Channel Reification: A Reflective Model for Distributed Computation

M. Ancona[†] **W. Cazzola**[‡] **G. Dodero**[†] **V. Gianuzzi**[†]
[†] **DISI**-University of Genova, Via Dodecaneso 35, 16146 Genova, Italy
E-mail: {ancona|dodero|gianuzzi}@disi.unige.it

[‡] **DSI**-University of Milano, Via Comelico 39-41, 20135 Milano, Italy
E-mail: cazzola@dsi.unimi.it

***Abstract*—The paper presents a new reflective model, called Channel Reification, which can be used in distributed computations to overcome difficulties experienced by other models in the literature when monitoring communication among objects.**
The channel is an extension of the message reification model. A Channel is a communication manager incarning successive messages exchanges by two objects: its application range between those of message reification and those of meta-object model.
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. Applications of channel reification to protocol implementation, and to fault tolerant object systems are shown. Future extensions to this model are also summarized.

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

I. Introduction

In the development of complex distributed applications, two concerns must simultaneously be kept into account: the problem which the application is intended to solve as well as how it shall be implemented. Among the latter types of concerns, portability, efficiency and fault tolerance may significantly impact on problem solution architecture.
In some cases, the underlying operating systems may support fully application transparent distribution and error processing facilities, or it is possible to integrate similar features as library modules. Unfortunately such support may be insufficient or unavailable on the selected platform. Then, specific tools must be developed at the same time as the application itself.
Object-oriented programming paradigms have been proposed as means intended to solve the above problems, and among object-oriented paradigms special attention should be given to that of *computational reflection*. A reflective object-oriented system is capable of monitoring its own behavior, 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. Such a feature improves reusability and extendibility of a system.
In a reflective system it is possible to modify application behavior in a transparent way, for instance by replicating objects for fault tolerance, without concerning the application designer with too much details about implementation of replication: choice of desired behavior is as simple as, say, object selection and instantiation from a given class.
Models for reflective systems appearing in the literature are: messages reification, meta-objects and meta-classes. A complete presentation is given in [1]. Meta-objects have been studied, for instance in [2], where an example of object replication for fault tolerance is detailed: the usefulness and ease of implementation are clearly shown. However there are other problems arising in the implementation of fault tolerant 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 main use is that of encapsulating and possibly redefining communication protocols in a distributed environment.
The channel reification model supports creation of a library of channel classes, each instance of them being a communication channel. Possibilities include reliable and atomic communication, distributed transactions, name service and load balancing. 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 II), a short discussion on the use of reflection in distributed environments (section III), a presentation of channel reification (section IV) and, in section V, some applications of the new model are presented. Finally, section VI is devoted to conclusions and future work in this area.

II. Computational Reflections

Computational reflection or simply reflection is defined as the activity performed by an agent when doing computations about its own computation [3].
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 behavior. Entities working in the other levels (meta-levels), called meta-entities, perform the reflective actions and define further characteristics beyond the application dependent system behavior.
Each level is causally connected to adjacent levels, i.e. entities working into a level have data structures representing (or, using a reflection-like term, *reifying*) the activities of the entities working into the underlying level and their actions are reflected into such data structures. Any

change to such data structures modifies entity behavior.
Meta-entities supervise the base-entities activity. The concept of *trap* could be used to explain how supervision takes place. Each base-entity action is trapped by a meta-entity, which performs a meta-computation, then it allows such base-entity to perform the action.
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-behavior. A computation starts with the computational flow in the base level; when the base-entity begins an action, such action is trapped by the meta-entity and the computational flow raises at meta-level (*shift-up* 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* [4], that is the minimal entity in a software system for which a reflective model defines a different meta-behavior. 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.

A. Meta-Object Model (MOM)

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++ [5] and ABCL-R [6] 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.
Referent actions trapped by meta-objects are method calls. When the meta-object has completed the meta-computation, it returns the computational flow to its referent, which actually calls the method.

B. Message Reification Model (MRM)

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-behavior performed by the message; different messages may have different kinds. Every method call, is reified into an object (named message) which provides to its own management (e.g., delivery) 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 behaviors 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). The reflective tower in this model consists only of two levels: the base and the meta-level.

III. Reflection and Distribution

Distributed architectures let users of individual, networked computers share programs and data resources. Distribution can also enhance availability, 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 inter-program communication on the network,
- coordinating the execution of distributed application.

Coordination models [7] 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. Upon receiving a client request, the server performs the requested service and returns the result.
The client|server coordination model offers simplicity in closely matching data with control flow. Such a coordination model, typically, provides a high-level application programming interface (API) to handle the interactions.
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. A broker can handle client|server interactions following one among several design models, the most used models are:

- A *forwarding or routing broker* relays a client's request to the relevant server application, retrieves the answers, and relays them back to the client.
- A *handle-driven or introduction broker* returns a service handle back to the client, containing information to interact with the server for the given service (such as its name and its network address); the clients uses such information to issue its request directly to the server, which then replies back to the client.

A. *Reflective Distributed Systems*

As remarked in [8], 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 [9]). 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 [6].

IV. Channel Reification Model (CRM)

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

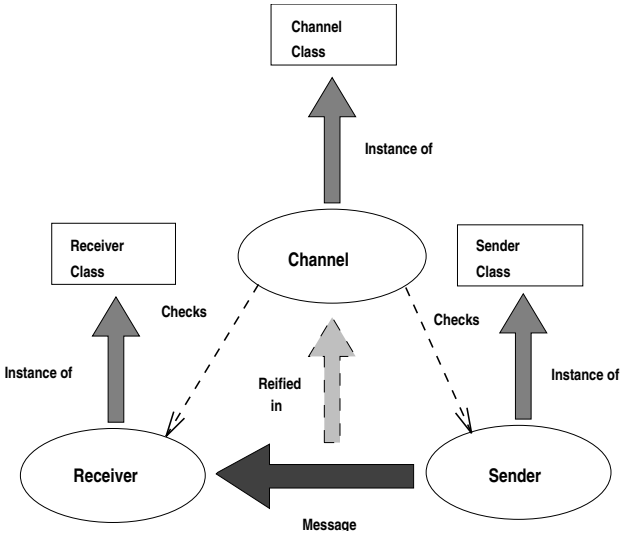

Figure 1. Channel Reification Model Scheme

(as shown in figure 1). A channel is characterized by a triple composed by 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 behavior for each method.
- ⋆ Monitored channel proliferation with pending channels elimination.
- ⋆ Possibility to keep information among meta-computations (information continuity).
- ⋆ 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 forwarding broker. Each channel kind specializes the behavior of a broker to specific requirements, and this specialization is transparent from the underlying application.

A. *CRM, MOM: How Do Their Features Compare?*

Channel Reification has been presented in the previous section as an extension to the message reification model. However channel persistence 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 behavior 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. Figure 2.a shows a service request from referent $\mathcal{A}$ to referent $\mathcal{B}$. The request is trapped by $\mathcal{A}$'s meta-object ($\mathcal{MA}$) and the reply is trapped by the server meta-object ($\mathcal{MB}$). The two meta-objects may also coordinate their meta-actions after additional communication at meta level.
Figure 2.b shows what happens in the channel reification model: the service request is reified into channel $\mathcal{C}$, so only one meta-entity is responsible for that.
In a distributed environment, the channel is a reflective abstraction of the forwarding 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, as will be shown in the examples, in correspondence with the kind of support required by useful abstractions.

V. Examples of Channel Reification Use

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;
- extendibility – we may build a reliable protocol over an unreliable one (eg., UDP), without affecting the client and server code;
- implementing binding methods (eg., those supported by **CORBA** or by **OSF DCE** 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, multi-packed message management.

We now detail two examples of use of channels: protocol substitution and reliable communication.

A. *Implementing Protocols with Channels*

Channels can be used to replace object communication protocols. For example, using channels, it is possible to transparently discriminate the service request formality. Taking advantages of channel granularity, within the same object some services can be synchronous, others asynchronous, or conditionally 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 behavior is achieved by developing several channels classes (making a channel class library), one for each service protocol (for

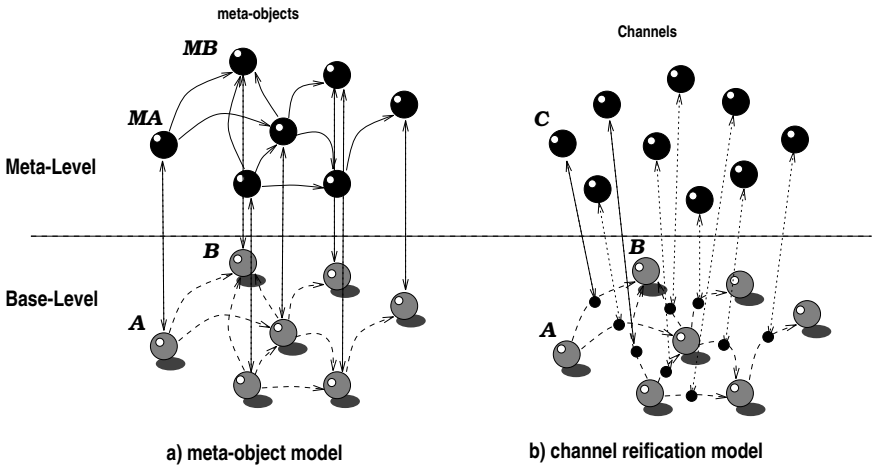

Figure 2: Meta-Object vs Channel Reification

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**, where **a** and **c** are synchronous services, **d** is asynchronous, and **b** is handled in a system-dependent default manner. Consider then the following specifications:

$$\textbf{synch\_channel} \equiv \{\textbf{a}, \textbf{c}\}$$
$$\textbf{asynch\_channel} \equiv \{\textbf{d}\}$$

they may be interpreted, at run-time, as dynamic binding requests 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.

B. Reliable Communication for Critical Objects

In [2] meta-objects are used to endow a system with fault tolerant behavior: each critical referent is replicated, and all base objects (including replicas) are equipped with meta-objects (as in Figure 3.a). Meta-objects take care of checkpointing, detecting and recovering a fault. The meta-object linked to a critical object is tightly coupled to meta-objects of the corresponding replicas, exchanging information to keep replica states consistent with that of the critical object.
Each action performed by a critical object is trapped by its meta-object which builds the checkpoint and sends it to meta-objects of replicas, which are responsible for updating their referents. This approach works well only if communication among critical objects is reliable, and fails if the underlying transmission protocol is not. In case this is a problem, a solution which uses meta-objects to implement reliable communication is of course feasible, but it would end up with rather obscure meta-object code, since each meta-object would manage intra-replicas communication as well as interaction with a replicated client or server.
Again, interactions are better monitored by channels as fault tolerant message routers established among critical objects (as in figure 3.b). Meta-objects and channels play a different role: meta-objects, as in Fabre's example, are used to keep the state of each replica coherent with the corresponding critical object, while channels guarantee a better control over service request routing.
The kind of channel used in this example is a *double-linked channel*: the double link means that it can be activated by both the client and the server, a feature that becomes useful when two critical objects interact. Such a channel traps service requests and routes them to an available server; similarly, it traps service results and returns them to the client.
A double-linked channel should implement the following actions:

- carbon-copy requests filtering,
- carbon-copy replies filtering,
- request|replies reliable delivery

All these actions are possible with the help of a *log-file*. Each request and its progress state (ie. to_send, sent, to_reply or replied tag and request|replies information) are checkpointed in a log-file stored in stable storage, identified by client and server ids and channel kind.
In case channel fault occurs, information contained in the log-file is sufficient for channel as well as communication recovery, and no channel replica is needed (the double-linked channel has no other status information). Either the client or the server may activate a double-linked channel if it does not exist or it does not answer. At startup, new double-linked channels collect interaction status information from associated log-files.
So a double-linked channel behaves like the mythical Phoenix rising from its ashes when it is needed to handle a new interaction. Thanks to its reincarnation semantic, this approach takes the *phoenix effect* name.

VI. Conclusion, Related 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.
Applications of channel reification to protocols implementation, and to fault tolerant object systems show the benefits of this new approach, which can be used to implement communication abstractions in order to extend communication 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.
Similar idea have been introduced in the **GARF** system [11] by means of the *mailer* concept, and in [12] with *connectors*; both systems do not support a concept like the *kind* which gives to channel reification an increased flexibility.
We discussed only one aspect of channels, that is their use in point-to-point communications. However the most challenging extensions are

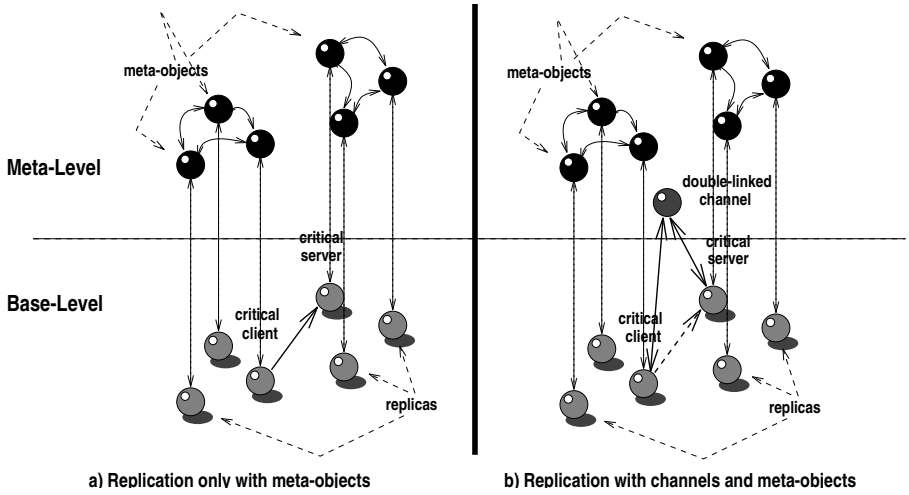

Figure 3: Reflective Fault Tolerant Systems: a Case Study

those towards multi-point communications, where significant applications are:

- multiple-RPC;
- broadcasts and object group communication;
- request serialization;
- load balancing among a group of servers;
- synchronization barriers;
- intra-replicas and inter-replicas communication.

An extension of channel reification model to multi-point communication is under development.
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 [13], where a prototype C++ implementation on top of PVM is also illustrated. Modeling and implementation 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.

VII. References

[1] 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)*. ACM, October 1989, vol. 24 of *Sigplan Notices*, pp. 317–326.

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

[3] Pattie Maes, "Concepts and Experiments in Computational Reflection", in *Proceedings of the 2nd Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'87)*, Norman K. Meyrowitz, Ed., Orlando, Florida, USA, October 1987, ACM, vol. 22 of *Sigplan Notices*, pp. 147–156.

[4] Massimo Ancona, Walter Cazzola, Gabriella Dodero, and Vittoria Gianuzzi, "Channel Reification: a Reflective Approach to Fault-Tolerant Software Development", in *OOPSLA'95 (poster section)*, Austin, Texas, USA, on 15th-19th October 1995, ACM, p. 137, Available at `http://homes.dico.unimi.it/~cazzola/references.html`.

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

[6] Hidehiko Masuhara, Satoshi Matsuoka, Takuo Watanabe, and Akinori Yonezawa, "Object-Oriented Concurrent Reflective Languages Can Be Implemented Efficiently", in *Proceedings of 7th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'92)*, Andreas Pæpcke, Ed., Vancouver, British Columbia, Canada, October 1992, ACM, vol. 27(10) of *Sigplan Notices*, pp. 127–144.

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

[8] Satoshi Matsuoka, Takuo Watanabe, and Akinori Yonezawa, "Hybrid Group Reflective Architecture for Object-Oriented Concurrent Reflective Programming", in *Proceedings of ECOOP'91*, Pierre America, Ed., Geneva, Switzerland, July 1991, Springer-Verlag, pp. 231–250.

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

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

[11] Benoît Garbinato, Rachid Guerraoui, and Karim R. Mazouni, "Distributed Programming in GARF", in *Object-Based Distributed Programming*, Rachid Guerraoui, Oscar Nierstrasz, and Michel Riveil, Eds. 1994, LNCS 901, pp. 1–32, Springer-Verlag.

[12] Stéphane Ducasse and Tamar Richner, "Executable Connectors: Towards Reusable Design Elements", in *Proceedings of ESEC'97*. 1997, LNCS 1301, pp. 483–500, Springer-Verlag.

[13] 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).