Communication Modeling by Channel Reification

M. Ancona[†] **W. Cazzola**[‡] **G. Dodero**[†] **V. Gianuzzi**[†]

[†] DISI-University of Genova, Via Dodecaneso 35, 16146 Genova, Italy
Tel: `(+39) 10 353 660`{5|3|2} Fax: `(+39) 10 353 6699`
E-mail: {`ancona`|`dodero`|`gianuzzi`}`@disi.unige.it`

[‡] DSI-University of Milano, Via Comelico 39-41, 20135 Milano, Italy
Tel: `(+39) 10 353 6709` Fax: `(+39) 10 353 6699`
E-mail: `cazzola@dotto.usr.dsi.unimi.it`

April 7, 1997

Abstract

The paper presents a new reflective model, called Channel Reification, which can be used to implement communication abstractions. After a brief review of existing reflective models and how reflections can be used in distributed systems, channel reification is presented and compared to the widely used meta-object model. An application to protocol implementation, and hints on other channel applications are also given.

Keyword: Object-Oriented, Computational Reflection, Reflective Distributed Systems.

1 Introduction

Reflective object-oriented programming paradigms have been proposed for developing incremental solutions to complex applications. A reflective object-oriented system is capable of monitoring its own behaviour, a more precise definition is given below. What makes reflection especially attractive in the design of complex systems is that it allows a clear separation between the application (problem dependent) and meta (dealing with implementation) functionalities, using a meta-level to hide complex implementation details from the application programmer.
Models for reflective systems appearing in the literature are: messages reification, meta-objects and meta-classes. A complete presentation is given in [6]. However, there are problems arising in the implementation of distributed communication for

which none of the above models provides simple and clear solutions.
For this reason we have defined a new model for reflective object systems, which we have called channel reification: its purpose is that of encapsulating and possibly redefining object interaction models. Examples include the definition of object communication protocols in distributed systems, where interaction modalities can be made transparent to the user: from tightly coupled interactions in a multiprocessor, to connections via geographic networks. Transparency may be exploited to find new interaction modalities with existing applications, moving them to a networked or distributed environment, or superimposing properties like reliability and fault tolerance.
The channel reification model supports creation of a library of channel classes, each instance of them being a communication channel. Different implementations with different semantics or performance may be made available within a channel library. Thus channel reification provides a unified and application transparent view of the underlying communication subsystem.
The rest of the paper contains an overview of computational reflection (section 2), a short discussion on the use of reflection in distributed environments (section 3), a presentation of channel reification (section 4) and, in section 5, a channel application is presented, concluding with some hints on how to implement channels.

2 Computational Reflections

Computational reflection or simply reflection is defined as the activity performed by an agent when doing computations about its own computation [11].
An object-oriented reflective system is logically structured in two or more levels, constituting a *reflective tower*. Entities (objects) working in the base level, called base-entities, define the system basic behaviour. Entities working in the other levels (meta-levels), called meta-entities, perform the reflective actions and define further characteristics beyond the application dependent system behaviour.
Each level is causally connected to adjacents levels, i.e. entities working into a level have data structures representing (or, using a reflection-like term, reifying) the activities of entities working at the underlying level and their actions are reflected into such data structures. Any change to such data structures modifies entity behaviour.
Meta-entities supervise base-entities activity. The concept of *trap* could be used to explain how supervision takes place. Each base-entity action is trapped by a meta-entity, which performs a meta-computation, then it allows such base-entity to perform the action.
We observed, going beyond the reflective tower of compilers|interpreters, that each reflective computation can be separated into two logical aspects: computational flow context switching and meta-behaviour. A computation starts with the computational flow in the base level; when the base-entity begins an action, such action is trapped by the meta-entity and the computational flow raises at meta-level (*shift-up* operation). Then the meta-entity completes its meta-computation, and when it allows the base-entity to perform the action, the computational flow goes back to

the base level (*shift-down* operation).
The use of meta-level programming permits transparent separation of application components from those providing additional properties to the application (separation of concerns). To this respect, it is useful to consider also *reflections granularity* [2], that is the minimal entity in a software system for which a reflective model defines a different meta-behaviour. A finer granularity allows more flexibility and modularity in the software system at the cost of meta-entity proliferation.
We now briefly describe different models for reflection, highlighting their advantages and limits.

2.1 Meta-Object Model

In this model, meta-entities (called meta-objects) are objects, instances of a proper class. Each base-entity, called also referent, can be bound to a meta-object. Such a meta-object supervises the work of the linked referent. The model makes few assumptions about relationships between base and meta-entities: in principles, each meta-object can be connected to many referents, and each referent can be linked to several meta-objects (one at a time) during its lifecycle. However most implementations, for reason of efficiency, restrict this freedom: in OpenC++ [4] and ABCL-R [14] a meta-object is linked to one referent only, and each referent can have only one meta-object during its lifecycle. As a consequence, reflection granularity is at object level.

2.2 Message Reification Model

In this model, meta-entities are special objects, called messages, which embody the actions that should be performed by the base-entities. The kind of a message defines the meta-behaviour performed by the message; different messages may have different kinds. At every method call, a message is created, in agreement with the kind of the meta-computation required, and when the meta-computation terminates, such a message is destroyed.
Then, granularity is at method level, since it is possible to define different behaviours for method calls performed by each object. Messages are not linked to the base-entity originating them and cannot access their structural information. Message lifecycle is the duration of the embodied action. Thus it is impossible to store information among meta-computations (lack of information continuity). On the other hand, every method call creates and then destroys an object (the message).

3 Reflection and Distribution

Distributed architectures let users of individual, networked computers share programs and data resources. Distribution can also enhance avaibility, reliability and performance (through techniques such as replication of programs or data and parallel computation). In achieving these benefits, distributed systems incur design costs that are not present in unitary systems.

Critical design issues to be solved in distributed systems include locating programs and data resources across the network, establishing and maintaining interprogram communication on the network, coordinating the execution of distributed applications.

Coordination models [1] represent one way to handle these diverse design issues coherently and uniformly. A coordination model establishes logical rules for executing distributed interactions. Rules specify who can initiate interactions, who can respond, how to retrieve results, how to handle errors, and so on.

Client|server architecture represents a widely used coordination model: an object, the client, requests an operation or service that another object, the server, is able to provide. This coordination model offers simplicity in closely matching data with control flow.

To achieve a better and clear separation of the application code from the interaction code, a different model, employing *agents* and *brokers* is increasingly used.

While an **agent** is a distinct, architectural component that mediates interactions between an application and the communications kernel, a **broker** is a dedicated control mechanism that mediates interactions between client applications needing services and server applications providing them. Status of services is maintained and recovered by the broker, so clients no longer need keep track about where and how to obtain particular services.

As remarked in [13], today's distributed systems either include coordination code very tightly coupled with the application code or completely separated from it, working transparently and out of designer's control.

Using computational reflection, the coordination model can be implemented at meta-level. Meta-entities use objects to encapsulate coordination model entities (brokers or agents). In this way, coordination code is clearly separated from application code (within base-entities) and the programmer may *customize* the coordination model without affecting application code (one such example is the *Object Communities* described in [5]). As a consequence, many coordination models may simultaneously be present in a system (one for each meta-entity), and meta-entities implementing such models, once implemented, can be easily changed and reused in different applications.

In order to achieve the desired goals, reflective distributed systems design should solve new problems, such as how to interface entities and meta-entities at the same time preserving transparency, and how to implement causality without sacrificing system efficiency. Such topics are examined in [10] and [12].

4 Channel Reification Model

We propose this model as an extension to the message reification model, aimed at solving some of its drawbacks, while keeping its advantages. Channel reification is based on the following idea: a method call is considered as a message sent through a logical channel established between an object requiring a service, and another object providing such a service. This logical channel is reified into an object called channel (as shown in figure 1). A channel is characterized by a triple composed by

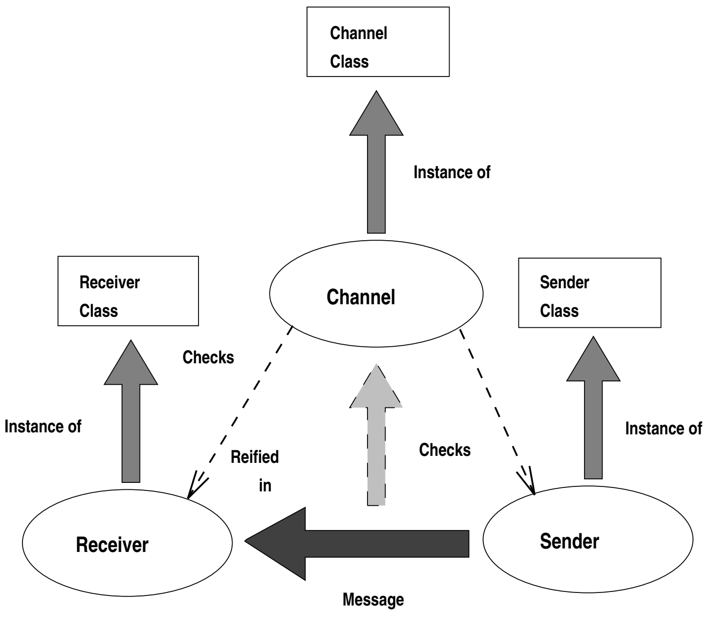

Figure 1: **Channel Reification Model Scheme**

the objects it connects and by the kind of the meta-computation it performs.

$$\text{channel} \equiv (\text{sender, receiver, channel\_kind})$$

A *channel kind* identifies the meta-behavior provided by the channel. In a typed object-oriented language the kind is also the type of the channel class. The kind is used to distinguish the reflective activity to be performed: several channels (distinguishable by the kind) can be established between the same pair of objects at the same moment.

The lack of information continuity of message reification is eliminated by making channels persist after each meta-computation. A channel is reused when a communication characterized by the same triple is generated. In this way, meta-level objects are created only once (when they are activated for the first time), and reused whenever possible. When an object is destroyed, all channels established from|to it are destroyed too. This lifecycle limits channel proliferation, since a garbage collector erases pending channels.

The features of the model are:

- ⋆ Method-level granularity, as for message reification: different method calls can be handled by different channels, thus specializing a reflective behaviour for each method.
- ⋆ Monitored channel proliferation with pending channels elimination.
- ⋆ Possibility to keep information among meta-computations (information continuity).

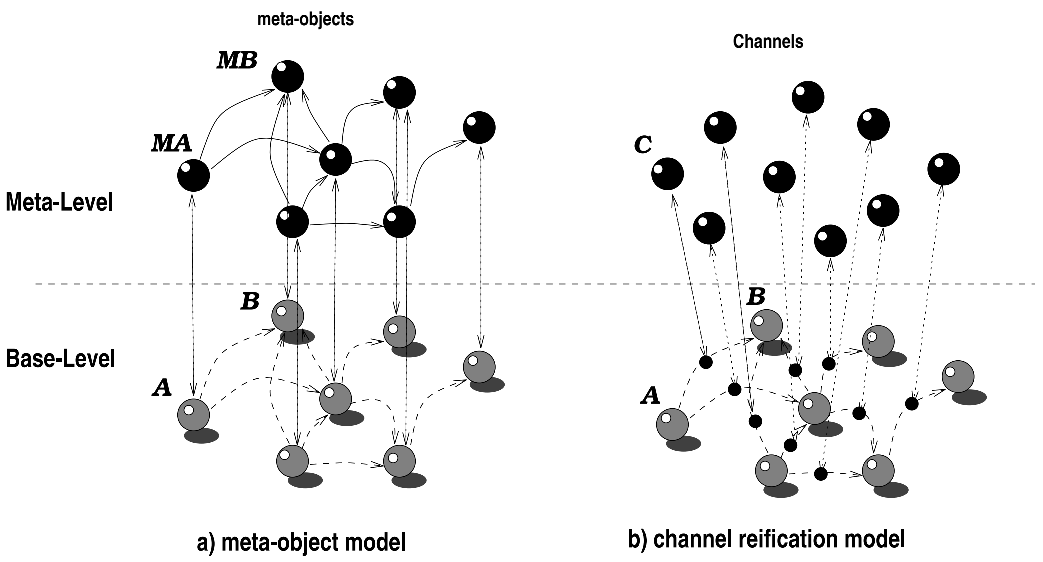

Figure 2: **Meta-Object vs Channel Reification**

⋆ Each channel completely supervises a communication, from the beginning to the end, sender and receiver's work inclusive.

Each service request is trapped (shift-up action) by the channel of the specified *kind* connecting client and server objects, if it exists. Otherwise, such a channel is created; in either case, it then performs its meta-computation and transmits the service request to the supplier. The server's answer is collected and returned to the requiring object (shift-down action).
A channel behaves like a reflective broker. Each channel kind specializes the behavior of a broker to specific requirements, and this specialization is transparent from the underlying application.

4.1 Comparing Channels to Meta-Objects

Channel Reification has been presented in the previous section as an extension to the message reification model. However channel persistency makes the new model more similar to the meta-object model. The difference between meta-objects and channels lies in their intended use, that is, a channel reifies and monitors communication between two objects, while a meta-object controls the behaviour of one specific object (as shown in figure 2). Meta-objects may be used to monitor communication as well, but only by means of cooperative actions with other meta-objects. In a distributed environment, the channel is a reflective abstraction of the broker model, mediating interactions between client applications needing services and server applications providing them. On the other hand, a meta-object plays a role closer to an agent, mediating interactions between applications and the communication kernel.

Thus the basic difference consists in the meta-communication protocol: in the meta-object model, in order to monitor interactions among several base objects, we must reify as many meta-objects as there are base objects. Such meta-objects interact via a communication graph which duplicates (or includes as subgraph) the communication graph at base level (see figure 2.a). In the channel reification model, at the meta-level, we reify object interactions (instead of single objects) that need not communicate between themselves (see figure 2.b). The channel communication graph, if any, is usually different from that at base level. This simplifies the reflective tower communication model at the expense of a larger number of reified meta-entities.
We believe that each application should make use of the reflection model better suited to its needs, the meta-object, the channel, or even both, in accordance with the kind of support required by useful abstractions.

5 When and How to Use Channel

Channel reification is designed for distributed communication. Channels are well suited *wrappers* for several communication abstractions that can thus be layered over application software:

- reliable messaging – obtained by encapsulating into channels all mechanisms to achieve the degree of communication reliability required by each application, instead of letting each applicative process provide its own mechanisms;
- implementing binding methods (eg., those supported by CORBA [9] or by OSF DCE [7] name service), thus relieving applications of the error prone task of retrieving and connecting to a *compatible server*;
- usual channel services such as: data marshaling|unmarshaling, communication error handling, multipacked message management.

Channels may be used to modify and specialize the service request formality, the next example shows how.

5.1 Implementing Protocols with Channels

Taking advantages of channel granularity, within the same object some services can be synchronous, others asynchronous, or loosely synchronous. The application programmer need not worry about synchronizing client with server and how to do it, the application need only specify a protocol to be used at each service request.
Such a behaviour is achieved by developing several channels classes (making a channel class library), one for each service protocol (for example, synch_channel and asynch_channel) and specifying for each service which protocol must be used.
Each channel class defines the kind of its own instances. In order to discriminate the protocol service it is necessary to bind each service to a channel kind. If the implementation language allows it, dynamic binding can be used to perform run-time selection of channel kind among available channel implementations.
For example, a server is offering services a, b, c, and d. At server start-up, we

may issue a binding request that all channels connecting any client to services a, c should be of kind synch_channel, those connecting to service d should be of kind asynch_channel, and those connecting to b should be as the client selects.

6 Conclusion and Future Works

The paper has illustrated channel reification, a new model for distributed reflective systems. As the name suggests, this model is especially suited for reflecting on communication among objects. Other reflective models have been compared to Channel Reification, and special attention has been devoted to the widely used meta-object model.
This new approach can be used to implement communication abstractions in order to extend communcation features of existing systems. The definition of a channel class library, where channel properties are selected by channel kind, would provide a significant simplification in application objects code.
The possibilities offered by channel reification are exploitable only by means of an efficient implementation of a channel library on widely available systems. A study of object-oriented and object based languages which would be suitable to support efficient channel implementation is presented in [3].
In such thesis the possibility to implement reflective systems is considered within several programming languages, such as SmallTalk, C++, Oberon and respective dialects (Oberon2, OpenC++ and so on). A minimal set of mechanisms needed for building up reflective systems into such languages has been studied. Such mechanisms should implement the shift-up and shift-down actions and consist on two primitives for context switching between meta and base level. Furthermore a prototype C++ implementation of channels on top of PVM [8] is also illustrated.
In another paper we discussed the use of channels in point-to-point communication. However the most challenging extensions are those towards multi-point communications, where significant applications are: multiple-RPC, broadcasts and object group communication, client request serialization and load balancing.
Modellization of extensions such as the above ones, towards multi-point communication is planned for the next future, at the same time keeping into account the implementation costs of these models.

References

[1] Richard M. Adler. Distributed coordination models for client/server computing. *IEEE Computer*, pages 14–22, April 1995.

[2] Massimo Ancona, Walter Cazzola, Gabriella Dodero, and Vittoria Gianuzzi. Channel reification: a reflective approach to fault-tolerant software development. In *OOPSLA'95 (poster section)*, Austin, Texas, October 1995. ACM. Available via anonymous ftp at `ftp.disi.unige.it/ftp/person/CazzolaW`.

[3] Walter Cazzola. Channel reification: a new reflective model. Analysis and comparison with other models and application to fault tolerant system. Master's

thesis, University of Genova – Department of Computer Science (DISI), April 1996. (Written in Italian).

[4] Shigeru Chiba. A meta-object protocol for C++. In *proceedings of OOPSLA'95*, Sigplan Notices, Austin, Texas, October 1995. ACM.

[5] Shigeru Chiba and Takashi Masuda. Design an extendible distributed language with a meta-level architecture. In *proceedings of ECOOP'93*, pages 482–501, Kaiserslautern, Germany, July 1993. Springer-Verlag.

[6] Jacques Ferber. Computational reflection in class based object oriented languages. In *proceedings of OOPSLA'89*, Sigplan Notices, pages 317–326. ACM, October 1989.

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

[8] Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, and Vaidy Sunderam. PVM 3 User's guide and reference manual. Technical Report ORNL/TM-12187, Oak Ridge National Laboratory, Oak Ridge, Tennesee 37831, May 1994.

[9] Object Management Group. Common object request broker architecture and specification. Technical Report 96.3.4 Revision 2.0, OMG, 1995.

[10] Shinji Kono and Mario Tokoro. Parallel reflection. Technical memo SCSL-TM-90-011, Sony CSL, June 1991.

[11] Pattie Maes. Concepts and experiments in computational reflection. In *proceedings of OOPSLA'87*, Sigplan Notices. ACM, October 1987.

[12] Hidehiko Masuhara, Satoshi Matsuoka, Takuo Watanabe, and Akinori Yonezawa. Object-oriented concurrent reflective languages can be implemented efficiently. In *proceedings of OOPSLA'92*, Sigplan Notices, Vancouver, Canada, October 1992. ACM.

[13] Satoshi Matsuoka, Takuo Watanabe, and Akinori Yonezawa. Hybrid group reflective architecture for object-oriented concurrent reflective programming. In *proceedings of ECOOP'91*, pages 231–250, Switzerland, July 1991. Springer-Verlag.

[14] Takuo Watanabe and Akinori Yonezawa. Reflection in an object-oriented concurrent language. In *proceedings of OOPSLA'88*, Sigplan Notices, pages 306–315, San Diego, September 1988. ACM.