Distrib. Comput. (2003) 16: 287-306
Digital Object Identifier (DOI) 10.1007/s00446-003-0094-8

Remote Method Invocation as a First-Class Citizen

Walter Cazzola

DICo - University degli Studi di Milano, Milano, Italy (e-mail: cazzola@dico.unimi.it)

Received: December 2001 / Accepted: April 2003

Abstract. The classical *remote method invocation* (RMI) mechanism adopted by several object-based middleware is 'black box' in nature, and the RMI functionality, i.e., the RMI interaction policy and its configuration, is hard-coded into the application. This RMI nature hinders software development and reuse, forcing the programmer to focus on communication details often marginal to the application he is developing. Extending the RMI behavior with extra functionality is also a very difficult job, because added code must be scattered among the entities involved in communications.

This situation could be improved by developing the system in several separate layers, confining communications and related matters to specific layers. As demonstrated by recent work on reflective middleware, reflection represents a powerful tool for realizing such a separation and therefore overcoming the problems referred to above. Such an approach improves the separation of concerns between the communication-related algorithms and the functional aspects of an application. However, communications and all related concerns are not managed as a single unit separate from the rest of the application, which makes their reuse, extension and management difficult. As a consequence, communications concerns continue to be scattered across the meta-program, communication mechanisms continue to be black-box in nature, and there is only limited opportunity to adjust communication policies through configuration interfaces.

In this paper we examine the issues raised above, and propose a reflective approach especially designed to open up the Java RMI mechanism. Our proposal consists of a new reflective model, called *multi-channel reification*, that reflects on and reifies communication channels, i.e., it renders communication channels first-class citizens. This model is designed both for developing new communication mechanisms and for extending the behavior of communication mechanisms provided by the underlying system. Our approach is embodied in a framework called mChaRM which is described in detail in this paper.

Correspondence to: Walter Cazzola, c|o DICo, University of Milano, Via Comelico 39/41, 20135, Milano, Italy. Fax +39-02-503-16000 (e-mail: cazzola@dico.unimi.it)

Key words: Reflection – Reflective Model – Reflective Middleware – Java RMI.

1 Introduction

The term *middleware* refers to a set of services that resides between the application and the operating system and aims to facilitate the development, deployment, and management of distributed applications [15]. The main objective of distributed middleware is to provide a convenient environment for the realization of distributed computations. Unfortunately, in many middleware interaction policies between the distributed objects are hard-coded into the platform itself. Some platforms, e.g., CORBA [39], provide mechanisms, such as *interceptors* and *POA/Servant Manager*, for redefining interaction details, but these allow for customization only within the scope envisaged by their designers. Full *adaptability* has not been achieved yet.

Another problem occurs because the monolithic nature of middleware forces distributed algorithms to be implemented at the application level. This results in an intertwining of distributed algorithm code with application code, does not achieving the *separation of concerns* [25] between functional and nonfunctional code. Some programming languages, e.g., Java [5], disguise remote interactions as local calls, thus rendering their presence transparent to the programmer. However their management, (i.e., tuning, and synchronizing the involved objects) is not so transparent and easily maskable to the programmer. Moreover, distributed algorithms are scattered among several objects, the complexity of these algorithms is augmented by introducing nonfunctional code for coordinating the work of the involved objects and accessing to remote data. We can summarize these kinds of problems with current middleware platforms as follows:

- ❶ interaction policies are hidden from the programmer who can not customize them (*lack of adaptability*);
- ❷ communication, synchronization, and tuning code is intertwined with application code (*lack of separation of concerns*);

❸ algorithms are scattered among several objects, thus forcing the programmer to explicitly coordinate their work (*lack of global view*).

Global view, adaptability, and separation of concerns are requirements that current middleware do not completely address and support. To address such open issues, classic middleware has been enhanced with concepts from computational reflection to create *reflective middleware* [7, 15].

The rest of this paper is organized as follows. Section 2 describes the reflective middleware approach and why it fails to deal with these open issues. Section 3 relates our communication-oriented reflective model, called the *multi-channel reification model*. Sections 4 and 5 are devoted to showing the applicability of our solution, describing the mChaRM framework based on our model, which opens up the Java RMI mechanism and its implementation. In section 6, we also show a few non-trivial applications using the proposed approach. The final three sections draw conclusions and present some related and future work.

2 Reflective Middleware

2.1 Computational Reflection.

Computational reflection (or *reflection* for short) is defined as the activity performed by an agent when doing computations about itself [32]. This activity involves two aspects: *introspection* and *intercession*. Bobrow et al. [8] define these two terms as follows:

> *Introspection is the ability of 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.*

Reflection applies quite naturally to the object-oriented paradigm [17, 19, 32]. Just as objects in the conventional object-oriented paradigm are representations of *real world* entities, objects can themselves be represented by other objects, usually referred to as *meta-objects*. Computation done by meta-objects (*meta-computation*) is for the purpose of observing and modifying the objects they represent, called *referents*. Meta-computation is often performed by meta-objects by *trapping* the normal computation of their referents. In other words, an action of the referent is trapped by the meta-object, which performs a meta-computation either replacing or encapsulating the referent's action. Of course, meta-objects themselves can be manipulated by meta-meta-objects, and so on. Thus, a reflective system can be structured in multiple levels, constituting a *reflective tower*. Base-level objects (termed *base-objects*) perform computations on the entities of the application domain. Objects in the other levels (termed *meta-levels*) perform computations on the objects residing in the lower levels. The interface between adjacent levels in the reflective tower is usually termed as *meta-object protocol* (MOP) [27].

Reification is an essential capability of all reflective models. Each level of the reflective tower maintains a set of data structures representing (*reifying*) lower level computation. Of course, which aspects are reified depends on the reflective model (e.g., structure, state and behavior, communication). In any case, the data structures comprising a reification must be *causally connected* to the aspect(s) of the system being reified. All changes to the reification are reflected in the system, and vice versa. Depending on the reflective model, the causal connection may operate at compile-, load- or run-time, but in all cases the meta-object programmer is not concerned about how the causal connection is achieved.

Transparency [41] is another key feature of all reflective models. In the context of reflection, transparency refers to the fact that the objects in each level are completely *unaware* of the presence and workings of objects in higher levels. In other words, each meta-level is added to the base-level without modifying the referent level itself. An important application of transparency is in the separation of functional features from (possibly several distinct) nonfunctional features. In a typical approach, objects at the base-level are entrusted to meet an application's functional requirements, while those at the meta-level add nonfunctional properties (e.g., fault tolerance, persistence, distribution, and so on). Software systems can benefit from such an approach for several reasons (e.g., easy adaptability, separation of concerns, and code stability). Of course, *separation of concerns* enhances the system's modifiability. Depending on whether a required modification to the system involves functional or nonfunctional properties, functional objects alone or nonfunctional objects alone may be modified.

2.2 The Reflective Middleware Approach.

A reflective approach, as stated in [9], can be considered as the glue to stick together distributed and object-based programming and to fill gaps in their integration. Reflection provides a programming environment that exposes the implementation details of a system, i.e., the interaction policies, and allows the programmer to manipulate them. Moreover, a reflective approach permits easy separation of interaction management code from application code. Due to such considerations many reflective middleware implementations have been developed. Current research in reflective middleware is focusing on improving customizability and rendering its use more transparent. However, most reflective middleware approaches leave two open issues:

❶ the system lacks a global view, and
❷ the independent customization of each individual remote method invocation or message exchange[1] is difficult.

The above problems, as stated by Kenneth Birman in his keynote at Middleware 2000, are very important from the development point of view because, their resolution would lead to an increased reuse of communication-based features and simpler implementations.

Global view. Writing object-based distributed applications in which several separate entities manage shared information is

[1] In the rest of this paper, notwithstanding the term hides the return values intrinsic in the (remote) method invocation, we use the term *message* to denote both the (remote) method invocation and the exchanging of data because this terminology adheres to the terminology previously adopted in the reflection community [19].

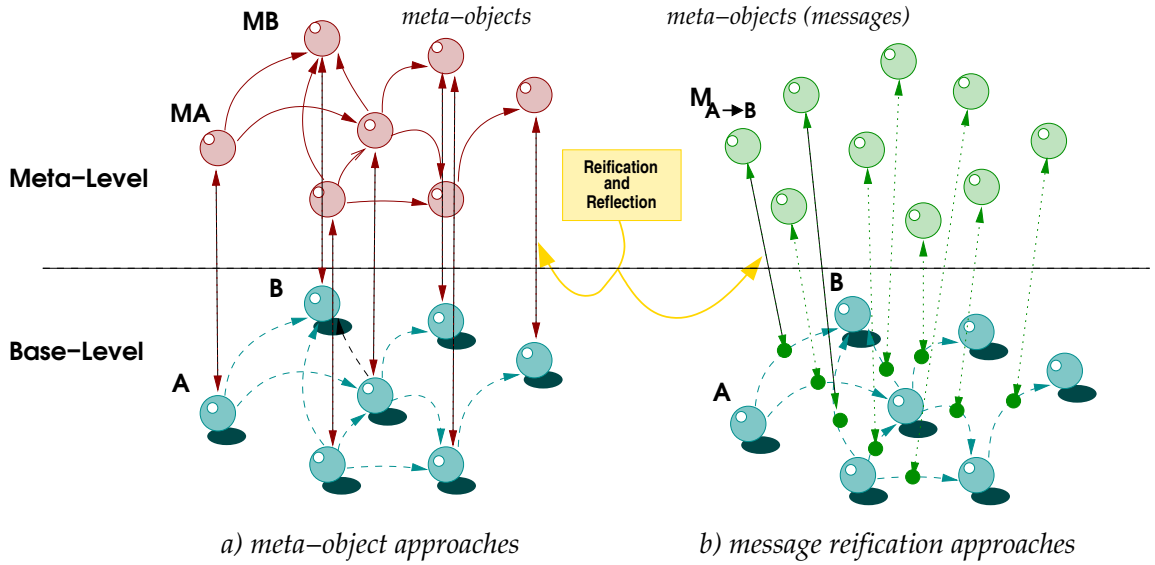

Figure 1. Base- and Meta-Level Communication Graphs. In Fig. a) the communication graph of the base-level is mimicked in the meta-level, whereas in Fig. b) it is reified in the meta-level.

problematic. In such situations, an algorithm usually intended to be a sequential and coherent whole must be divided among several distributed entities. Since no individual entity knows the entire algorithm, the programmer must write additional code to synchronize the distributed objects and to keep updated and consistent the shared data. The meta-programmer[2] must face similar problems when the application domain of his meta-program is the communication among base-entities. Hence, he must write some additional code to keep each reified aspect of the communication in touch with components reifying other communication aspects. Such additional code increases the complexity of the meta-level program and also increases the chances of coding errors. Moreover, scattering the algorithm among several objects contrasts with the object-oriented philosophy, which states that data and the algorithms managing them should be encapsulated into the same entity. Since objects do not have a global view of the data they are managing, we can state that most current reflective middleware platforms *lack a global view*. The global view relevance has been explored by Holland in [24]. He has verified how the intrinsic global view property provided by the *contracts* [23] helps in building reusable components. Therefore, in the current reflective middleware the potential for object reuse is compromised[3]. Moreover, the implementation of many non-functional communication-oriented properties may benefit by achieving the global view property. For example, each filter-driven communication (e.g., encrypted, compressed, and so on) is implemented in a straightforward manner without duplicating the filter and all the related data. Analogously, the realization of nonfunctional features such as load balancing and reliability, that are based on data owned by the sender or the receiver (e.g., their status or load, service availability and so on), would not need to retrieve such data because they are already encapsulated in the meta-entity enriching the corresponding communication.

Customization of communications. Current reflective middleware platforms only support changes to the mechanisms responsible for message transfer, neglecting the separate management of individual messages. To implement different meta-behaviors, (that is, the behavior of the meta-objects) for a single message or for a group of messages, the meta-programmer must write a separate meta-program for each type of incoming message and then '*switch*' based on the message type. Moreover, adapting meta-object to deal with new message patterns requires manually including code for the related handlers. Unfortunately, this increases the complexity and size of the meta-program to the detriment of its readability and maintainability.

2.3 Communications as Application Domain for Reflection.

In summary, the global view requirement is typically not achieved and customizing each single communication is hard to achieve. Fundamentally, this is due to the features of the reflective models embedded in the current reflective middleware. Most reflective models are object-based or limited to consider the single exchanged message instead of the whole communication, that is none of these reflective approaches considers communications as its application domain. In the rest of this section we examine these approaches explaining why they are not suitable for performing meta-computations on communication.

Object-based reification. In these models (basically, derived from the *meta-object model* [32]), every object (called *referent*) is associated with a meta-object which traps messages

[2] By the term meta-programmer we mean the person who designs and programs the program at the meta-level (*meta-program*).

[3] According to [6], a component (in our case the realization of a (non)functional feature) must be self contained to be reusable. That is, the services the component provides, are mostly realized without interacting with the other components, i.e., the component is loosely coupled with the other components.

sent to its referent and implements the behavior of the invocation. Such models thus focus their efforts on managing objects rather than interactions. They have been designed for handling different requirements (i.e., reflecting on various base-level objects), hence they do not address communication-oriented nonfunctional requirements. In particular, by adopting an object-based model to reflect on distributed communications, the meta-programmer often has to duplicate the base-level *communication graph* explicitly at the meta-level (see Fig. 1.a). Implementing new features for base-level communications requires that exchanged messages be trapped at the source, sent to the meta-level, managed and then dispatched to the meta-entity at the destination. In this manner, the meta-level is mimicking the corresponding base-level communication. The problem with this approach is that the meta-programmer has to introduce synchronization and communication code in the meta-program, as well as in the base-level program, increasing its complexity. Therefore, object-based approaches to reflection-on-communication simply move the problem, which reflection tries to remedy, of intertwined non-functional/functional code [25] from the base- to the meta-level. Simulating a base-level communication at the meta-level, as advocated in [35], allows performing meta-computations related to *either* sending *or* receiving actions, *but* not related to the whole communication or involving information owned either by the sender or by the receiver without continuously interacting with them. In addition, object-based reification approaches directly inherit the problem of lack of a global view from the object-oriented methodology [18], which encapsulates computation orthogonally to communication.

Message reification. Problems related to reflect on communications have already been stressed in the literature [19, 33]. Most researchers agree that to overcome these problems one must adopt a reflective model specially designed for dealing with communications. Ferber [19] proposed the *message reification approach*. This approach consists of reifying each exchanged message in a meta-entity, as shown in Fig. 1.b. Messages are reified in the meta-level as long as they exist in the base-level. Meta-entities reifying messages can easily extend the semantics of the communication mechanism, but they do not know senders and receivers of the reified communication therefore they can not manipulate the communication aspects related to senders and receivers. Moreover, due to the *ephemeral* nature of the message, the meta-program can carry out meta-computations involving previously computed information only if this information is stored between two reifications. We term the just described phenomenon as *lack of information continuity*. Profiling and load balancing are examples of nonfunctional requirements that are difficult to implement without information continuity.

Although the message reification model is more suitable for enriching the semantics of communication than object-based approaches, we still think that the message reification approach is not well-suited for distributed computing. The message reification model not only lacks information continuity, but also does not permit interacting with the object involved in the communication, and is limited to point-to-point communications. Thus, the realization of a reflective middleware that completely supports communications as application domain for reflection needs further examination.

3 Communication Channels as First-Class Citizens

To address the problems described in section 2.2 we have to design from scratch a reflective approach that reifies and reflects on communications directly. This means designing a reflective model whose domain is not a base-object (as in the meta-object model) or a message (as in the message reification model), but rather an entire communication among base-objects. That is, we need to encapsulate message exchanges (not only messages) into a single logical meta-object instead of scattering the relevant information related to it among several meta-objects. Hence, communication channels must be treated as *first-class citizens*.

To fulfill this purpose, we have designed a communication-oriented model of reflection which we call the *multi-channel reification model*. This approach is inspired by the message reification model [19], which is message-oriented but not communication-oriented. We have extended that approach by introducing the concept of a *communication channel*. As we will show, this extension provides a way to structure middleware with the global view property.

3.1 Multi-Channel Reification Model.

The *multi-channel reification model* considers a synchronous method invocation as a message sent through a logical channel established among an object requiring a service (in the following termed as *sender*), and a set of objects providing such a service (in the following termed as *receivers*)[4]. The model then supports the reification of such logical channels into logical objects called *multi-channels*[5]. In this way, the abstract concept of a communication channel is embodied as an object; i.e., the communication channel becomes a first-class citizen. Each multi-channel monitors the exchange of messages and potentially enriches the underlying communication semantics with new features. Each method call is trapped by the multi-channel when performed, then it is (potentially) imbued with new semantics, and finally it is delivered to the designated destinations. Multi-channels can be viewed as interfaces established between objects requiring services and objects providing such services with the aim of enriching these services with new communication-related features.

The Kind Concept. The purpose of a multi-channel is to enhance the behavior of its associated communication channel with a single new feature. Each such feature is termed a *kind*; examples are verbosity, check-pointing, authentication, load balancing, and so on. Each message sent from a sender to a given group of receivers is different and could be characterized by different requirements. The kind concept allows the

[4] We treat senders and receivers as separate entities, but this choice should not be considered mandatory or restrictive — senders and receivers can coincide without problems. In this case, the multi-channel will look like a meta-object able to reflect both on outgoing, and on ingoing messages.

[5] Our meta-objects are called *multi-channels* after the fact that, in our view, multi-cast is the most general communication model and all communication can be modeled on top of it by using our notion of channel.

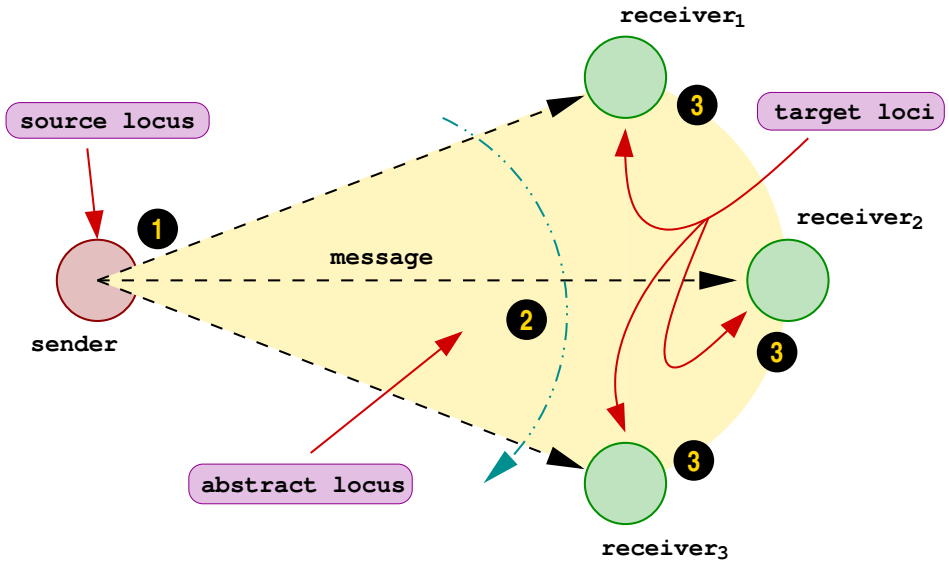

Figure 2. Objects and loci involved in a multi-point communication.

programmer to differentiate the management mechanism associated with each exchanged message. Several multi-channels can be established among many senders and the same group of receivers. Each of these multi-channels implements a different behavior (behavior identified by its kind) and monitoring a different set of exchanged messages. For example, we could consider a critical system that we want to enhance so that it checkpoints exchanged messages. In this case, it is convenient to store information only about methods whose execution causes state changes, i.e., to partition messages into two sets: non-modifying and modifying messages. Hence, we establish two kinds of multi-channels between the same group of objects: one managing non-modifying messages and the other managing the modifying messages with enhanced semantics defining the expected checkpointing mechanism.

How Multi-Channels Are Looked-up. Having introduced the notion of kind, we are in a position to define a multi-channel as follows:

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

That is, each multi-channel is characterized by its kind, and the set of referents playing the role of receivers. This characterization of the multi-channel is sufficient to allow the runtime system to determine the multi-channel to which each message is to be passed. Note that because of the intrinsic dynamicity of the communication semantics, the sender of the message is not relevant to the characterization of a multi-channel. Different senders can hook themselves to the channel at different times and for different requests.

Reification. Each reification is related to a communication channel established among the interacting objects, and takes place when a communication channel is used for the first time, i.e., when a sender first sends a message to a particular group of receivers. The new multi-channel is compliant with the characteristics of the communication it embodies, i.e., its kind satisfies the requested behavior and its referents are the receivers of the message, and so on. Each message requiring an already used channel is managed by the multi-channel which embodies such a 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 to compute the expected result. In summary, when an object requests a service to another object the following steps take place: the exchanged message is transparently trapped and sent to the multi-channel; the multi-channel reifies the communication channel that must be used; the multi-channel delivers the message, in compliance with its behavior, to the designated receivers, then the multi-channel returns the result. The efficiency of the context switch between the base- and the meta-level is addressed in section 7.3

The Loci of Meta-Computations. As depicted in Fig. 2, a communication has a wide *area of influence* (the gray cone in the picture) and involves several aspects and components of the system. By *area of influence*, we mean both performed actions, (e.g., data marshaling and unmarshaling, message delivery, and so on), and participating entities (sender and receivers). To determine where and when meta-computations on communications might be performed, we have to analyze and partition the area of influence in accordance with where and when actions necessary to carry out the communication are performed. Each such partition, which we name a *communication locus*, is an abstraction of a part of the logical path that a message traverses to get to the designated receivers. In our taxonomy, an area of influence has three kinds of communication loci (numbers refer to the situation depicted in Fig. 2):

❶ the sender and the beginning of the communication, named the *source locus*,
❷ the dispatching and delivering of the message to the designated receivers, named the *abstract locus*, and

Kinds	::=	`kinds` KindList†
KindList	::=	`kind` <*kind-name*>‡ `with` ReceiverNameList `to` MethodNameList KindList |
		`kind` <*kind-name*> `with` ReceiverNameList `to` MethodNameList
ReceiverNameList	::=	<*receiver-name*> , ReceiverNameList | <*receiver-name*>
MethodNameList	::=	<*method-name*> , MethodNameList | <*method-name*>

† Terminals are written using a non-proportional font, whereas non-terminals are written using a proportional font.
‡ Identifiers in brackets represent strings whose meaning is clearly defined by their name.

Grammar 1: kinds and kind.

❸ the receivers and the execution of the just-delivered message, named the *target loci*.

These loci represent the reification of the whole communication and of the entities (hidden or not) that implement a distributed communication. Each multi-channel represents these loci. It moves the communication mechanism into the meta-level under the control of the meta-program and frees the base-level from the responsibility of managing the communication. For example, messages, which are marshaled by the sender, are moved into the source locus where the meta-program can piggyback them with extra data. To allow meta-programs to customize the semantics of each communication, the multi-channel performs a specific meta-computation for each locus extending their standard behavior. Meta-computations performed on source and target loci are always carried out at the site of the senders and receivers, respectively. The taxonomy we have adopted derives from requirements of performance, reliability and availability. Locating part of the meta-program directly at the source and target loci is convenient for reducing inter-process communications, while working on the abstract locus allows meta-computations to be decentralized and the reliability and the availability of the provided services to be improved at the meta-level. For example, the abstract locus can be used to perform checkpointing, rollbacking, filtering, and so on, that is, to perform actions related to neither the sender's nor the receiver's site.

4 Programming with Channels

We have extended the Java RMI framework to a system called mChaRM, an acronym for multi-Channel Reification Model, supporting our model. This section describes the set of extensions to Java to support our model and a simple API for meta-programming.

4.1 Base-Level Language Extensions.

For the run-time system to select the right multi-channel to use, each request for remote service must provide information about both the communication channel (the receivers of the message) and the required behavior (the kind of the communication channel they intend to use). To provide this information we have introduced in the base-level language a construct for specifying a link between each method-call, the receivers of such calls and the behavior that will be used for carrying out such calls.

4.1.1 **Kinds**.

The **kinds** statement binds communication channels to multi-channels. Each binding expresses which multi-channel is implicitly used to deal with a message sent through a specific communication channel, in agreement with the characterization of multi-channels given in section 3.1. Each binding is introduced by the **kind** keyword and expressed through its components: a set of receivers, the kind and a list of messages that the related multi-channel must trap and send to the meta-level. The syntax of the **kinds** statement is described by Grammar 1.

Kind-statement bindings are managed by the compiler/interpreter and translated into statements that the run-time system uses to reify the related communication channel, to decouple the base-entities from the communication loci, and to route the method calls toward the correct multi-channel. Of course, this translation is tied to the adopted multi-channel architecture. More about this is explained in section 5.

The **kinds** statement has to be inserted in the class definition of each potentially reflective sender. It contains meta-information that describes the interface of each instance towards the meta-level. A reflective sender, written in mChaRM, looks like:

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

Note that, to improve transparency and to remove meta-information from the base-level code, data related to the communication channel can also be passed to the system at compile-time through a directive to the compiler.

In the rest of the paper, we denote the kind of a multi-channel with symbols «». For example, in the above snippet of code, we inform the run-time system that multi-channels of kind «`verbose`» deliver $\texttt{dummy}_1$ messages to the given receivers `A`, `B` and `Z` sent from instances of the class `dummy`.

ClassBody♭	::=	KindDefinition SourceLocusSection AbstractLocusSection TargetLocusSection♯
KindDefinition	::=	`kind:` <*label*>†
SourceLocusSection	::=	`source-locus:` NormalBody‡ | ε
AbstractLocusSection	::=	`abstract-locus:` NormalBody | ε
TargetLocusSection	::=	`target-locus:` NormalBody | ε
Expr♭	::=	MethodCall° | FieldAccess° | LocusExpr
LocusExpr	::=	LocusIdentifier.MethodCall | LocusIdentifier.FieldName
LocusIdentifier	::=	`source-locus` | `abstract-locus` | `target-locus(`<*ReceiverName*>`)`

♯ To simplify the exposition we have set an order among locus sections.
† <*label*> represents a string used to identify the kind of meta-behavior.
‡ NormalBody is not further expanded in this paper and represents the body of a Java class.
♭ ClassBody and Expr are nonterminals of the Java grammar and need not be further expanded.
° MethodCall, and FieldAccess are nonterminals of the Java grammar, which describe method-calls and access to object attributes, respectively.

Grammar 2: Multi-channels.

4.2 Meta-Level Programming Language.

We have developed a programming environment that allows programmers to write classes describing multi-channels benefitting from the global view property. The resulting programming language is derived from Java and permits multi-channel description through its loci. The extensions to Java are described by Grammar 2.

4.2.1 Kind Definition.

Each class of multi-channels describes the nonfunctional behavior, i.e., the kind, applied to the reified communication channel. Programmers can specify such a kind through the keyword **kind:**.

4.2.2 Locus Sections.

To get the global view property, communication loci have to transparently interact, allowing the meta-programmer to manage the trapped message without worrying about passing it from one locus to another, and without explicitly coordinating loci work. Classes describing multi-channels are divided into three sections, beginning with one of these new qualifiers:

- **source-locus:**
- **abstract-locus:**
- **target-locus:**

Each section represents the locus of the communication channel that it reifies. Each section contains all methods and fields related to such a locus. These methods are written in standard Java and they use the API described in section 4.3. It is not mandatory to write all sections. If a section is omitted, the corresponding locus either does not perform actions or performs inherited actions when messages pass through it.

4.2.3 Locus Representatives.

Distributed nonfunctional features are often based on data owned by both the sender and the receivers involved in the communication. For example, to distribute service requests equally among two or more servers, the client needs information about the load of each server. Hence, loci should cooperate with each other to carry out some services. Of course, meta-programmers should manually coordinate the access to such services and data. To achieve the global view property, we have to carry out a high-level and transparent loci coordination. Each locus has a representative that masks and takes care of intra-object communications, i.e., it delivers each request to the corresponding component at the proper moment instead of having to be specified by the programmer. These representatives have the same name as the locus they represent, e.g., **abstract-locus** for the abstract locus. We use the dot notation to access their services and data, for example, **target-locus**(i).retrieveField(field)[6].

4.2.4 Multi-Channel Interactions and Relations.

In spite of our extensions to Java, all the advantages of the object-oriented paradigm are retained. Multi-channels can also extend existing multi-channels instead of having to be programmed from scratch. Obviously, the inheritance relation which binds two multi-channel classes is propagated to their components. Hence, each locus inherits methods and fields defined in the corresponding locus section from the parent multi-channel class.

Multi-channels can interact (through remote method invocations) with other multi-channels (see [3]). Of course these communications as every other communication can be reified by multi-channels, thereby providing support for a reflective tower.

[6] Please note that the abstract locus knows all its target loci, and a target locus can easily refer to another locus by taking from the abstract locus representative the needed information.

Generic Multi-Channel Introspection
`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.*
Senders and Receivers Side Introspection and Intercession
`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(String methodName, Object[] args)` *invokes a specified method of the referent with the specified arguments.*

Table 1. A portion of the API provided by mChaRM for carrying out introspection and intercession on senders and receivers.

Message Introspection and Intercession
`String getMethodName()` *retrieves the name of the method called.*
`String setMethodName(String methodName)` *through the name changes the method which will be really activated.*
`Object inspectArgument(int position)` *retrieves the value of a specified actual argument.*
`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.*
`Object getReturnValue()` *retrieves the return value, can be used only after the return value has been calculated.*
`Object setReturnValue(Object newValue)` *replaces the return value with a new one, returns the old value, can be used only after the return value has been calculated.*

Table 2. A portion of the API provided by mChaRM for carrying out introspection and intercession on messages.

Meta-Behavior API
Abstract locus
`Object coreMetaBehavior(mChaRMMethodCall msg)` *this method embodies the reflective behavior realized by the multi-channel.*
Source locus
`void beforeSenderSideMetaBehavior(mChaRMMethodCall msg)` *elaborates the just trapped message on the sender site.*
`void afterSenderSideMetaBehavior(mChaRMMethodCall msg)` *performs sender side meta-computations on the message, immediately before forwarding it to the abstract locus.*
Target loci
`void beforeReceiverSideMetaBehavior(mChaRMMethodCall msg)` *elaborates the just received message on the receiver site.*
`void afterReceiverSideMetaBehavior(mChaRMMethodCall msg)` *performs receiver side meta-computations on the message, immediately before giving it back to the abstract locus.*

Table 3. APIs provided by mChaRM for carrying out meta-computation.

4.3 APIs used by Multi-Channels.

We propose a simple API that aims to be sufficient to support a wide range of multi-channel uses. Our API does not claim to be exhaustive, we have kept it simple for presentation purposes. In spite of its simplicity our API is powerful enough to support sophisticated communication mechanisms and to show the capabilities of the multi-channel approach. The prototype implements a more extended API that will be widened in the near future. Methods are classifiable, according to their purpose, into three categories: *introspection*, *intercession*, and *meta-behavior*.

We postpone the description of the classes containing the methods of the API, which depend on the particular multi-channel architecture, to section 5 where the multi-channel structure and its implementation are described.

4.3.1 Introspection and Intercession.

The proposed model is designed for supporting the enhancement of communication semantics, and not for managing the base-object structure or semantics. For this reason the part of the API devoted to intercession on multi-channel referents

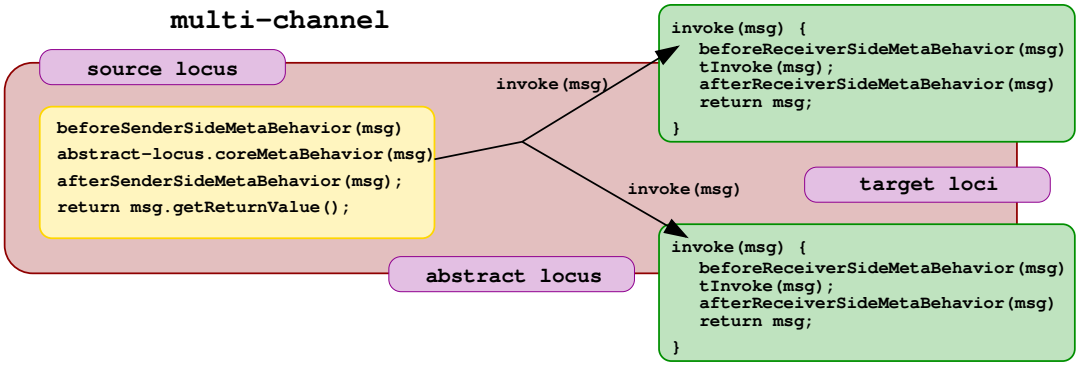

Figure 3. Following a method call through the meta-level.

is kept simple and consists of the few methods shown in Table 1. Since haphazard intercession might lead to an inconsistent state of involved referents, we constrain the current API as follows: a multi-channel can only look into the state of its referents and invoke a method of one of its referents playing the role of receiver.

The part of the API devoted to carrying out intercession and introspection on messages dispatched through a multi-channel is more complete. This part of the API is composed of the methods in Table 2 and Table 3. These methods represent the core of the whole mechanism allowing multi-channels to alter messages that pass through them. The API provides the meta-programmer methods for looking into the contents of the actual parameters, for modifying their contents, for piggybacking extra arguments to the method call we are handling, and so on. Section 6 shows how to use these methods to implement communication protocols.

4.3.2 Meta-Behavior.

Methods belonging to this category implement the multi-channel kind and so define how multi-channels must behave. The meta-programmer must override these methods to build new kinds of multi-channels.

Methods shown in Table 3 define the meta-computation that a multi-channel carries out on messages when they transit through the three loci (see section 3.1). Their arguments (an instance of class `mChaRMMethodCall`) represent the exchanged message and its components, i.e., the name of the activated method, the value of the actual arguments of the method, and its designated receivers.

Methods `beforeSenderSideMetaBehavior`, `beforeReceiverSideMetaBehavior`, `afterReceiverSideMetaBehavior` and `afterSenderSideMetaBehavior` perform meta-computations on messages when they are passing through the source (first and fourth methods) and the target loci (second and third methods). They do not return values. Nevertheless, they allow the multi-channel to modify the exchanged message via side-effects performed on their actual arguments. Their default behavior is to do nothing.

The method `coreMetaBehavior` coordinates the result of the multi-channel computations performed on the target loci and decides how to handle the trapped messages and where to demand their computation. Its default behavior forwards the messages to the given receivers and returns the last received value to the caller.

The meta-program is basically defined by the execution of these five methods. These methods are implicitly activated by the meta-computation when messages flow through the meta-level. As depicted in Fig. 3, `beforeSenderSideMetaBehavior` is called when the message is intercepted and handled at the meta-level in the source locus. The message is passed to the abstract locus and managed by the `coreMetaBehavior` routine[7]. When the message gets to a target locus, it is managed by the `beforeReceiverSideMetaBehavior` (it is activated each time the meta-program calls the method `invoke`), then it is delegated to the base-level and the return value is managed by the `afterReceiverSideMetaBehavior` just before being returned to the abstract locus. Finally, the return value is again managed by the `afterSenderSideMetaBehavior` when the abstract locus returns it to the source locus.

The following example shows how the behavior of a multi-channel is described: the snippet of code below implements a multi-channel of kind «`verbose`». This kind of multi-channel is designed for tracing a message while it is passing through the meta-level loci, i.e., it traces when a message leaves the sender, when it goes through the multi-channel computation, and when it gets to the designated receivers.

```
class verboseChannel extends multiChannel {
  kind: verbose;
  abstract-locus:
    public Object coreMetaBehavior(mChaRMMethodCall m) {
      System.out.print("abstract locus: ");
      m.printmChaRMMethodCall();
      return super.coreMetaBehavior(m);
    }

  source-locus:
    public
     void beforeSenderSideMetaBehavior(mChaRMMethodCall m) {
      System.out.print(
         "source locus["+retrieveField("whoAmI")+ \
```

[7] No assumptions are made about the location where the `coreMetaBehavior` is performed. Thus, it has to contain only statements whose execution is location-independent. For location-dependent statements the meta-programmer must override `beforeSenderSideMetaBehavior`, `afterSenderSideMetaBehavior`, `beforeReceiverSideMetaBehavior` or `afterReceiverSideMetaBehavior` routines.

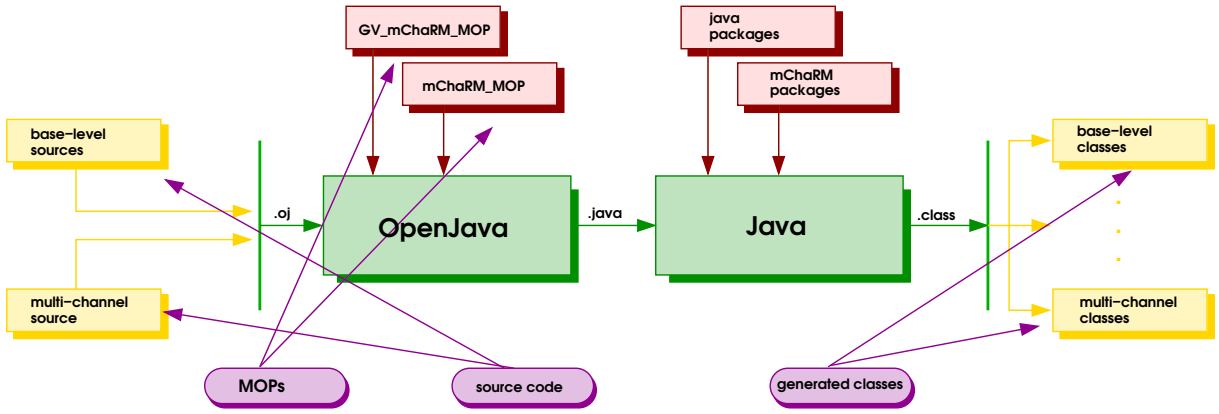

Figure 4. Compiling process.

```
        "]: --> "+m.getMethodName()+"( ");
      if (m.hasArgs)
        for(int i=0;i<m.actualArguments().length;i++)
          System.out.print((m.actualArguments())[i]+" ");
      System.out.println(")");
    }
    public
     void afterSenderSideMetaBehavior(mChaRMMethodCall m) {
      System.out.println(
         "source locus["+retrieveField("whoAmI")+ \
         "]: <-- "+m.getReturnValue());
    }

  target-locus:
    public
     void beforeReceiveSideMetaBehavior(mChaRMMethodCall m) {
      System.out.print(
         "target locus["+retrieveField("whoAmI")+ \
         "]: --> "+m.getMethodName()+"( ");
      if (m.hasArgs())
        for(int i=0;i<(m.actualArguments()).length;i++)
          System.out.print((m.actualArguments())[i]+" ");
      System.out.println(").");
    }
    public
     void afterReceiverSideMetaBehavior(mChaRMMethodCall m) {
      System.out.println(
         "target locus["+retrieveField("whoAmI")+ \
         "]: <-- "+m.getReturnValue());
    }
}
```

5 mChaRM: Architecture

The multi-channel reification model and the described language extensions have been realized by the mChaRM framework, which is developed in Java. mChaRM consists of three components:

❶ a preprocessor dealing with the language extensions,
❷ a Java package (`mChaRM.multichannel`) containing the skeleton classes needed to develop new multi-channels, and
❸ a Java package (`mChaRM.mChaRMCollection`) collecting multi-channels implementing some sample kinds, e.g., «`verbose`», «`validation`», «`multi-trig`», and so on.

The preprocessor has been realized using OpenJava [13]. OpenJava has a compile-time MOP which helps in prototyping new programming languages. We have written two meta-objects (`mChaRM_MOP` and `GV_mChaRM_MOP`) which drive the OpenJava compiler (`ojc`) during the translation of mChaRM code into pure Java code. The former meta-object expands the **kinds** section adapting the generated code to support the described reflective approach, whereas the latter meta-object translates multi-channel classes to fit the chosen architecture requirements. Figure 4 shows how mChaRM source code is compiled into Java bytecode.

Our framework allows programmers to enhance multi-point communications with new features. Java does not support multi-point communications, such as broadcast, multicast, or multi-point RMI. Hence, mChaRM adopts its own realization of multi-point RMI based on KaRMI [38]. A multi-point communication can be explicitly started by invoking the method `multiRMI`:

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

The semantics of `multiRMI` is very simple. It multicasts a specified method call to a given set of servers and then gathers their results and returns the first received value to the caller.

At the moment, `mChaRM_MOP` adds `multiRMI` to the base-level code and also hides the reification/reflection mechanism. We are already working on decoupling `multiRMI` from the mChaRM framework to improve the transparency of the reflective mechanism (see [12]). We have also kept its implementation simple because the main purpose of mChaRM is to render the multi-communication protocol customizable by the (meta-)programmer, and not to realize an efficient or complete multi-point remote method invocation mechanism for Java. For simplicity we also use the `multiRMI` for opening up the point-to-point RMI. Of course, communications that do not need to be extended with extra functionality can be carried out with the traditional communication mechanisms provided by Java.

The rest of this section describes the architecture chosen for the multi-channels and how it has been realized. We also briefly describe OpenJava, the developed meta-objects, and how programs are translated into pure Java.

5.1 Multi-Channel Structure.

To achieve both a good balance among performance, availability, reliability, and transparency and to comply with the

model requirements, we have chosen to design and implement the multi-channel as a distributed entity, composed of a central kernel (termed *core*), if necessary replicated, and as many stubs as its referents. Each stub is an object associated with a base-level object running on the same site and in the same addressing space. Each stub is designed for interfacing its referent to the multi-channel core, and vice versa. Sender stubs embody source loci, receiver stubs embody target loci, and the core encapsulates the functionality of the abstract locus. A sender stub transparently traps each message directed to the multi-channel of which it is a part. On the other hand, receiver stubs actually invoke the method that the multi-channel is managing. Many stubs may be attached to each base-level object, one stub for each multi-channel connected to that object. Stubs also deal with source and target locus meta-computations and implement the related API methods (see Table 3).

Both sender and receiver stubs are dynamically attached to the multi-channel core. Following its characterization (see section 3.1), the multi-channel knows which of its referents is playing the role of receiver. During its initialization, the multi-channel orders its receivers to create a stub that will be part of the multi-channel. By parsing the **kinds** section in the sender, the preprocessor knows which multi-channels the sender could use during its lifecycle. Thus, clients, once activated, ask the multi-channel core for a stub to use for hooking to the multi-channel. To minimize the amount of exchanged data the core only supplies the name of stub classes to them. Then senders use such an information and the methods provided by the Java core reflection library [42] to really create a stub.

Each stub can be viewed as an extension of its own referent. It communicates with its referent via a local method call thereby avoiding problems of communication reliability, and network partitioning. Sending and receiving actions appear as if they are moved to the meta-level where they are managed by using the multi-channel semantics. In this way, the system's functional behavior may be held constant while varying the underlying communication algorithms.

The multi-channel core does not reside in a specific location with respect to its referents. Its computation is independent of its location and usually depends only on the implemented behavior (for example, if it implements a fault tolerant behavior then it will either be located on a failure-resistant machine or replicated on several machines). The core communicates with stubs via remote method calls. Communications among multi-channel components are considered as intra-object communications and are performed by the framework transparently to the 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. Message communications are managed as follows[8]:

❶ the sender stub of the multi-channel, whose kind matches the one specified in the call, traps the message;
❷ the stub performs the `beforeSenderSideMetaBehavior`;
❸ it calls the `coreMetaBehavior` into the abstract locus (this call is a remote call);
❹ the core performs its computation;
  ⓐ the core calls the `invoke` into the target loci (optional computation);
  ⓑ the corresponding receiver stubs drive the execution of the message and return the result to the core;
❺ the core performs some computations on the result and returns it to the sender stub;
❻ the stub performs the `afterSenderSideMetaBehavior` on the return message;
❼ it returns the result to its referent, as the required call should have done.

Note that the optional computations in the target loci (subpoints ⓐ and ⓑ of point ❹ in the algorithm above) involve both the `beforeReceiverSideMetaBehavior`, `afterReceiverSideMetaBehavior`, and the execution of the trapped message by the referents of the target loci.

We mitigate the overhead due to the extra remote method invocation between the source and abstract loci (point ❸ in the algorithm above), without sacrificing the flexibility of the approach, by entrusting all the remote communications performed by the multi-channel to the efficient RMI mechanism provided by KaRMI [38]. Moreover, for the sake of efficiency we have also provided a special kind of multi-channels, called «`compact`», which merges the source and abstract locus in a single component, lying in the same addressing space of the sender, without compromising the functionality of the source and abstract loci. Such an architecture slightly contrasts with the multi-channel characterization described in section 3.1 fixing the sender for each multi-channel, but it provides a very efficient multi-channel (see section 7.3) that can be used when the multi-channel will always be used by the same sender.

5.2 Supporting Framework.

Our framework provides the package `mChaRM.multichannel` containing the classes: `senderStub`, `receiverStub` and `channelCore`. These classes represent the skeleton of a multi-channel and must be used by the multi-programmer to derive new stubs and core classes and therefore new kinds of multi-channels. These classes provide the basic services common to every locus. In the rest of this subsection we examine their implementation.

5.2.1 Stubs.

Stubs represent the basic components of a multi-channel. Both sender and receiver stubs inherit from a common class, called `stub`, which keeps data about the multi-channel identity and methods for handling these data. The abstract locus, i.e., the multi-channel core, can perform both introspection and intercession on multi-channel referents by delegating introspection and intercession to the stubs implementing source and target loci. Hence, each kind of stub must play the server role. Sender stubs' main behavior consists of passing to the multi-channel core the modified message.

[8] This algorithm fills the gaps left in Fig. 3 and in the description given in the previous section explaining how the multi-channel hooks to the base-level.

Structure and methods, defined for each kind of stub, change according to their desired functionality. The main job performed by a sender stub consists of forwarding the (modified) call to the multi-channel core. The caller implicitly (locally) invokes the `stubBehavior` method of the multi-channel it would like to use. `stubBehavior` performs a call to `beforeSenderSideMetaBehavior` and then it passes the modified information about the original call to its multi-channel core. The method `afterSenderSideMetaBehavior` is activated on the (modified) message, when the core returns the control to the source locus.

```
  final public Object stubBehavior(mChaRMMethodCall msg) {
    beforeSenderSideMetaBehavior(msg); // source locus
meta-computation
      // dispatching the message to the abstract locus
    Object res = WhoIsMyCore().coreMetaBehavior(msg);
    msg.setReturnValue(res);
    afterSenderSideMetaBehavior(msg); // source locus
meta-computation
    return msg.getReturnValue();
  }
```

`stubBehavior`, `beforeSenderSideMetaBehavior` and `afterSenderSideMetaBehavior` can not be directly invoked by the meta-program. Both `SenderSideMetaBehaviors` can be overridden in classes derived from `senderStub` to build new kinds of multi-channels.

The main service provided by each receiver stub is the method `invoke`. Its behavior consists of delegating the message filtered by the multi-channel to the base-level. The `invoke`, before delegating the message for execution, invokes the `beforeReceiverSideMetaBehavior`, and immediately after it filters the message (containing also its result) invoking the `afterReceiverSideMetaBehavior`. The modified message is really delegated to the base-level using the `tInvoke` method.

```
  final public Object invoke(mChaRMMethodCall msg) {
    beforeReceiverSideMetaBehavior(msg); // target locus
meta-computation
    Object res = tInvoke(msg);
    msg.setReturnValue(res);
    afterReceiverSideMetaBehavior(msg); // target locus
meta-computation
    return msg.getReturnValue();
  }
```

The private method `tInvoke` applies the message using Java core reflection library [42] features.

5.2.2 Core.

The multi-channel core (described by the class `channelCore`) must be bound to the stubs of its referents when it begins its execution. To do that, its constructor requires all related data, i.e., its kind, the name of its known referents, and the class name of its stubs. The constructor binds the just created instance with the stubs of the given referents, and initializes the multi-channel structures.

New kinds of multi-channels are developed by extending the class `channelCore`. As explained, the behavior of the multi-channel is determined by the method `coreMetaBehavior`. Every new kind of multi-channel will override this method. The default behavior of `coreMetaBehavior` consists of forwarding the trapped message to the specified receiver stubs for its execution, and of dispatching the result of the computation of the first receiver back to the callers.

```
public Object coreMetaBehavior(mChaRMMethodCall msg) {
  Object[] result = new Object[(msg.receivers()).length];
    // dispatching the message to the target loci
  for(int i=0; i<(msg.receivers()).length; i++)
    result[i] = (receiverStub((msg.receivers())[i])).invoke(msg);
  return result[0];
}
```

«`normal`» is the default kind provided by our framework. The behavior of a multi-channel of this kind simply consists of realizing a context switch between base- and meta-level and vice versa, and of delivering the trapped message, without alterations, to the given receivers. It represents a good starting point for deriving new communication behaviors.

5.3 mChaRM Preprocessor.

Translation of mChaRM code to pure Java code is carried out through compile-time reflection. This is implemented by two meta-objects: `mChaRM_MOP`, and `GV_mChaRM_MOP`, which drive the OpenJava compiler (`ojc`) during the translation of mChaRM code into pure Java code.

5.3.1 OpenJava.

OpenJava [13] is a compile-time MOP for Java. It can be seen as an *advanced macro processor* that performs a source-to-source translation of a set of classes written in an enriched version of Java into a set of classes written in standard Java.

Translations to be applied to a base class are described in a meta-class associated, via the **instantiates** clause with the base class. The meta-class is written in standard Java by using a class library that extends the Java reflection API [42] with classes that reify language constructs.

Macro expansion 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 driven by the `translateDefinition` method of the associated meta-objects. As a result, writing a translation is straightforward because of the object-oriented design of the library.

5.3.2 Meta-Object to Manage the Base-Level.

OpenJava, through the meta-object `mChaRM_MOP`, manages the extensions to the base-level (see section 4.1). This meta-object takes care of expanding the kinds clause found in senders and of adding into both senders and receivers all the necessary code to support the approach (`multiRMI`, binds to the stubs, and so on).

mChaRM_MOP knows that it is parsing a sender class when it detects the keyword **kinds**. In that case, mChaRM_MOP renders available to the instances of such a class the mapping expressed by the **kinds** clause. This is done, at senders' creation, by filling a hashtable, indexed on messages and their potential receivers, with the kind of the multi-channels that have to be used for managing such messages. Data stored into such a hashtable are used by the multiRMI method for intercepting and passing the message toward the intended multi-channel.

mChaRM_MOP also adds the multiRMI method to each class it detects to be a sender class. The method multiRMI delivers a specified message to the designated receivers. It verifies in the hashtable if a multi-channel has been associated with such a message. If so, it asks such a multi-channel to deliver the message by using the related stub. Otherwise, it directly delivers the message to the given receivers.

Both to sender and receiver classes, the mChaRM_MOP adds the method attachingStub. This method binds a receiver stub to the current object during the multi-channel bootstrap.

5.3.3 Meta-Object to Manage the Meta-Level.

As explained in section 4.2, each multi-channel is represented by a single class, rather than multiple classes. However, a multi-channel is effectively composed of several objects (see section 5.1). The meta-object GV_mChaRM_MOP drives decomposition of a multi-channel class into the classes describing its components. Note that this is transparent to the meta-programmer.

The meta-object GV_mChaRM_MOP creates three classes and fills each of them with attributes and methods related to the corresponding locus being described. This job is simplified by the presence of the qualifiers **abstract-locus:**, **source-locus:**, and **target-locus:**. They introduce all data (i.e., methods, attributes, and so on) related to the multi-channel components. Besides, they uniquely identify the membership of each method and attribute. In fact, each method or attribute under the aegis of one of these qualifiers belongs to the corresponding locus (e.g., methods under the aegis of **target-locus:** belong to the target locus, hence to the class describing receiver stubs).

The meta-object GV_mChaRM_MOP implicitly adds to the newly created classes the code to remotely interconnect the components of the multi-channel. It also deals with locus references. Each access that the meta-programmer considers as a local access to a method or an attribute of another locus is transparently transformed into a remote call by the meta-object. In this way, the global view property is achieved.

A restricted inheritance relation among multi-channels is the only limitation due to the fact that we code multi-channels as a whole and then we split them in separate and distributed components. In fact, we cannot decide from which class a component class has to inherit. All the classes inherit from the same group of classes, i.e., when a multi-channel class inherits from another multi-channel class, the classes representing its components must inherit respectively from the classes of the components of the parent multi-channel. We are studying how to relax this constraint about inheritance among multi-channels.

6 Multi-Channels at Work

Our approach aims to experiment with adding complex communication behaviors transparently to the application. Data compression, efficient data marshaling and unmarshaling, encrypted communications, future-based remote method invocation, message checkpointing, and load balancing, are examples of nonfunctional requirements that can be managed by multi-channels. At the moment, we have realized a few multi-channel kinds: «verbose», «validation», «historical-validation», «RMP». All are distributed with the mChaRM framework.

This section describes two applications of the multi-channel reification approach. Both examples show snippets of code written using the API and the syntax described earlier. The first example shows how to build a multi-channel and how the classes describing multi-channels' referents look like. The second example is a little more complex and in this simplified and paper-tailored form tries to show a wide range of features of mChaRM: the use of locus representatives, message manipulation, locus intercession and so on.

This section does not provide an exhaustive description. It only tries to explain some details of the multi-channel reification approach through simple examples. More detailed and complex examples can be found either in [11] or in the framework distribution.

6.1 Authorization Policies.

A potential communication extension consists of checking if a message can or cannot be delivered to a receiver in agreement with a given authorization policy. Each message can be considered as a service request forwarded from the sender to a given receiver. In this case, a multi-channel established between two objects plays the role of judge, verifying if the sender has the permission for requesting such a service. As stated in [4] associating the validation phase to the communication instead of the receiver hinders malicious attacks. The code reprises the ATM example presented in [4], with customers and ATMs. In this example customers can deposit in and withdraw from ATMs. Basically a multi-channel is established between a customer and an ATM that verifies that only the owner can withdraw from his account. The class ATM simply defines methods withdraw and deposit.

```
public class ATM instantiates mChaRM_MOP {
  public void deposit(String id, String accNum, int sum) {
    // code for depositing the «sum» on the «id»'s account.
  }
  public void withdraw(String id, String accNum, int sum) {
    // code for withdrawing the «sum» from the «id»'s account.
  }
}
```

The class customer is more interesting than the class ATM. It shows how it is possible to use the kind mechanism to monitor only a subset of all the provided services.

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

  public static void main(String[] args) {
    customer c = new client("Walter");
    c.multiRMI("deposit", new String[]{"ATM1"},
          new Object[]{"Walter", "#1237", new Integer(1000)});
    c.multiRMI("withdraw", new String[]{"ATM1"},
          new Object[]{"Walter", "#1237", new Integer(2000)});
  }
}
```

Our example requires that anyone can make a deposit, whereas a withdrawal can be performed only by the owner of the account. To ensure such a behavior, a multi-channel validating only the withdrawal messages is enough. Hence, a multi-channel of kind «validation» is associated with the execution of the method withdraw (see above for the customer definition).

```
class validationChannel extends multiChannel {
  kind: validation;

    public validationChannel(String permissionFileName) {
       // initialize the right table from permissionFileName
    }

  abstract-locus:
    private Hashtable right;
    public Object coreMetaBehavior(mChaRMMethodCall msg) {
      bool permission =
          right.get(key(client, receiversName, methodName));
      if (!permission) throw IllegallRequestForAService;
      else return super.coreMetaBehavior(msg);
    }
}
```

Most of the work is performed in the abstract locus. Validation is an action independent of where it is performed. Thus we have only redefined the coreMetaBehavior to carry out the validation phase. In [3, 4], we present examples of multi-channels dealing with more complex authentication policies than a simple access matrix, e.g., a policy based on the history of communications.

6.2 Reliable Multicast Protocol.

Our second example outlines a multi-channel realizing the *reliable multicast protocol* by Todd Montgomery [36].

Reliable Multicast Protocol (RMP) is a reliable multicast transport protocol offering a variety of services of different quality. Each sender can sequence its packets separately, or RMP can sequence all the packets sent to a group. RMP also offers a feature called total ordering, which means that RMP ensures that every member of the group has received the packets before passing them to the application. RMP does not require a special multicast server or group manager. It is based on a token rotating scheme and on 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 tokens and acknowledgements are usually sent as multicasts. When other group members discover that they are missing some packets, they can send negative acknowledgements to the original sender.

We have realized a multi-channel, whose kind is «RMP», that implements the RMP algorithm by using remote method invocation rather than message passing as in the original algorithm. This implementation does not claim to have all the same properties of the original RMP (we concentrate our efforts on the total ordering property) but is sufficient to prove that mChaRM can be used to modify the underlying communication protocol.

By using a multi-channel, the RMP algorithm is scattered among the multi-channel components. There are two types of components involved in the RMP algorithm: the message sender and its receivers. Target loci play the role of receivers, acknowledging to each other the receipt of a message, and waiting for the acknowledgements from each of the other target loci before delivering the message to the receiver. When a locus receives an acknowledgement for a message not yet received, it asks the abstract locus to send the missing message again. The target section that implements the RMP protocol looks as follows:

```
  // Target locus mimics the role of the servers in the RMP algorithm.

 target-locus:
   private Hashtable acks = new Hashtable();

      // It checks if the message has really arrived
      // or it is only an ack. If it has not arrived
      // it sends a negative ack to the abstract
      // locus.
   public void pACK(int TS) {
     if (acks.get(TS)==null) abstract-locus.nACK(TS, whoAmI);
     else acks.put(TS, new Integer(acks.get(TS).intValue()+1));
   }

    // It is called on an ack receipt. It waits acks from the
other loci.
   private void ackTheMsg(int TS) {
     for(int i=1; i<howManyReceivers(); i++)
       target-loci(i).pACK(TS);
   }

    // Notifies other target loci upon message receipt.
   public void beforeReceiverSideMetaBehavior(
                              mChaRMMethodCall msg) {
     int TS = msg.removeArgument(0).intValue();
     acks.put(TS, new Integer(1));
     ackTheMsg(TS);
     while(acks.get(TS).intValue()<howManyReceivers());
   }
```

The abstract locus plays the role of sender, replacing the real one. It triggers the message propagation and when a target locus loses a message it sends the missing message again, but only to the interested locus. The abstract locus section looks like[9]:

[9] We do not explicitly discuss that, because irrelevant for the explanation, but in order to work as expected, multi-channels implementing the reliable multicast protocol have to be multi-threaded.

```
  // Abstract locus plays the role of the client in the RMP
algorithm.

 abstract-locus:
   private int localTS = 0; // local message time stamp
   private Hashtable messages = new Hashtable();

    // If called means that a target locus lost a message.
   public void nACK(int TS, String name) {
     mChaRMMethodCall msg = messages.get(TS);
     target-locus(get(name)).invoke(msg.name(),msg.args());
   }

      // It piggybacks the message with the
      // timestamp and forwards it to the target
      // loci.
   public Object coreMetaBehavior(mChaRMMethodCall msg) {
     msg.insertArgument(0, new Integer(++localTS));
     messages.put(localTS, msg);
     return super.coreMetaBehavior(msg);
   }
```

We have implemented a tight collaboration between abstract and target loci synchronizing the final message execution in a simple way. This example shows several aspects of mChaRM, e.g., how to manipulate messages (each message is piggybacked with a timestamp in order to distinguish them), how the multi-channel components cooperate, how to reflect on ingoing messages, and so on. This example characterizes many problems tied to reflect on ingoing messages, and that are hard to handle by the traditional reflective approaches.

7 Multi-Channel Perspectives

Given that the multi-channel approach maintains all the typical software development advantages (i.e., separation of concerns, code reuse improvement, transparent extendibility, and so on) that other reflective approaches offer, we focus on purposes, benefits, and drawbacks peculiar to the multi-channel approach. In particular, we focus on the gaps left open by other reflective middleware approaches but filled by our approach.

7.1 Multi-Channel Purposes.

The multi-channel reification model is especially designed for modeling and reifying communications. The design principle is to encapsulate and abstract communication mechanisms, and to permit modular enhancement of the semantics of communication. A multi-channel can easily extend the communication semantics to include new features such as tracing mechanisms, reliability control, fault tolerant behavior and so on. This approach also provides a framework for testing and simulating novel communication protocols.

Once communication mechanisms have been encapsulated into multi-channels, the meta-programmer is able to focus on enriching or replacing the behavior of the encapsulated mechanisms, without considering implementation details. The meta-programmer need not be concerned with issues such as: how to pass information from one multi-channel component to another, when and how to serialize and synchronize ingoing messages, and so on. In this way, the development of distributed systems is simplified, and novel behaviors can be attained by replacing either the current multi-channel or some of its components.

7.2 Multi-Channel Properties.

Basic properties such as separation of concerns and transparent extendibility are common to every reflective approach. In addition, our approach is characterized by a global view of the communication channel, a finer granularity of reflection and communication channels modeled by open/closed systems.

Global View. As advocated in [18] and stressed in section 2.2, object-oriented methodologies (including object-oriented reflective methodologies), are inadequate for managing distributed communications because they do not maintain a *global view* of the state of communications.

The multi-channel reification model provides an abstraction that moves the underlying communication mechanism into the meta-level. Implementation details of the communication mechanism are not directly exposed to the meta-programmer, but its behavior can be easily altered by managing the multi-channel behavior (see section 6). Multi-channels encapsulate the communication mechanism and reify all the aspects of the whole communication (i.e., sender, receivers and so on) into a single logical entity, i.e., the multi-channel itself. Multi-channels directly access both state and operations of each communicating object and the exchanged messages without coordinating their actions with other meta-entities. Each meta-program, using specific methods provided by an API, imbues messages circulating on the communication channel (trapped by the corresponding multi-channel) with extra behaviors. Each multi-channel owns all data related to the reified communication channel and to messages circulating through it. These data are directly at the meta-program's disposal. That is, the multi-channel approach maintains, at any time, a global view of communications.

Communications As Open/Closed Systems. The proposed approach moves the communication mechanism into the meta-level. Each object involved in a reified communication is really composed of two parts. One runs at the base-level and is developed by the application programmer. The second part is the reification of the first, runs at the meta-level — the implementation of the source and target loci — and is developed by the meta-programmer. The meta-level part abstracts all mechanisms for managing messages either sent by or received from any other object, and exposes such mechanisms to manipulation and redefinition by the meta-program. Multi-channels gather data related to messages filtered by them and monitor each communication phase. Hence, each multi-channel transforms the logical channel it reifies and messages passed through it in a *open/closed system* with few if any interactions with base-level entities or with other meta-entities external to the multi-channel itself. Such a complete separation improves reusability of novel communication features because

	single host	*ratio*	*two hosts*	*ratio*	*three hosts*
Java RMI	0.86 msec		1.15 msec		
KaRMI	0.77 msec	0.89	0.68 msec	0.59	
mChaRM (KaRMI)	1.85 msec	2.15 (2.40)	1.59 msec	1.38 (2.33)	1.37 msec
mChaRM (KaRMI+«compact»)	0.93 msec	1.08 (1.20)	0.79 msec	0.69 (1.16)	

Table 4. Evaluation of the mChaRM performance.

the code implementing the novel feature is part of the multi-channel code and does not need any other code. The separation of communication-related features from the rest of the functional code also simplifies experimenting, testing, and maintaining (novel) features related to communication channels. To change the semantics of a communication channel, i.e., to update the multi-channel associated with a communication channel, is enough for testing novel features. Multi-channels can be considered *open systems* because they are structured in an inheritance hierarchy and therefore each of its components can be easily extended and refined. They are also *closed systems* because they can be used as-is without interacting with external entities. This feature allows increasing abstraction and encapsulation of communication, still enforcing a global view of the communication state. The well-known composition problem [37][10] can be limited using multi-channels, because each multi-channel is a closed system and encapsulates the whole communication. In addition, the kind feature permits differentiation of meta-behaviors from behaviors' composition. Thus, it is likely that the meta-programmer needs to combine multiple meta-entities.

Granularity. We define *granularity of reflection* [10] as the smallest aspect (e.g., objects, methods, method calls and so on) of the base-system that can be reified by two different meta-entities. The multi-channel reification approach allows the meta-program to use a different multi-channel to reify each communication channel used in the base-level. Multi-channels span across several objects, reifying the communication channel they use, and filtering only specific message patterns. Multi-channels have a twofold granularity. From the point of view of a single object the granularity is at message level because different messages sent to or from the same object can be reified by different multi-channels. On the other hand, by considering all the objects involved in a communication, the granularity is at the communication channel level. This is because two multi-channels with a different kind can be established among the same group of objects, applying two different behaviors to messages using them. In general, a finer granularity improves the flexibility of the model. A finer granularity allows a more detailed control of the work of the base-level system, but at the cost of meta-entity proliferation: to observe the base-level system deeply, we need a greater number of observers (i.e., meta-entities).

[10] The composition problem is defined as the behavior conflict we have to face when we are trying to compose two or more meta-objects. Two meta-objects can work on the same information or can offer overlapped services with obvious synchronization problems.

7.3 Performance Evaluation.

Performance is a potential problem of the multi-channel reification approach, or of any reflective approach for realizing middleware. Each reflective mechanism is realized by trapping a call and hijacking it towards the meta-level. Such mechanisms introduce additional computation to each communication. Our approach is no different in that respect. However, it traps only those calls specified by the **kinds** clause, thereby reducing the performance impact. Moreover, transitions between base- and meta-level are dealt with by local method calls. Despite this, performance is not as good as expected because the adopted multi-channel architecture replaces the single explicit remote method invocation with two implicit remote method invocations (one from the sender stub to the core and one from the core to the receiver stub). To limit the performance impact, we have followed two strategies:

- adopting KaRMI [38] for dispatching messages among multi-channel components instead of standard Java RMI. KaRMI has proved itself to be more efficient than standard RMI. Every mChaRM application takes advantage of this strategy;
- providing an alternative multi-channel architecture, called *compact*, which merges the source with the abstract locus. Note that its use sacrifices some of the flexibility of the approach (see section 5.1).

We have quantified the overhead due to our architecture with respect to standard Java RMI. Our experiments are related to point-to-point communications. Experiments with multi-point communications are not germane because both standard Java RMI and KaRMI do not directly support this model of communications. The aim of our experiments consists of measuring how long mChaRM (both standard and compact architectures), Java RMI and KaRMI take to perform a remote method invocation. Please note that our experiments were also done using KaRMI to highlight the benefits that mChaRM has achieved by adopting KaRMI. All the experiments were performed on a network composed of AMD 1800+ PC's with 512Mb RAM running Linux (kernel version 2.4.8), `jdk v1.4` and KaRMI v1.06β.

Message Reification Overhead. The scenario of our experiments is composed of a receiver that returns a dummy message of small fixed size (to limit the overhead of marshaling and unmarshaling the message), and a sender that repeatedly requests that service. The mChaRM implementation involves two multi-channels of kind «`normal`» and «`compact`», which only trap the message and deliver it to the designated receiver without performing meta-computations.

Table 4 summarizes the results of our experiments. The second, fourth, and sixth columns represent how long the method

call takes by using a network composed of one, two, or three (only with mChaRM) computers. The third and fifth columns show the performance ratios between mChaRM or KaRMI and Java RMI; the performance ratios between mChaRM and KaRMI is reported enclosed in parenthesis. As expected, the mChaRM is slightly slower than the Java RMI framework, but its *compact* version, due to the use of KaRMI (cf. the second row of Table 4), improves the performances of a remote method invocation with respect to the standard Java RMI framework.

We are improving mChaRM's performance by developing our efficient group communication [12], parallelizing the effective dispatching of messages to the receivers, and reengineering the multi-channel architecture by using stream sockets instead of RMIs. Note that the focus of our work is on flexibility and reusability rather than efficiency.

8 Related Work

At the moment, several reflective middleware projects are under development, i.e., projects that try to mix reflection and distributed middleware. There are many approaches and as a result it is almost impossible to consider them all. However, most efforts focus on improving middleware adaptability, flexibility, and separation of concerns. GARF, and CodA are milestones in the reflective middleware research area, whereas systems such as: OpenORB, OpenCORBA, and DynamicTAO represent new approaches.

GARF [20,21] lies between the system layer (which supplies the distribution mechanisms) and the application layer. Its basic purpose consists in wrapping the distribution primitives and supplying a uniform and abstract interface to them. Hence, GARF does not implement the distribution environment but it masks each access to it and offers the programmer a tool for changing and enriching the basic behavior. The GARF model is derived from the meta-object model [32] with some adaptations for communications management. It links a meta-object, called *encapsulator*, to the referents. Two encapsulators communicate via another meta-object, called *mailer*. This architecture provides safe communication customizability via a built-in library containing several mailers and encapsulators. As the authors state, mailers can only control out-going messages, thus an object has to receive the message in order to decide whether to take care of it or not. Additionally, GARF mimics each communication in the meta-level, decoupling the communication-related code from the application code. Unfortunately, each communication is reified by three separated meta-objects: a mailer and two encapsulators. The mailer carries out computations on the trapped message, while the encapsulators carry out the actual inter-process communication. Such a decomposition does not permit any of them to have a global view of the whole communication.

CodA [34,35] provides a distributed framework from scratch. Distribution is entrusted to the meta-level and each object is reified by seven meta-objects. Each meta-object manages a specific distribution aspect (e.g., message queue, sending, and so on). This meta-level factorization simplifies the definition of new behaviors for specific distribution aspects. Unfortunately, it also jeopardizes the mechanism's flexibility by selecting which aspects might be managed. Also, it might complicate the meta-program code, which is scattered among several meta-objects and might render the composition and coordination of the behavior of so many meta-objects error-prone. Meta-objects reify and reflect on how an object has to manage a message, and not on the specific semantics of message management and related aspects. Hence, the CodA framework lacks a global view, although the author suggests it is achieved by communications and cooperation among meta-entities. However, a simple mechanism to vary the applied meta-behavior according to the single exchanged message is not provided.

OpenCORBA [31] implements the CORBA API in NeoClasstalk. It reifies various properties of CORBA's brokers through explicit meta-classes, allowing customization of some internal characteristics of the CORBA ORB. Meta-classes permit the modification and reconfiguration of the ORB mechanisms. OpenCORBA is a reflective ORB, which allows the programmer to dynamically adapt the behavior of the broker. Two basic aspects of the CORBA software bus have been reified: the remote invocation mechanism via a proxy, and 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. As defined by the CORBA specification [39], both the remote invocation mechanism and type checking are completely independent of the functionalities provided by the server class. Therefore, they can be separated from the base code and constituted as a class property that is implemented by meta-classes (respectively `ProxyRemote`, and `TypeChecking`). The dynamic adaptability both of the invocation mechanisms, and of the type checking in OpenCORBA is achieved by extending default meta-classes `ProxyRemote`, and `TypeChecking`. Adaptability is however limited to only two communication aspects. The global view concept is not provided and several communication aspects (such as brokers, name service, and so on) are not accessible to (and can not be manipulated by) the meta-program, but it should be simple to extend the OpenCORBA system in order to open up such aspects.

DynamicTAO [29,30] extends the TAO [40] system providing a CORBA compliant 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 run-time, 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. This architecture permits consistent strategy reconfiguration. Component implementations are shipped as dynamically loadable libraries linked to the ORB process at run-time. They are organized in categories representing different aspects of the ORB internal engine or different types of servant components. The `Dynamic Service Configurator` contains the `DomainConfigurator` that supplies common operations for the dynamic configuration of components at run-time. It delegates some of its functions to specific component configurators.

The DynamicTAO approach is quite different from that adopted by other reflective middleware. This approach is very low-level and quite powerful in order to change the strategies the system adopts. Unfortunately, it is neither simple nor powerful when programmers want to extend the communication behavior with features not directly related to strategies already involved in communication, e.g., to introduce a message checkpointing policy from scratch.

OpenORB [14, 16] adopts a component-based model of computation. Its component model supports components with interfaces, interfaces for continuous media, and the creation of explicit bindings between compatible interfaces. The meta-level exposes the actual implementation. This approach makes it possible to access the meta-level of a meta-level. It supports per-interface meta-spaces. Several meta-spaces govern all the aspects of the underlying system. The content of a component is represented by two distinct meta-models, namely the encapsulation and the composition meta-models. The activities (message arrival, enqueueing, and so on) of the underlying system are represented by the environmental meta-model. This approach also provides a fine level of control over the support provided by the middleware platform.

OpenORB provides an interesting and powerful approach to adaptation and to the development of reflective middleware. Meta-spaces provide a flexible mechanism to reify and to reflect on any system aspect, but it is component-oriented and it behaves similarly to object-oriented models in that it neglects the handling of communications as a whole. Notwithstanding this fact, their approach provides fine-grained reflection which permits varying the behavior of a single component. Explicit bindings reify communication channels, but not the involved components. They have a low-level approach, handling exchanged messages as a stream instead of higher-level abstractions. So, it is easy to filter or to handle them as a whole, but is difficult to manage them at a higher level, e.g., piggybacking information.

Aspect-Oriented Programming. Aspect-oriented programming (AOP) [28] is an alternative approach to implementing a separation of concerns. This approach provides the separation between cross-cutting concerns during development of a system (in our case, cross-cutting concerns are about communications). The program elements representing the cross-cutting concerns are *woven* by the compilation process so that the separate elements do not remain separate at run-time. AOP as well as compile-time reflection (that is, reflective changes to the code carried out during compilation, e.g., OpenJava performs compile-time reflection) is more efficient than run-time reflection (that is, changes to the program carried out during execution, e.g., Java core reflection and mChaRM perform changes at run-time), but it loses this advantage when meta-objects' behavior depends on dynamic information as message content. Another point about AOP is that it encapsulates code that implements cross-cutting concerns in a single module and at compile-time the code is *woven* into the application code. In contrast, although mChaRM also encapsulates cross-cutting concerns in a single module, we maintain a separation at all times between the cross-cutting code and the application code. One advantage of our approach is that it allows for dynamic adjustment of communication semantics, unlike an AOP approach.

Composition Filters [2] has been a forerunner of the AOP methodology. Akşit et al. provide an *actor-based* framework [1] that associates an actor description with a set of filters for ingoing and outgoing messages. These filters manage the exchanged messages (altering the overall behavior of the system), but they do not offer an abstraction mechanism that permits getting a global view of communication. Furthermore, they neither allow manipulation of the actors' state nor offer a mechanism to keep a history of message manipulations.

9 Conclusions and Future Work

The role of reflection in monitoring object communication in complex distributed applications is a crucial issue. We have presented a reflective model, called multi-channel reification, that abstracts and encapsulates inter-object communications and enables the meta-programmer to enrich and/or replace the predefined communication semantics. The approach provides a way to reify the abstract concept of communications into a logical entity, i.e., communication channels become first-class citizens. It allows programmers to access communication details without writing the code necessary to realize them. The multi-channel approach fills most of the gaps of the traditional reflective approaches. Its main achievement consists of improving the global view of communication without compromising encapsulation. The approach also offers a finer reification/reflection granularity than previous approaches, and a simplified approach to the development of communication-oriented software. By allowing an object to hook itself to several multi-channels, the multi-channel reification approach achieves a finer granularity than with an object-based approach. However, we also have to deal with consistency problems. In fact, some of the multi-channels hooked to a base-object could decide to carry out intercession on the state of their referent, and without a careful synchronization of the accesses the state might become inconsistent or incorrect (due to the well known *race-condition problem*). For this reason, we have limited multi-channel intercession on its referents. In the future we are going to introduce a mechanism to implicitly lock

and unlock the state of the multi-channel referents and to improve the intercession mechanism of the approach.

The proposed model is realized by a framework, called mChaRM, written in Java by means of the language extension capabilities of the OpenJava [13] reflective tool. The usability of the approach is shown by prototyping some non-trivial applications [3, 4]. A simple API for supporting the development of meta-programs has also been provided. The mChaRM system can be downloaded from `http://www.disi.unige.it/person/CazzolaW/mChaRM_webpage.html`.

In addition to API improvements and bug fixes, we are analyzing how to improve the performance of our system. As shown in section 7.3, mChaRM is very flexible and facilitates addition of extra features to communications, but its weak point is performance. Obviously, we have to pay for flexibility, but we are sure that performance of the current implementation can be improved by using Java proxies and applying code localization. Other future work consists of designing a composition relation among multi-channel loci in order to create new kinds of multi-channels by means of existing locus composition (for example, by composing a future-based source locus with a reliable multicast protocol). This would further enhance the reusability and readability of the meta-level code. Finally, we are planning to extend mChaRM with a native support for *futures* [22, 26] in order to capture asynchronous as well as synchronous communication models.

Acknowledgements. The author wishes to thank Prof. Massimo Ancona from the University of Genova and Prof. Francesco Tisato from the University of Milano Bicocca for having introduced me to reflection and middleware; Prof. Shigeru Chiba from Tokyo Institute of Technology both for his interest in this work and for his valuable contribution to the ideas presented in this paper; Dr. Ira R. Forman from IBM research labs at Austin, Frank Morgan and Prof. Hanan Samet from the University of Maryland and Ian S. Welch from the University Victoria of Wellington for their inestimable help in rendering this work in English. The author has also to thank Prof. Gordon S. Blair, Dr. Geoff Coulson, Dr. Giovanni Lagorio, Prof. Robert J. Stroud, and Dr. Emiliano Tramontana for their advice and to have been the proof-readers of previous drafts of this paper. Finally, the author wishes thank the anonymous reviewers for their useful advice and comments.

References

1. Gul Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, Cambridge, 1986.
2. Mehmet Akşit, 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. Massimo Ancona, Walter Cazzola, and Eduardo B. Fernandez. A History-Dependent Access Control Mechanism Using Reflection. In Peter Sewell and Jan Vitek, editors, *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.
4. 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 35–49. Springer-Verlag, July 1999.
5. Ken Arnold and James Gosling. *The Java Programming Language*. The Java Series ... from the Source. Addison-Wesley, Reading, Massachusetts, second edition, December 1997.
6. Paul Bassett. *Framing Software Reuse: Lessons from the Real World*. Prentice Hall, 1996.
7. Gordon S. Blair and Roy Campbell. Proceedings of the Workshop on Reflective Middleware. Available on-line at `http://www.comp.lancs.ac.uk/computing/RM2000/`, April 2000.
8. Daniel G. Bobrow, Richard G. Gabriel, and Jon L. White. CLOS in Context - The Shape of the Design Space. In Andreas Pæpcke, editor, *Object Oriented Programming: The CLOS Perspective*, pages 29–61. MIT Press, 1993.
9. 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.
10. 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.
11. Walter Cazzola. *Communication-Oriented Reflection: a Way to Open Up the RMI Mechanism*. PhD thesis, Università degli Studi di Milano, Milano, Italy, February 2001.
12. Walter Cazzola, Massimo Ancona, Fabio Canepa, Massimo Mancini, and Vanja Siccardi. Enhancing Java to Support Object Groups. In *Proceedings of the Third Conference on Recent Object-Oriented Trends (ROOTS'02)*, Bergen, Norway, on 17th-19th of April 2002.
13. 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.
14. 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.
15. Geoff Coulson. What is Reflective Middleware? In IEEE Distributed Systems On-Line, 2000. `http://boole.computer.org/dsonline/middleware/RM.htm`.
16. Geoff Coulson, Gordon S. Blair, Michael Clarke, and Nikos Parlavantzas. The Design of a Configurable and Reconfigurable Middleware Platform. *Distributed Computing Journal*, 15(2):109–126, April 2002.
17. 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.
18. Mohamed E. Fayad and Rachid Guerraoui. OO Distributed Programming Is Not Distributed OO Programming. *Communications of the ACM*, 42(4):101–104, April 1999.

19. 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.
20. Benoît Garbinato, Rachid Guerraoui, and Karim R. Mazouni. Distributed Programming in GARF. In Rachid Guerraoui, Oscar Nierstrasz, and Michel Riveil, editors, *Object-Based Distributed Programming*, LNCS 901, pages 1–32. Springer-Verlag, 1994.
21. Rachid Guerraoui, Benoît Garbinato, and Karim R. Mazouni. GARF: A Tool for Programming Reliable Distributed Applications. *IEEE Concurrency*, 5(4), October-December 1997.
22. Robert H. Halstead Jr. Multilisp: A Language for Concurrent Symbolic Computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, October 1985.
23. A. Richard Helm, Ian M. Holland, and Dipayan Gangopadhyay. Contracts: Specifying Behavioral Compositions in Object-Oriented Systems. In *Proceedings of Object-Oriented Programming Systems, Languages, and Applications Conference*, Special Issue of Sigplan Notices, pages 169–180, Ottawa, Canada, October 1990. ACM Press.
24. Ian M. Holland. Specifying Reusable Components Using Contracts. In Ole Lehrmann Madsen, editor, *LNCS*, number 615 in Proceedings of the 6th European Conference on Object-Oriented Programming (ECOOP'92), pages 287–308, Utrecht, the Netherlands, July 1992. Springer-Verlag.
25. Walter Hürsch and Cristina Videira Lopes. Separation of Concerns. Technical Report NU-CCS-95-03, Northeastern University, Boston, February 1995.
26. Morry Katz and Daniel Weise. Continuing into the Future: On the Interaction of Futures and First-Class Continuations. In *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, pages 176–184, Nice, France, June 1990. ACM Press.
27. Gregor Kiczales, Jim des Rivières, and Daniel G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, Cambridge, Massachusetts, 1991.
28. Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-Oriented Programming. In *11th European Conference on Object Oriented Programming (ECOOP'97)*, Lecture Notes in Computer Science 1241, pages 220–242, Helsinki, Finland, June 1997. Springer-Verlag.
29. 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.
30. 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.
31. 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.
32. 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.
33. Satoshi Matsuoka, Takuo Watanabe, and Akinori Yonezawa. Hybrid Group Reflective Architecture for Object-Oriented Concurrent Reflective Programming. In Pierre America, editor, *Proceedings of ECOOP'91*, pages 231–250, Geneva, Switzerland, July 1991. Springer-Verlag.
34. 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.
35. 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.
36. Todd Montgomery. RMP - Reliable Multicast Protocol version 2. Document Available from `http://research.ivv.nasa.gov/projects/RMP/index.html`, October 1996.
37. Philippe Mulet, Jacques Malenfant, and Pierre Cointe. Towards a Methodology for Explicit Composition of MetaObjects. In *Proceedings of the 10th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'95)*, volume 30 of *Sigplan Notice*, pages 316–330, Austin, Texas, USA, October 1995. ACM.
38. Christian Nester, Michael Philippsen, and Bernhard Haumacher. A More Efficient RMI for Java. In *Proceedings of ACM 1999 Java Grande Conference*, pages 152–157, San Francisco, California, June 1999.
39. Object Management Group. The Common Object Request Broker: Architecture and Specification. Technical Report 2001.02.01 Revision V.2.4.2, OMG, February 2001.
40. Douglas C. Schmidt and Chris Cleeland. Applying Patterns to Develop Extensible ORB Middleware. *IEEE Communications Magazine Special Issue on Patterns*, 37(4), April 1999.
41. Robert J. Stroud. Transparency and Reflection in Distributed Systems. *ACM Operating System Review*, 22:99–103, April 1992.
42. SUN Microsystems. Java™ Core Reflection API and Specification. Technical report, SUN Microsystems, February 1997.

Walter Cazzola is currently an assistant professor at the Department of Informatics and Communication (DICo) of the Università degli Studi di Milano, Italy. Previously, Cazzola was researcher assistant at the Department of Informatics and Computer Science (DISI) of the Università degli Studi di Genova. His research interests include computational reflection, programming techniques and languages, and distributed systems. He has written and has served as reviewer of several technical papers about computational reflection. He has also taken part of the programme committee of the International Conference Reflection 2001. Cazzola received his Laurea in Computer Science from the Università degli Studi di Genova, Italy in 1996; Ph.D. in Computer Science from the Università degli Studi di Milano, Italy in 2001.