------------------------------------------------------------
 This method insert in graph the starting method and all method called in.
 It does the transitive closure of ojMethod.
 ---------------------------------------------------------------------------
 arguments: ojMethod starting method to find all methods invoked in.
            className name of class that contains starting method
 ---------------------------------------------------------------------------
*/
private void insertMethodsFrom( OJMethod ojMethod, int className ) {
 OJMethod tempOJMethod;
 Stack workMethods = new Stack();
 workMethods.push( ojMethod );

 while( !workMethods.isEmpty() ) {
   OJMethod tempMethod = (OJMethod)workMethods.pop();
   Vector ojmTemp = new Vector();
   Vector allEnvs = new Vector();
   PreUtils.getCalledMethods( tempMethod, ojmTemp, allEnvs );

   for( int i=0; i<ojmTemp.size(); i++ )
     if( ( tempOJMethod = methodCall2OJMethod((MethodCall)ojmTemp.elementAt(i),
```

```
            (Environment)allEnvs.elementAt(i)) ) != null )
        if( ojGraph.insertMethod( tempOJMethod, className ) )
          workMethods.push( tempOJMethod );
  }
}
```

/ methodCall2OJMethod*

This method perform the translation of MethodCall into OJMethod.
This translation is done only if the MethodCall deals a method belong to start class, otherwise it return null.
To do the translation it searchs a OJMethod with same name and same parameters type.

---

arguments: mcMethod MethodCall to translate in OJMethod
environment current Environment
return: OJMethod invoked by specified MethodCall.

**/*

```
public OJMethod methodCall2OJMethod( MethodCall mcMethod, Environment environment ) {
  Expression referExpr;
  OJMethod returnMethod = null;
  String name = mcMethod.getName();
  int i;
  boolean correctMethod = true;

  if( !( (referExpr = mcMethod.getReferenceExpr()) instanceof SelfAccess &&
        ((SelfAccess)referExpr).getAccessType() == SelfAccess.SUPER) ) {
    ExpressionList lexp = mcMethod.getArguments();
    try {
      if( lexp.size() == 0 ) {
        for( i=0; i<ojMethods.size(); i++)
          if( ((OJMethod)ojMethods.elementAt(i)).getParameters().length == 0 &&
              name.equals( (returnMethod = (OJMethod)(ojMethods.elementAt(i))).getName() ) )
            return returnMethod;
      } else {
        OJClass[ ] types = new OJClass[lexp.size()];
        for( i=0; i<lexp.size(); i++) types[i]= lexp.get(i).getType(environment);
        for( i=0; i<ojMethods.size(); i++) {
          OJClass[ ] temp = ((OJMethod)ojMethods.elementAt(i)).getParameterTypes();
          if( name.equals( (returnMethod = (OJMethod)(ojMethods.elementAt(i))).getName() ) &&
               temp.length == types.length ) {
            for( int j=0; j< temp.length; j++ )
              if( !temp[j].equals(types[j]) ) {
                correctMethod = false;
                break;
              }
            if(correctMethod) return returnMethod;
            correctMethod = true;
          }
        }
      }
    } catch( CannotAlterException e ) {
      PreUtils.printExceptionText("Unexpected FATAL ERROR, in MethodCall2OJMethod", e );
      System.exit(-1);
    } catch( Exception e ) {
      PreUtils.printExceptionText("Unexpected generic FATAL ERROR",e);
      System.exit(-1);
    }
  }
  return null;
}
```

```
/* prepareMainClass

 This method delete main class and add three instance of the generated classes.
 It does that to force the classes compiling

*/
private void prepareMainClass() {
 try {
   OJMethod[ ] myMethod = getDeclaredMethods();
   for(int i=0; i<myMethod.length; i++) removeMethod( myMethod[i] );

   OJField[ ] myField = getDeclaredFields();
   for(int i=0; i<myField.length; i++) removeField( myField[i] );

   OJConstructor[ ] myConstructor = getDeclaredConstructors();
   for(int i=0; i<myConstructor.length; i++) removeConstructor( myConstructor[i] );

   if( isDefined[CORE−1] ) {
     OJClass newTypeCore = OJClass.forClass(Object.class);
     OJSystem.env.record(ojsCore.getName(), newTypeCore);
     addField( new OJField(getEnvironment(), this,
                           new FieldDeclaration( new ModifierList(ModifierList.PUBLIC),
                           new TypeName( ojsCore.getName() ),
                           new VariableDeclarator( "Core", null ) ) ) );
   }

   if( isDefined[SENDER−1] ) {
     OJClass newTypeServer = OJClass.forClass(Object.class);
     OJSystem.env.record(ojsSender.getName(), newTypeServer );
     addField( new OJField(getEnvironment(), this,
                           new FieldDeclaration( new ModifierList(ModifierList.PUBLIC),
                           new TypeName( ojsSender.getName() ),
                           new VariableDeclarator( "Sender", null ) ) ) );
   }

   if( isDefined[RECEIVER−1] ) {
     OJClass newTypeReceiver = OJClass.forClass(Object.class);
     OJSystem.env.record(ojsReceiver.getName(), newTypeReceiver );
     addField( new OJField(getEnvironment(), this,
                           new FieldDeclaration( new ModifierList(ModifierList.PUBLIC),
                           new TypeName( ojsReceiver.getName() ),
                           new VariableDeclarator( "Receiver", null ) ) ) );
   }

   setSuperclass( OJClass.forName("java.lang.Object") );
 } catch( OJClassNotFoundException e ) {
   PreUtils.printExceptionText("FATAL ERROR, Object not found",e);
   System.exit(−1);
 } catch( CannotAlterException e ) {
   PreUtils.printExceptionText("FATAL ERROR, unable to modify own class",e);
   System.exit(−1);
 }
}

// This method write on file the code of the three classes if they are defining.

private void printClasses() {
 if( isDefined[CORE−1] ) ojsCore.printOutputClass();
 if( isDefined[SENDER−1] ) ojsSender.printOutputClass();
 if( isDefined[RECEIVER−1] ) ojsReceiver.printOutputClass();
}
```

```
/* translateDefinition
  ------------------------------------------------------------------
  This is the starting point of pre-compiling.
  Initially it checks th syntax, then creates and initializes the output classes, finally it prints
  all three classes and prepare main class.
  All this things are doing using several methods.
  ------------------------------------------------------------------
*/
public void translateDefinition() {
 ExtendedSyntax = (ObjectList)this.getSuffix("kind");
 openClasses();
 prepareClasses();
 printClasses();
 prepareMainClass();
}

/* expandVariableDeclaration
  ------------------------------------------------------------------
  This method is invoked on every VariableDeclaration instance of start class.
  It change type of Declaration into Core Type.
  ------------------------------------------------------------------
  arguments: env current Environment
             var current VariableDeclaration
  ------------------------------------------------------------------
*/
public Statement expandVariableDeclaration( Environment env, VariableDeclaration var ) {
  HandleFirstTime( var.getTypeSpecifier().getName() );
  var.setTypeSpecifier( new TypeName( OJMIclassNames[CORE] ) );
  return var;
}

/* expandTypeName
  ------------------------------------------------------------------
  This method is invoked on every TypeName instance of start class.
  It change type name into Core Type.
  ------------------------------------------------------------------
  arguments: env current Environment
             type current TypeName
  ------------------------------------------------------------------
*/
public TypeName expandTypeName(Environment env, TypeName type) {
  HandleFirstTime( type.getName() );

  if( type.getParent() instanceof AllocationExpression ||
      type.getParent() instanceof VariableDeclaration )
    return type;

  if( type.getName().equals( getName() ) )
      type.setName( OJMIclassNames[CORE] );

  return type;
}

/* expandAllocation
  ------------------------------------------------------------------
  This method is invoked on every AllocationExpression instance of start class.
  It search the constructor and change it into a new constructor of Core class.
  To do that change name of current constructor and add it three parameters, KIND of Channel
  name of SENDER and RECEIVER.
  ------------------------------------------------------------------
  arguments: env current Environment
             expr current AllocationExpression
```

```
*/
public Expression expandAllocation( Environment env, AllocationExpression expr ) {
  HandleFirstTime( expr.getClassType().getName() );
  if ( expr.getClassType().getName().equals( getName() ) ) {
    expr.setClassType( new TypeName( OJMIclassNames[CORE] ) );
    ExpressionList paramList = new ExpressionList();
    paramList.add( new Literal( Literal.STRING, "\""+OJMIclassNames[KIND]+"\"" ) );
    paramList.addAll( expr.getArguments() );
    paramList.add( new Literal( Literal.STRING, "\""+OJMIclassNames[SENDER]+"\"" ) );
    paramList.add( new Literal( Literal.STRING, "\""+OJMIclassNames[RECEIVER]+"\"" ) );
    expr.setArguments( paramList );
  }
  return expr;
}

/* expandMethodCall
```

This method is invoked on every MethodCall instance of start class not tested yet.
It check if a method is static, if so it translate it, otherwise don't make anything.
To translate calling changeReferenceExpr

---

arguments: env current Environment
call current MethodCall

```
*/
public Expression expandMethodCall( Environment env, MethodCall call ) {
  if( call.getReferenceType() == null ) return call;
  HandleFirstTime( call.getReferenceType().getName() );
  boolean find = false;
  OJMethod method = methodCall2OJMethod( call, env );
  if( coreMethods != null ) {
    for( int i=0; i<coreMethods.length ; i++ )
      if( coreMethods[i].equals( method ) ) {
        if( coreMethods[i].getModifiers().isStatic() )
          changeReferenceExpr( call, env );
        find = true;
        break;
      }
    if( find == false ) {
      System.err.println("FATAL ERROR: Wrong Method Access ");
      System.exit(-1);
    }
  }
  return call;
}

/* HandleFirstTime
```

This method takes the info in OJMI file of sterting class.
This method is performs only once.

---

arguments: expr string representig the Core name

```
*/
public void HandleFirstTime( String expr ) {
 if ( firstTime ) {
   try {
     OJClass CoreClass = OJClass.forName( expr );
     coreMethods = CoreClass.getAllMethods();
     OJMIclassNames[KIND] = CoreClass.getMetaInfo("Kind");
     OJMIclassNames[CORE] = CoreClass.getMetaInfo("Core");
     OJMIclassNames[SENDER] = CoreClass.getMetaInfo("Sender");
```

```
      OJMIclassNames[RECEIVER] = CoreClass.getMetaInfo("Receiver");
      firstTime = false;
    } catch( OJClassNotFoundException e ) {
      PreUtils.printExceptionText("FATAL ERROR: Unable to find Core Class ", e);
      System.exit(-1);
    }
  }
}

/* changeReferenceExpr
```

This method do a recursive parsing to find typeName to change. When find it delete it, because an instance of class can access to static member.

arguments: call Expression
env current Environment

```
*/
private void changeReferenceExpr( Expression call, Environment env ) {
  if( call instanceof MethodCall ) {
    Expression RefExpr = ((MethodCall)call).getReferenceExpr();
    TypeName RefType = ((MethodCall)call).getReferenceType();
    if( RefExpr != null ) changeReferenceExpr( RefExpr, env );
    else if( !(call instanceof ArrayAccess) && RefExpr != null
             && RefType.getName().equals(OJMIclassNames[CORE]) )
      ((MethodCall)call).setReferenceType( new TypeName(getMetaInfo("Core")) );
  }

  if( call instanceof FieldAccess ) {
    Expression RefExpr = ((FieldAccess)call).getReferenceExpr();
    TypeName RefType = ((FieldAccess)call).getReferenceType();
    if( RefExpr != null ) changeReferenceExpr( RefExpr, env );
    else if( !(call instanceof ArrayAccess) && RefExpr != null
             && RefType.getName().equals(OJMIclassNames[CORE]) )
      ((FieldAccess)call).setReferenceType( new TypeName(getMetaInfo("Core")) );
  }

  if( call instanceof ArrayAccess ) {
    Expression RefExpr = ((ArrayAccess)call).getReferenceExpr();
    if( RefExpr != null ) changeReferenceExpr( RefExpr, env );
  }
}

}
```

Bibliography

[1] Richard M. Adler. Distributed Coordination Models for Client|Server Computing. *IEEE Transactions on Computers*, pages 14–22, April 1995.

[2] Mehmet Aksit, Ken Wakita, Jan Bosch, Lodewijk Bergmans, and Akinori Yonezawa. Abstracting Object Interactions Using Composition Filters. In *Proceedings of Object-Based Distributed Programming (ECOOP'94 Workshop)*, Lecture Notes in Computer Science 791, pages 152–184. Springer-Verlag, July 1994.

[3] Jonathan Aldrich, James Dooley, Scott Mandelsohn, and Adam Rifkin. Providing Easier Access to Remote Objects in Client-Server Systems. In *Proceedings of the 31th Hawaii International Conference on System Sciences*, January 1998.

[4] Massimo Ancona, Walter Cazzola, Gabriella Dodero, and Vittoria Gianuzzi. Channel Reification: a Reflective Approach to Fault-Tolerant Software Development. In *OOPSLA'95 (poster section)*, page 137, Austin, Texas, USA, on 15th-19th October 1995. ACM. Available at `http://www.disi.unige.it/person/CazzolaW/references.html`.

[5] Massimo Ancona, Walter Cazzola, Gabriella Dodero, and Vittoria Gianuzzi. Channel Reification: A Reflective Model for Distributed Computation. In *Proceedings of IEEE International Performance Computing, and Communication Conference (IPCCC'98)*, 98CH36191, pages 32–36, Phoenix, Arizona, USA, on 16th-18th February 1998. IEEE.

[6] Massimo Ancona, Walter Cazzola, and Eduardo B. Fernandez. Reflective Authorization Systems. In *Proceedings of ECOOP Workshop on Distributed Object Security (EWDOS'98)*, in 12th European Conference on Object-Oriented Programming (ECOOP'98), pages 35–39, Brussels, Belgium, on 20th-24th July 1998. Unité de Recherche INRIA Rhône-Alpes.

[7] Massimo Ancona, Walter Cazzola, and Eduardo B. Fernandez. A History-Dependent Access Control Mechanism Using Reflection. In *Proceedings of 5th ECOOP Workshop on Mobile Object Systems (EWMOS'99)*, in 13th European Conference on Object-Oriented Programming (ECOOP'99), Lisbon, Portugal, on 14th-18th June 1999.

[8] Massimo Ancona, Walter Cazzola, and Eduardo B. Fernandez. Reflective Authorization Systems: Possibilities, Benefits and Drawbacks. In Jan Vitek and Christian Jensen, editors, *Secure Internet Programming: Security Issues for Mobile and Distributed Objects*, Lecture Notes in Computer Science 1603, pages 35–51. Springer-Verlag, July 1999.

[9] Ken Arnold and James Gosling. *The Java Programming Language*. The Java Series ... from the Source. Addison-Wesley, Reading, Massachusetts, second edition, December 1997.

[10] Jean-Serge Banino, Jean-Charles Fabre, M. Guillemont, Gérard Morisset, and Marc Rozier. Some Fault-Tolerant Aspects of the CHORUS Distributed Systems. In *Prooceedings of IEEE 5th International Conference on Distributed Computing Systems*, pages 430–437, May 1985.

[11] Philip A. Bernstein and Nathan Goodman. Concurrency Control in Distributed Database Systems. *ACM Computing Surveys*, 13(2):185–221, June 1981.

[12] Andrew D. Birrell and Bruce Jay Nelson. Implementing Remote Procedure Calls. *ACM Transactions on Computer Systems*, 2(1):39–59, February 1984.

[13] Andrew P. Black, Norman C. Hutchinson, Eric Jul, and Henry M. Levy. Object Structure in the Emerald System. In *Prooceedings of the ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'86)*, number 21(11) in Sigplan Notices, pages 78–86, November 1986.

[14] Andrew P. Black, Norman C. Hutchinson, Eric Jul, Henry M. Levy, and Larry Carter. Distribution and Abstract Types in Emerald. *IEEE Transactions on Software Engineering*, 13(1):65–76, January 1987.

[15] Gordon Blair, Fábio Costa, Geoff Coulson, Fabien Delpiano, Hector Duran, Bruno Dumant, François Horn, Nikos Parlavantzas, and Jean-Bernard Stefani. The Design of a Resource-Aware Reflective Middleware Architecture. In Pierre Cointe, editor, *Proceedings of the 2nd International Conference on Reflection'99*, LNCS 1616, pages 115–134, Saint-Malo, France, July 1999. Springer-Verlag.

[16] Daniel G. Bobrow, R.G. Gabriel, and J.L. White. *CLOS in Context - The Shape of the Design Space. In Object Oriented Programming - The CLOS Perspective*. MIT Press, 1993.

[17] Forrest D. Brewer and Jonathan M. Nash. The Chinese Wall Security Policy. *IEEE Symposium on Security and Privacy*, pages 215–228, 1989.

[18] Jean-Pierre Briot and Pierre Cointe. Programming with Explicit Metaclasses in SmallTalk-80. In USA Portland, Oregon, editor, *Proceedings of OOPSLA'89*, volume 24(10) of *Sigplan Notices*, pages 419–431. ACM, October 1989.

[19] Jean-Pierre Briot, Rachid Guerraoui, and Klaus-Peter Löhr. Concurrency and Distribution in Object-Oriented Programming. *ACM Computing Surveys*, 30(3):291–329, September 1998.

[20] Walter Cazzola. Channel Reification: a New Reflective Model. Analysis and Comparison with Other Models and Application to Fault Tolerant System. Master's thesis, University of Genova – Department of Computer Science (DISI), April 1996. (Written in Italian).

[21] Walter Cazzola. Evaluation of Object-Oriented Reflective Models. In *Proceedings of ECOOP Workshop on Reflective Object-Oriented Programming and Systems (EWROOPS'98)*, in 12th European Conference on Object-Oriented Programming (ECOOP'98), Brussels, Belgium, on 20th-24th July 1998. Extended Abstract also published on ECOOP'98 Workshop Readers, S. Demeyer and J. Bosch editors, LNCS 1543, ISBN 3-540-65460-7 pages 386-387.

[22] Walter Cazzola and Massimo Ancona. mChaRM: a Reflective Middleware for Communication-Based Reflection. Technical Report DISI-TR-00-09, DISI, University degli Studi di Genova, May 2000. Available at `http://www.disi.unige.it/person/CazzolaW/references.html`.

[23] Walter Cazzola, Andrea Sosio, and Francesco Tisato. Reflection and Object-Oriented Analysis. In Walter Cazzola, Robert J. Stroud, and Francesco Tisato, editors, *Proceedings of the 1st Workshop on Object-Oriented Reflection and Software Engineering (OORaSE'99)*, pages 95–106. University of Milano Bicocca, November 1999.

[24] Walter Cazzola, Andrea Sosio, and Francesco Tisato. Shifting Up Reflection from the Implementation to the Analysis Level. In Walter Cazzola, Robert J. Stroud, and Francesco Tisato, editors, *Reflection and Software Engineering*, Lecture Notes in Computer Science 1826, pages 1–20. Springer-Verlag, Heidelberg, Germany, June 2000.

[25] Shigeru Chiba. OpenC++ Release 1.2 Programmer's Guide. Technical Report 93-3, Department of Information Science, University of Tokyo, 1993.

[26] Shigeru Chiba. A Meta-Object Protocol for C++. In *Proceedings of the 10th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'95)*, volume 30 of *Sigplan Notices*, pages 285–299, Austin, Texas, USA, October 1995. ACM.

[27] Shigeru Chiba and Takashi Masuda. Designing an Extendible Distributed Language with a Meta-Level Architecture. In Oscar M. Nierstrasz, editor, *Proceedings of 7th European Conference for Object-Oriented Programming (ECOOP'93)*, LNCS 707, pages 482–501, Kaiserslautern, Germany, July 1993. Springer-Verlag.

[28] Shigeru Chiba, Michiaki Tatsubori, Marc-Olivier Killijian, and Kozo Itano. OpenJava: A Class-based Macro System for Java. In Walter Cazzola, Robert J. Stroud, and Francesco Tisato, editors, *Reflection and Software Engineering*, Lecture Notes in Computer Science 1826, pages 119–135. Springer-Verlag, Heidelberg, Germany, June 2000.

[29] Roger S. Chin and Samuel T. Chanson. Distribuited Object-Based Programming Systems. *ACM Computing Surveys*, 23(1):91–124, March 1991.

[30] Pierre Cointe. MetaClasses are first class objects: the ObjVLisp model. In Norman K. Meyrowitz, editor, *Proceedings of the 2nd Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'87)*, volume 22(10) of *Sigplan Notices*, Orlando, Florida, USA, October 1987. ACM.

[31] Fábio M. Costa, Hector A. Duran, Nikos Parlavantzas, Katia B. Saikoski, Gordon Blair, and Geoff Coulson. The Role of Reflective Middleware in Supporting the Engineering of Dynamic Applications. In Walter Cazzola, Robert J. Stroud, and Francesco Tisato, editors, *Reflection and Software Engineering*, Lecture Notes in Computer Science 1826, pages 79–99. Springer-Verlag, Heidelberg, Germany, June 2000.

[32] François-Nicola Demers and Jacques Malenfant. Reflection in Logic, Functional and Object-Oriented Programming: a Short Comparative Study. In *Proceedings of the IJCAI'95 Workshop on Reflection and Metalevel Architectures and their Applications in AI*, pages 29–38, Montréal, Canada, August 1995.

[33] Jim Dowling, Tilman Schäfer, Vinny Cahill, Peter Haraszti, and Barry Redmond. Using Reflection to Support Dynamic Adaptation of System Software: A Case Study Driven Evaluation. In Walter Cazzola, Robert J. Stroud, and Francesco Tisato, editors, *Reflection and Software Engineering*, Lecture Notes in Computer Science 1826, pages 171–190. Springer-Verlag, Heidelberg, Germany, June 2000.

[34] Carolyn K. Duby, Scott Meyers, and Steven P. Reiss. CCEL: A Metalanguage for C++. Technical Report 02912 CS-92-51, Department of Computer Science Brown University, Providence, Rhode Island, October 1992.

[35] Stéphane Ducasse. Evaluating Message Passing Control Techniques in SmallTalk. *Journal of Object-Oriented Programming (JOOP)*, November 1998.

[36] Jean-Charles Fabre, Vincent Nicomette, Tanguy Pérennou, Robert J. Stroud, and Zhixue Wu. Implementing Fault Tolerant Applications Using Reflective Object-Oriented Programming. In *Proceedings of FTCS-25 "Silver Jubilee"*, Pasadena, CA USA, June 1995. IEEE.

[37] Mohamed E. Fayad and Rachid Guerraoui. OO Distributed Programming Is Not Distributed OO Programming. *Communications of the ACM*, 42(4):101–104, April 1999.

[38] Jacques Ferber. Computational Reflection in Class Based Object Oriented Languages. In *Proceedings of 4th Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'89)*, volume 24 of *Sigplan Notices*, pages 317–326. ACM, October 1989.

[39] Eduardo B. Fernandez, Rita C. Summers, and Christopher Wood. *Database Security and Integrity*. Addison-Wesley, Reading, Massachusetts, 1981.

[40] Brian Foote and Ralph E. Johnson. Reflective Facilities in SmallTalk-80. In Norman K. Meyrowitz, editor, *Proceedings of the 4th Conference on Object-Oriented Programming: Systems, Languages, and Applications (OOPSLA'89)*, volume 24(10) of *Sigplan Notices*. ACM, October 1989.

[41] Open Software Foundation. Introduction to OSF DCE. Technical report, Open Software Foundation, Cambridge, USA, 1992.

[42] Benoît Garbinato, Rachid Guerraoui, and Karim R. Mazouni. Distributed Programming GARF. In Rachid Guerraoui, Oscar Nierstrasz, and Michel Riveil, editors, *Object-Based Distributed Programming*, LNCS 901, pages 1–32. Springer-Verlag, 1994.

[43] Steffen Görzig. C++ Interface to PVM. Technical report, University of Stüttgart, Germany, 1999.

[44] Brendan Gowing and Vinny Cahill. Making Meta-Object Protocols Pratical for Operating Systems. In *Proceedings of 4th International Workshop on Object Oriented in Operating Systems*, pages 52–55, April 1995.

[45] Brendan Gowing and Vinny Cahill. Meta-Object Protocols for C++: The Iguana Approach. In *Proceedings of Reflection'96*, April 1996.

[46] Nicolas Graube. Metaclass Compatibility. In Norman K. Meyrowitz, editor, *Proceedings of the 4th Conference on Object-Oriented Programming: Systems, Languages, and Applications (OOPSLA'89)*, volume 24(10) of *Sigplan Notices*, pages 305–316, New Orleans, Louisiana, USA, October 1989. ACM.

[47] Object Management Group. Common Object Request Broker Architecture and Specification. Technical Report 96.3.4 Revision 2.0, OMG, 1995.

[48] Object Management Group. The Common Object Request Broker: Architecture and Specification. formal-document 98-07-01 Revision 2.2, OMG, February 1998.

[49] Rachid Guerraoui, Benoît Garbinato, and Karim R. Mazouni. GARF: A Tool for Programming Reliable Distributed Applications. *IEEE Concurrency*, 5(4), October-December 1997.

[50] Walter Hürsch and Cristina Videira Lopes. Separation of Concerns. Technical Report NU-CCS-95-03, Northeastern University, Boston, February 1995.

[51] Gregor Kiczales, Jim des Rivières, and Daniel G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, Cambridge, Massachusetts, 1991.

[52] Gregor Kiczales, John Lamping, and Shigeru Chiba. Avoiding Confusion in Metacircularity: The Meta-Helix. In *Proceedings of the 2nd International Symposium on Object Technologies for Advanced Software (ISOTAS'96)*, LNCS 1049, pages 157–172, Kanazawa, Japan, March 1996. Springer Verlag.

[53] Fabio Kon, Roy Campbell, and Manuel Román. Design and Implementation of Runtime Reflection in Communication Middleware: the DynamicTAO Case. In *Proceedings of ICDCS'99 Workshop on Middleware*, 1999.

[54] Fabio Kon, Manuel Román, Ping Liu, Jina Mao, Tomonori Yamane, Luiz Claudio Magalhães, and Roy H. Campbell. Monitoring, Security, and Dynamic Configuration with the dynamicTAO Reflective ORB. In *Proceedings of IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'2000)*, volume 30, New York, USA, April 2000.

[55] John Lamping, Gregor Kiczales, Luis H. Rodriguez Jr, and Erik Ruf. An Architecture for an Open Compiler. In Akinori Yonezawa and Brian C. Smith, editors, *Proceedings of the Int'l Workshop on Reflection and Meta-Level Architecture*, pages 95–106, 1992.

[56] Thomas Ledoux. *Réflexion dans les Systèmes Répartis: application à CORBA et SmallTalk*. PhD thesis, Université de Nantes, École des Mines de Nantes, Nantes, France, March 1998.

[57] Thomas Ledoux. OpenCorba: A Reflective Open Broker. In Pierre Cointe, editor, *Proceedings of the 2nd International Conference on Reflection'99*, LNCS 1616, pages 197–214, Saint-Malo, France, July 1999. Springer-Verlag.

[58] Arthur H. Lee and Joseph L. Zachary. Using Meta Programming to Add Persistence to CLOS. In *International Conference on Computer Languages*, Los Alamitos, California, 1994. IEEE.

[59] Barbara Liskov. Distributed Programming in Argus. *Communications of the ACM*, 31(3):300–312, March 1988.

[60] Orlando Loques, Julius Leite, Marcelo Lobosco, and Alexandre Sztajnberg. Integrating Meta-Level Programming and Configuration Programming. In Walter Cazzola, Robert J. Stroud, and Francesco Tisato, editors, *Proceedings of the 1st Workshop on Object-Oriented Reflection and Software Engineering (OORaSE'99)*, pages 137–151. University of Milano Bicocca, November 1999.

[61] Orlando Loques, Alexandre Sztajnberg, Julius Leite, and Marcelo Lobosco. On the Integration of Configuration and Meta-Level Programming Approaches. In Walter Cazzola, Robert J. Stroud, and Francesco Tisato, editors, *Reflection and Software Engineering*, Lecture Notes in Computer Science 1826, pages 191–210. Springer-Verlag, Heidelberg, Germany, June 2000.

[62] Pattie Maes. *Computational Reflection*. PhD thesis, Artificial Intelligence Laboratory, Vrije Universiteit, Brussel, Belgium, 1987.

[63] Pattie Maes. Concepts and Experiments in Computational Reflection. In Norman K. Meyrowitz, editor, *Proceedings of the 2nd Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'87)*, volume 22 of *Sigplan Notices*, pages 147–156, Orlando, Florida, USA, October 1987. ACM.

[64] Hidehiko Masuhara, Satoshi Matsuoka, Takuo Watanabe, and Akinori Yonezawa. Object-Oriented Concurrent Reflective Languages Can Be Implemented Efficiently. In Andreas Pæpcke, editor, *Proceedings of 7th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'92)*, volume 27(10) of *Sigplan Notices*, pages 127–144, Vancouver, British Columbia, Canada, October 1992. ACM.

[65] Jeff McAffer. The CodA MOP. In *Proceedings of the 8th Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'93)*, Workshop on Object-Oriented Reflection and Metalevel Architectures, Washington, DC, USA, 1993. ACM.

[66] Jeff McAffer. Meta-Level Programming with CodA. In Walter Olthoff, editor, *Proceedings of the 9th European Conference on Object-Oriented Programming (ECOOP'95)*, LNCS 952, pages 190–214. Springer-Verlag, 1995.

[67] Todd Montgomery. RMP - Reliable Multicast Protocol version 2. Document Available from `http://research.ivv.nasa.gov/projects/RMP/index.html`, October 1996.

[68] Louise E. Moser, Michael P. Melliar-Smith, Deborah A. Agarwal, Ravi K. Budhia, and Colleen A. Lingley-Papadopoulos. **Totem**: A Fault-Tolerant Multicast Group Communication System. *Communications of the ACM*, 39(4):54–63, April 1996.

[69] Sape J. Mullender and Andrew S. Tanenbaum. The Design of a Capability-Based Distributed Operating System. *The Computer Journal*, 29(4):289–300, March 1986.

[70] Hideaki Okamura and Yutaka Ishikawa. Object Location Control Using Meta-level Programming. In Mario Tokoro and Remo Pareschi, editors, *Proceedings of the 8th European Conference on Object-Oriented Programming (ECOOP'94)*, LNCS 821, pages 299–319, Bologna, Italy, July 1994. Springer-Verlag.

[71] Nikos Parlavantzas, Geoff Coulson, Mike Clarke, and Gordon Blair. Towards a Reflective Component Based Middleware Architecture. In Walter Cazzola, Shigeru Chiba, and Thomas Ledoux, editors, *On-Line Proceedings of ECOOP'2000 Workshop on Reflection and Metalevel Architectures*, June 2000. Available at `http://www.disi.unige.it/RMA2000`.

[72] Graham D. Parrington, Santosh K. Shrivastava, Stuart M. Wheater, and Mark C. Little. The Design and Implementation of Arjuna. *USENIX Computing Systems Journal*, 8(3):255–308, July 1995.

[73] David Powell, Marc Chérèque, and David Drackley. Fault-Tolerance in Delta-4. *Operating Systems Review*, 25(2):122–125, 1991.

[74] Ramana Rao. Implementational Reflection in Silica. In Pierre America, editor, *Proceedings of ECOOP'91*, pages 251–266, Switzerland, July 1991. Springer-Verlag.

[75] Thomas Riechmann and Jürgen Kleinöder. Meta-Objects for Access Control: Role-Based Principals. In Colin Boyd and Ed Dawson, editors, *Lecture Notes in Computer Science*, number 1438 in Proceedings of 3rd Australasian Conference on Information Security and Privacy (ACISP'98), pages 296–307, Brisbane, Australia, July 1998. Springer-Verlag.

[76] Frédéric Rivard. *Evolution du Comportement dans les Langages Réflexifs Dynamiquement Typés*. PhD thesis, Université de Nantes, 1997.

[77] Ravi Sandhu. Lattice-Based Enforcement of Chinese Walls. *Computers and Security*, 11(8):753–763, December 1992.

[78] Mark Shapiro. Structure and Encapsulation in Distributed Systems: The Proxy Principle. In *Proceedings of the 6th International Conference on Distributed Computer Systems*, pages 198–204, Cambridge, MA, May 1986.

[79] Brian C. Smith. Reflection and Semantics in a Procedural Language. Technical Report 272, MIT Laboratory of Computer Science, 1982.

[80] Alfred Z. Spector, Joshua J. Bloch, Dean S. Daniels, Richard Draves, Dan Duchamp, Jeffrey L. Eppinger, Sherri G. Menees, and Dean S. Thompson. The Camelot Project. *Database Engineering Bulletin*, 9(3):23–34, 1986.

[81] Raj Srinivasan. RPC: Remote Procedure Call Protocol Specification Version 2. RFC 1831, SUN Microsystems, August 1995.

[82] Christoph Steindl. Reflection in Oberon. Technical Report CS-SSW-P96-11, Department of Computer Science (System Software), Johannes Kepler University, Linz, Austria, November 1996.

[83] Robert J. Stroud. Transparency and Reflection in Distributed Systems. *ACM Operating System Review*, 22:99–103, April 1992.

[84] Robert J. Stroud and Ian Welch. Dynamic Adaptation of the Security Properties of Application and Components. In *Proceedings of ECOOP Workshop on Distributed Object Security (EWDOS'98)*, in 12th European Conference on Object-Oriented Programming (ECOOP'98), pages 41–46, Brussels, Belgium, July 1998. Unité de Recherche INRIA Rhǒne-Alpes.

[85] Robert J. Stroud and Zhixue Wu. Using Meta-Object Protocol to Implement Atomic Data Types. In Walter Olthoff, editor, *Proceedings of the 9th Conference on Object-Oriented Programming (ECOOP'95)*, LNCS 952, pages 168–189, Aarhus, Denmark, August 1995. Springer-Verlag.

[86] Robert J. Stroud and Zhixue Wu. Using Metaobject Protocols to Satisfy Non-Functional Requirements. In Chris Zimmerman, editor, *Advances in Object-Oriented Metalevel Architectures and Reflection*, chapter 3, pages 31–52. CRC Press, Inc., 2000 Corporate Blvd.,N.W., Boca Raton, Florida 33431, 1996.

[87] SUN Microsystems. Java™ Core Reflection API and Specification. Technical report, SUN Microsystems, February 1997.

[88] SUN Microsystems. Java™ Remote Method Invocation - Distributed Computing for Java. White paper, SUN Microsystems, 1998. Internet Publication - `http://www.sun.com`.

[89] Alexandre Sztajnberg and Orlando Loques. Reflection in the R-RIO Configuration Programming Environment. In Gordon Blair and Roy Campbell, editors, *On Line Proceedings of the Workshop on Reflective Middleware*, New York, USA, April 2000. Available on-line at `http://www.comp.lancs.ac.uk/computing/RM2000/`.

[90] Andrew S. Tanenbaum. *Distributed Operating Systems*. Prantice-Hall, 1995.

[91] Michiaki Tatsubori. An Extension Mechanism for the Java Language. Master of engineering dissertation, Graduate School of Engineering, University of Tsukuba, University of Tsukuba, Ibaraki, Japan, February 1999.

[92] Julien Vayssière. Security and Meta Programming in Java. In Walter Cazzola, Shigeru Chiba, and Thomas Ledoux, editors, *On-Line Proceedings of ECOOP'2000 Workshop on Reflection and Metalevel Architectures*, June 2000. Available at `http://www.disi.unige.it/RMA2000`.

[93] Takuo Watanabe, Amano Noriki, and Kenji Shinbori. A Reflective Framework for Reliable Mobile Agent Systems. In Walter Cazzola, Shigeru Chiba, and Thomas Ledoux, editors, *On-Line Proceedings of ECOOP'2000 Workshop on Reflection and Metalevel Architectures*, June 2000. Available at `http://www.disi.unige.it/RMA2000`.

[94] Takuo Watanabe and Akinori Yonezawa. An Actor-Based Metalevel Architecture for Group-Wider Reflection. In J. W. de Bakker, W. P. de Rover, and G. Rozenberg, editors, *Foundations of Object-Oriented Languages*, pages 405–425. Springer-Verlag, 1990.

[95] Brian Whetten, Todd Montgomery, and Simon Kaplan. A High Performance Totally Ordered Multicast Protocol. In Kenneth P. Birman, Friedemann Mattern, and André Schiper, editors, *Theory and Practice in Distributed Systems*, LNCS 938, pages 33–57. Springer-Verlag, Berlin, Heidelberg, April 1995.

[96] Yasuhiko Yokote. The ApertOS Reflective Operating System: The Concept and Its Implementation. In Andreas Pæpcke, editor, *Proceedings of the 7th Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'92)*, volume 27(10) of *Sigplan Notices*, pages 414–434, Vancouver, British Columbia, Canada, October 1992. ACM.