Evaluation of Object-Oriented Reflective Models

Walter Cazzola

DSI-University of Milano, Via Comelico 39-41, 20135 Milano, Italy
Phone: (+39) 10 353 6709 Fax: (+39) 10 353 6699
E-mail: cazzola@dsi.unimi.it

July 4, 1998

Abstract

In this paper we explore the object-oriented reflective world, performing an overview of the existing models and presenting a set of features suitable to evaluate the quality of each reflective model. The purpose of the paper is to determine the context applicability of each reflective model examined.

Keywords: Object-Orientation, Reflection, Reification, Reflective Model.

1 Introduction

Computational reflection is a programming paradigm suitable to develop *open systems*. An open system is a software system which can be extended in a simple manner. All the advantages of open systems manifest themselves in the software development. The system can comprise parts developed at different times by independent teams. It is possible to use specific meta-entities to test the system and then discard such meta-entities, when the test phase ends, removing the meta-level from the final system. Computational reflection improves the software reusability and stability, reducing development costs.
Nowadays, computational reflection has been used in several fields, for example for developing operating systems [43, 21], fault tolerant systems [17, 1] and compilers [27].
In the literature, several models of computational reflection (meta-class, meta-object, message reification, and channel reification) have been presented, and each model has different features absent from the others models. Thus each model should be best suitable for specific tasks.
The purpose of this paper is to analyze such models and to determine the best applicability context for each model.
In section 2 we present computational reflection (subsection 2.1) and some reflective features to evaluate the models (subsection 2.2). In section 3 we present the main reflective models and in section 4 we evaluate them using the measures described. In the conclusions we determine the applicability context of each model.

2 What one Needs to Know

In this section we present computational reflection and some reflective features (dimensions) which can be used as measures to evaluate the quality of a reflective model.

2.1 What is Reflection

Reflection appeared first in AI before propagating to various fields in computer science such as logic programming, functional programming and object-oriented programming [13].
It was introduced in object-oriented programming thanks to the famous works of Pattie Maes [31, 30].
Reflection is the ability of a system to watch its computation and possibly change the way it is performed. Observation and modification imply an "underlay" that will be observed and modified. Since the system reasons about itself, the "underlay" is itself, i.e. the system has a *self-representation* [31].
Bobrow et al. consider that observation and modification are two aspects of reflection:

"Reflection is the ability of a program to manipulate as data something representing the state of the program during its own execution. There are two aspects of such manipulation: introspection and intercession.
Introspection is the ability for a program to observe

and therefore reason about its own state. Intercession is the ability for a program to modify its own execution state or alter its own interpretation or meaning. Both aspects require a mechanism for encoding execution state as data; providing such an encoding is called reification." [3]

An object-oriented reflective system is logically structured in two or more levels, constituting a *reflective tower*. The first level is the *base-level* and describes the computations that the system is supposed to do. The second one is the *meta-level* and describes how to perform the previous computations. The entities (objects) working in the base level are called base-entities, while the entities working in the other levels (meta-levels) are called meta-entities.
Each level is causally connected to adjacent levels, i.e. entities working into a level have data structures reifying the activities and the structures of the entities working into the underlying level and their actions are reflected into such data structures. Any change to such data structures modifies entity behavior. Each level, except the first and the last one, is a base-level for the above level and is a meta-level for the underlying level. Meta-entities supervise the base-entities activity. The concept of *trap* could be used to explain how supervision takes place. Each base-entity action is trapped by a meta-entity, which performs a meta-computation, then it allows such base-entity to perform the action.
The infinite regression of the reflective tower can be managed in different ways. Brian Smith suggested the use of lazy evaluation in 3-Lisp [36]: an interpreter is not created unless needed. Others solutions are represented by the following techniques: *meta-circularity* [12] and *meta-helix* [25].
It is possible to observe, going beyond the reflective tower of compilers|interpreters, that each reflective computation can be separated into two logical aspects: computational flow context switching and meta-behavior. A computation starts with the computational flow in the base level; when the base-entity begins an action, such action is trapped by the meta-entity and the computational flow raises at meta-level (*shift-up* action). Then the meta-entity completes its meta-computation, and when it allows the base-entity to perform the action, the computational flow goes back to the base level (*shift-down* action).

2.2 How Evaluating Reflective Models

The characteristics of reflective models, that can be used to evaluate their quality and their context applicability, can be classified into three categories: *generic measures, meta-entities features* and *type of reflection*.

Some of the features examined are well-known in the reflection literature, while other features (eg. granularity, and visibility) are considered here for the first time.

Type of Reflection

The type of reflection supported by the reflective model specifies which system aspect can be monitored and changed by the meta-entities.
Types of reflection are not mutually exclusive, on the contrary a reflective model can support more than one type of reflection, but the problem is that not all object-oriented languages have the features necessary to support all types of reflection in a simple way.

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

Structural reflection allows to reify and to manipulate the computational system code. From early times, functional languages (eg. lisp) and logic languages (eg. prolog) have statements allowing to manipulate the program representation. Such statements are based on the interpretative nature of such languages and they make it easier to introduce structural reflection into it. Most object-oriented languages are compiled (eg. C++, Oberon and Eiffel) and there is no code representative at run-time. Only pure object-oriented languages, like SmallTalk, have code representative at run-time (ie. the class) and it is such representative which realizes the structural reflection, see [35, 15]. Structural reflection in non pure object-oriented languages is realized in one of the following ways, but in all cases its potentials are limited: at compile-time as in OpenC++ from version 2 [9], or introducing run-time structures representing (reifying) program code [37]. In the former method all structural reflective actions are static, in the latter the limit is represented by which aspect of the code is reified (eg. in [37] the structural reflection consists only in substitute methods code using first class procedures feature of the Oberon language).

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

Behavioral reflection manipulates the behavior of the computational system. The manipulation is realized by two phases: *method look-up or shift-up action* and *message application or shift-down action*. In the former phase, when the base-entity sends a message (calls a method) the meta-entity intercepts it and looks for the meta-computation to perform for that message. The computational flow shifts from the base-level up to the meta-level. In the latter phase, the meta-entity has the control of the computation and applies, if needed by the meta-computation, the requested message and then it returns the results to the base-entity. The computational flow shifts from the meta-level down to the base-level. Examples of behavioral reflection realization are [8, 19].

Generic Measures

The measures classified as generic derive from the reflective concepts, and — if present — both the development and the execution phase of the reflective system can benefit from them. We consider *transparency, separation of concerns, extensibility, concurrency, reflexivity* and *visibility*.

Reflexivity, under this term we mean to group three different aspects, the first two related to introspection and the last related to intercession:

1. how much time the computational flow spends in the meta-level,
2. when the computational flow shifts up or down among levels, and
3. which aspects are reified by the meta-entities.

The first aspect specifies how many times during a computation the flow shifts among levels, while the second aspect specifies on which event (eg. method calls, or variable accesses) it shifts among levels. These two aspects are important for efficiency reasons. Each level shift generates an overhead due to the context switch, for this reason, techniques to realize efficient reflective languages are based on performing meta-computation as soon as possible (for example at compile-time), more details in [10]. The last aspect specifies which parts of the system are subject to reflection.

Transparency, as stated in [31] a reflective system is logically structured into a tower of several levels and the entities of each level work independently from the work of the entities of the level above. Thus, introspection and intercession should be performed transparently; the *transparency degree* defines how transparently introspection and intercession are performed, more on transparency in [38]. The transparency degree is measured through the number of changes that must be made to the base-level code to integrate it with the meta-level.

Extensibility and **Separation of Concerns**, in reflection philosophy, each different system's functionality is the concern of a different level of the reflective tower, ie. the base-level performs its functional aspect[1] and each level extends the system, composed of the underlying tower levels, with different non-functional aspect[2] (eg. fault tolerance [17], atomicity [39], concurrence [11] and persistence [29]). Thus reflection permits to extend a computational system. Entrusting a different aspect to a different level is termed *separation of concerns*.

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

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

Concurrency, as stated in [26] concurrency and reflection are two concepts hard to integrate. Integration problems are due to the tight interconnection among base-, and meta-entities and the causal connection existing among entities working in adjacent levels. The causal connection requires to maintain a consistent meta-representation of the state and of the behavior of the underlying entities introducing all the typical problem of keeping consistence of replicated information in a distributed environment. Moreover, the tight interconnection causes a large information traffic among levels. For these reasons to support concurrency the reflective model implementations split the introspection from intercession encapsulating the intercession in the entity to reify and the introspection into a separate meta-entity. An example of a distributed object-oriented reflective language is ABCL-R [41, 42].

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

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

Meta-Entities Features

The features classified as belonging to the meta-entities are measures for properties related only to the meta-entities. We consider *lifecycle, granularity* and *proliferation* of the meta-entities.

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

Meta-Entities Lifecycle, we mean the period of the system execution in which a specific meta-entity has to exist. Granularity and lifecycle are related measures since the existence of a meta-entity depends on the existence of the base aspect which it reifies. For example, if the granularity is at method call level, the existence of a meta-entity is limited to the execution of the called method that it reifies. The lifecycle measure is important for optimization reasons, a long lifecycle increases the waste of memory and the number of active meta-entities, while a short lifecycle increases the overhead due to the meta-entities' creation and destruction.

Meta-Entities Proliferation, as meta-entities proliferation we mean the estimate of the number of meta-entities involved by the system computation. The proliferation is a quantity depending both on reflection granularity and on meta-entities lifecycle. Of course a large number of active meta-entities can degrade the system performance.

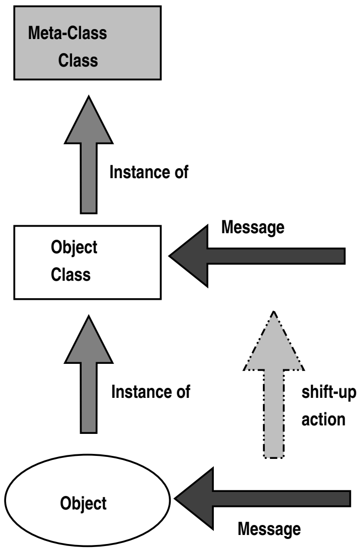


Figure 1: **Meta-Class Model Scheme**

3 Reflective Models Presentation

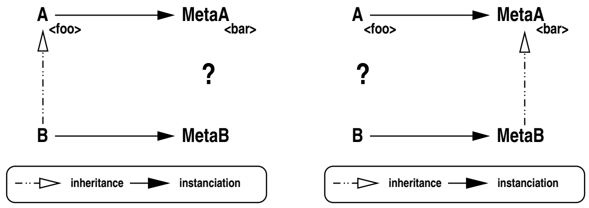
In [18], a first reflective model classification has been pointed out. Ferber remarks the existence of two major reflective approaches: *meta-object* and *communication reification*.
The meta-object approach consists in linking each base-entity — also termed *referent* — with one or more meta-entities — also termed meta-objects — reifying it. The communication approach consists in reifying only the base-entities interactions into specific meta-entities.
In this paper, for the former approach we analyze the *meta-class* and *meta-objects* models, while for the latter we analyze the *message reification* and *channel reification* models.

3.1 Meta-Class (MCM)

A class describes both the structure and the behavior of its instances. The structure is the set of instance variables whose value will be associated within the instances. The behavior is the set of methods to which instances of the class will be able to respond [23].

The meta-class model [12, 6] is a variant of the meta-object approach, in which the reflective tower is realized by the instantiation link. The meta-object reifying a base-entity is its class, the meta-meta-object reifying a meta-object is its meta-class, and so on (see figure 1).
Classes fit perfectly the role of controller and modifier of structural information, because they keep such information hardwired in their nature. Their problem is

represented by the difficulty of specializing the meta-behavior of a single instance. Any instance of a class has the same meta-object; hence all instances share the same meta-behavior. To specialize the meta-behavior for each instance, it is possible to use the inheritance relation (building up a dummy daughter class, differing from the original class only for the meta-behavior) or dictionaries keeping information and methods about/for each instance.
To dynamically change the behavior of an object it is necessary to substitute its meta-class, but not all languages allow the dynamic substitution of the class of an object; moreover changing the class of an object can lead to inconsistencies.
Usually the meta-behavior manifested by an instance is the result of meta-class composition obtained using the inheritance relation. In [23], Graube pointed out that meta-class use raises the *meta-class compatibility* problem. Bouraqadi-Saâdani and al. further explored compatibility problems in [4].



(a) Upward

(b) Downward

Figure 2: **Meta-class Compatibility Problems**

Consider the situation represented in 2(a) where a class A instance of the meta-class MetaA implements a method foo which sends the message bar to the class of the receiver of foo, and there is another class B derived by A and instance of MetaB, and the message bar is known only by MetaA, and there is no relation between MetaA and MetaB. What happens when an instance of B receives the message foo? Such a situation is termed *upward compatibility* problem. Consider the situation represented in 2(b), where the class A is an instance of the meta-class MetaA and implements the method foo, and MetaA implements the method bar which creates a new instance of the receiver and sends the foo message to it. What happens when bar is sent to B which is an instance of MetaB which inherits by MetaA? This situation is termed *downward compatibility* problem. These problems have been solved thanks to a model implemented in NeoClassTalk [5].

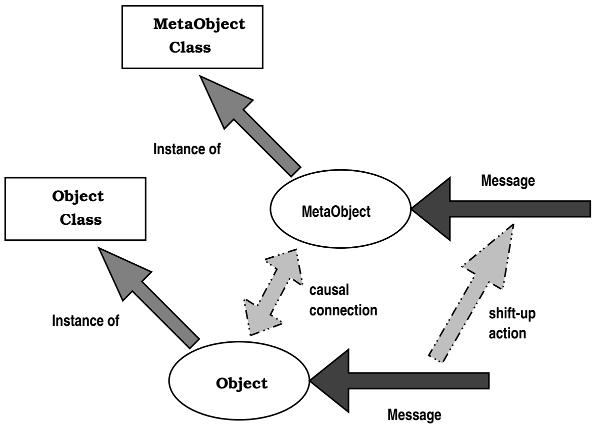


Figure 3: Meta-Object Model Scheme

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

3.2 Meta-Object (MOM)

The meta-object model [24] is a variation of the meta-object approach. But, instead of identifying the meta-object with the class of the base-entity, meta-objects are instances of a specific class *MetaObject* or of derived classes (see figure 3). The reflective tower is realized by the clientship relation. In this model, separate entities handle intercession and introspection on each base-entity. Each meta-object intercepts (shift-up action) the messages sent to its referent, and performs its meta-computation on such messages before actually delivering (shift-down action) them to its referent.
The meta-object model makes few assumptions about the relationships between base and meta-entities: in principle, each meta-object can be connected to many referents, and each referent can be linked to several meta-objects (one at a time) during its lifecycle. Usually, a meta-object is linked to an object through an instance variable, so that is in order to change the meta-object it is possible to change the value associated to that slot.
However most implementations, for efficiency reasons, restrict this freedom: in CCEL [14], OpenC++ [9], Iguana [22] and ABCL-R [32] a meta-object is linked to one referent only, and each referent can have only one meta-object during its lifecycle.
It is simple to specialize the meta-behavior per object, developing a new class of meta-objects refining the original one with the specialized meta-behavior. To specialize the meta-behavior per method or finer entities we

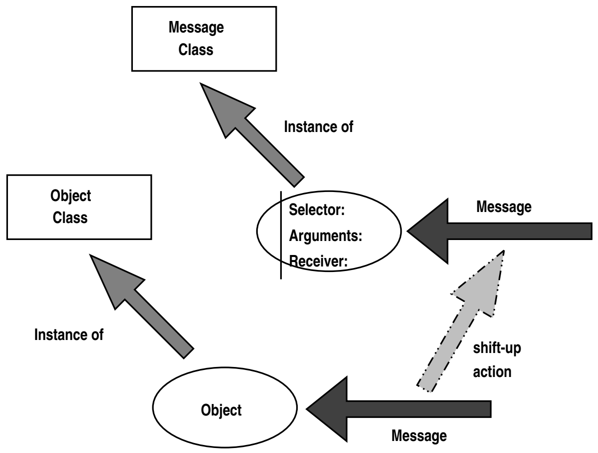


Figure 4: Message Reification Model Scheme

need to develop the meta-object in case-style checking each possibility.
The major drawback of this model is that a meta-object can monitor a message only once it has been received by the referent. Thus the meta-object loses information about the sender and cannot perform actions related to the sender's identity.
The meta-object is the most used model, and its applications are not limited to programming languages, but they also involve operating system (eg. ApertOS [43]), distributed systems (eg. CodA [33]) and graphic interfaces (eg. Silica [34]).

3.3 Message Reification (MRM)

The message reification model [18] is a variant of the communication reification approach. In this model, meta-entities are special objects, called *messages*, which reify the actions that should be performed by the base-entities. The *kind* of a message defines the meta-behavior performed by the message; different messages may have different kinds. Every method call, is reified into an object — termed message — which is charged with its own management (e.g., delivery) in according to the kind of the meta-computation required, and when the meta-computation terminates, such a message is destroyed.
It is possible to define different behaviors for method calls performed by each object, specifying a different kind for each method call. Messages are not linked to the base-entity originating them and cannot access their structural information. The message object exists only for the duration of the action which it embodies. Thus it is impossible to store information among meta-computations (lack of information continuity). On the other hand, every method call creates and then destroys an object (the message). The reflective tower in this model consists only of two levels: the base and the meta-level.

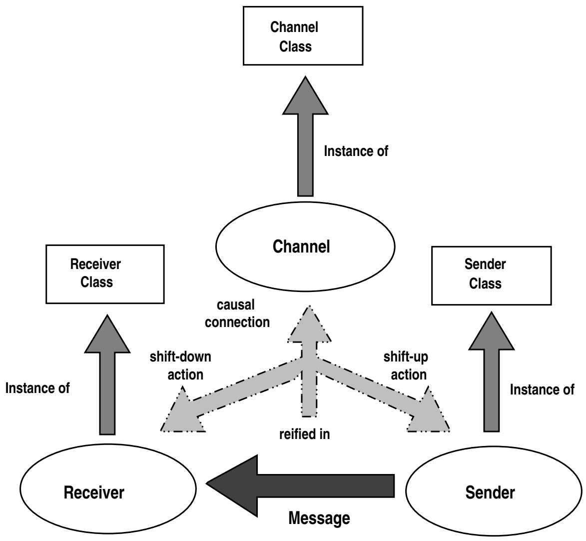


Figure 5: Channel Reification Model Scheme

3.4 Channel Reification (CRM)

The channel reification model [1, 7, 2] is an extension of the message reification model, aimed to overcome some of its limits, 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 5). 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 sender and receiver 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 receiver's answer is collected and returned to the requiring object (shift-down action).
A channel behaves like a forwarding broker. Each channel kind specializes the behavior of a broker to specific requirements, and this specialization is transparent from the underlying application.
GARF [20] and *executable connectors* [16] are examples of reflective system that can be classified as channel reification based.

4 Reflective Models Evaluation

In this section, we analyze each reflective model filtering them by measures; this examination should highlight the advantages and the drawbacks of each reflective model.

Meta-Class Model, this model is very suitable for structural reflection because the meta-entities (classes) keep all structural information of their referents (instances) hardwired into their structure. It is more difficult to change their behavior, because it involves changing the class, and this affects all their instances.
Reflexivity. All the instances of a class are either reflective base-entities (ie. the class is a meta-class) or non-reflective base-entities (ie. the class is not a meta-class). In the latter case the computational flow is always in the base-level, while in the former case it shifts up in the meta-level for each action, ie.a method call or a variable access are trapped by the meta-entity. The meta-entities reify in a simple way common information of their referents (their instances), but they have problems to reify and manipulate specific aspects of a single referent.
Transparency and Extensibility. System extensions are achieved by class composition, in despite of meta-class composability problems this mechanism guarantees a good transparency of reflection, encapsulating all reflective primitives into a separated class and hiding their use by wrapping them in filters.
Visibility. In this model, the visibility is weird because the scope of meta-entity actions are its instances and the interactions among its instances, but meta-entities have only a partial view of an interaction between one of its instances and an instance of another class. Thus in several cases the homogeneity between meta-, and base-level is lost and only in few cases it is possible to manipulate all aspects of a computation.
Concurrency. Given the nature of the model, the meta-entity is integrated with its referents. To separate them we need special statements. So concurrency, if it is needed, is only possible among base-entities or among meta-entities, but not among a meta-entity and its referents.
Meta-Entities Features. Obviously the reflection granularity is at class level. In the normal execution of the reflective system, each meta-entity exists from the beginning (each class must exist before the creation of each of its instances) to the end (a class must exist even without instances, because an instance can be created at any time during execution) of the computation. The meta-entity proliferation is very low, because there is one meta-entity for many instances.

The only major problem for the implementation of this model is represented by the necessity of a mechanism to support meta-classes at run-time. Best languages candidates are those that consider a class as an object (pure object-oriented languages) and which have class or support for them at run-time, for example SmallTalk. Other languages need to be extended with mechanisms to support classes at run-time and meta-class programming [40].

Meta-Object Model, this model is the most widespread and used one. Its advantages are the simplicity of its protocol, the adaptability of the mechanism, and the meta-objects interchangeability, interoperability and composability. Meta-objects handle behavioral

reflection, while structural one is often deferred to the base-entity classes.
Reflexivity. A base-entity may or may not have a meta-object reifying it. Encapsulated in the shift-up mechanism or it is up to the interpreter|compiler to check the existence of the related meta-entity, and the shift-up actually take place only if such meta-entity exists. This makes it possible to discriminate in space (reflective instance) and in time (removing the link with the meta-entity for a period of time) when to perform the meta-computation. Each meta-entity reifies the related base-entity and traps each message directed to it.
Transparency and Extensibility. A good degree of transparency is achieved encapsulating the shift-up and shift-down mechanisms into the interpreter|compiler. In this way the only change necessary to integrate two levels amounts to specifying who reifies who. In most reflective languages the developer specifies the meta-object's class and it is up to the compiler|interpreter to instantiate the meta-object and associate it to its referent.
Visibility. A meta-entity can take actions on messages delivered to its referent and on its referents. But, it has no control on the sender of the messages and its visibility is limited only to what concerns its referent.
Concurrency. In this model, because of causal connection, the computation of meta-objects and that of their referents are very tightly coupled. To separate them on different machines, algorithms need to be implemented to keep the referent consistent with its meta-representation. Usually, to simplify the model implementation (see [11]) a meta-object with its referent are considered as an unity of parallelism.
Meta-Entities Features. A meta-object performs introspection and intercession on all the actions of its referent; so the granularity is at the object level. Likewise the meta-object lifecycle is bound to the lifecycle of its referent. If the meta-object cannot change its referent in the worst case it is created when the referent is created and it is no longer needed when the referent is destroyed. If the meta-object can dynamically change its referent then it can be more long-lived because its lifecycle is bound to the lifecycle of several referents. The model presents an average meta-entity proliferation, if at any time, we snapshot the system computation there is only one meta-entity for each reflective entity.

Message Reification Model, in this model the meta-entities embody messages exchanged among base-entities and are not related neither to the sender nor to the receiver of the message. For this reason they are not suitable to handle structural reflection, but only behavioral reflection.
Reflexivity. Each method call provokes the control flow shift up from base- to meta-level. There is no control on which method call originates the meta-computation and which no. So a reflective system developed using this model spends much time in the meta-level and it accuses a large overhead due to the continuous context switch. Of course, meta-entities reify only the method calls.
Transparency and Extensibility. In this model, the system extensibility is limited by the limited dimension of the reflective tower. Since, the reflective tower has only two level, we must combine all non-functional aspect in only one level. To integrate the two levels we need only specify the kind of the meta-computation that each computation needs. So in this model there are more breaking points of the transparency than in the meta-object model.
Visibility. Each meta-entity has only a limited visibility. The meta-entities only reify method calls and therefore they can only act on such aspect of the system. If we don't consider the subjects of the base-computations then we can see a minimal homogeneity between base-, and meta-computation.
Concurrency. Given the nature of this model, each meta-entity is at most loosely coupled with base-, and other meta-entities. The meta-entities cannot modify the base-entities state, but only the computation of the called method. These facts guarantee the possibility to have a high degree of parallelism.
Meta-Entities Features. The model has a granularity at level of method call. Using the kind mechanism it is possible to differentiate the meta-behavior for each method call. Each method call provokes the creation, and subsequently the destruction of a meta-entity. If at any time, we snapshot the system computation there is a meta-entity for each base-computation under execution. Moreover a meta-entity exists only for the time needing to perform the base-computation it reifies.

The big advantage of this model is represented by the very fine granularity, which allows to differentiate the meta-behavior of each method call, but such an advantage is mitigated by its lack of information continuity. Because of this problem it is impossible to use this model to realize meta-behavior which need historical information, like profiling and keeping statistics.

Channel Reification Model, this model was thought to overcome the drawbacks of the message reification model. Channels embody the messages exchanged between two base-entities and they also reify the sender

and the receiver of the message. So the model is suitable to manage both behavioral and structural reflection.
Reflexivity. The model permits to shift-up in the meta-level only when this is actually needed, so it is possible to have base-entities which actions rarely activate a channel and base-entities whose every action activates a channel. So it is very hard to predict how long the computational flow stays in the meta-level. In the worst case each method call provokes the passage in the meta-level. A channel embodies a set of service requests from a client to a server and also both the client and the server.
Transparency and Extensibility. As in the message reification model, in this model to integrate two levels we need only specify the kind of the meta-computation that each computation needs. But in this model it is possible to group several method calls specifying only one meta-behavior. In this way, the model presents less breaking points of the transparency than the message reification model. Given the nature of the model complex extensions are achieved by channels composition. A good level of extensibility is guaranteed by the flexibility of the model and by the possibility of extending channels through other channels.
Visibility. Channels have a full control on each entity, and each aspect of the computation they embody. So channels enjoy of a global view of the base-computation and the meta-level reflect in a homogeneous manner the base-level.
Concurrency. This model embodies the client-server model. For this reason it is very suitable to be used in a distributed environment. A channel is loosely coupled to its referents so it can be instantiated on a different machine. Also, a channel is loosely coupled with the other channels and usually, the execution of a channel it is independent from the execution of any other channel.
Meta-Entities Features. The reflection granularity offered by this model is weird; a channel is associated to the communications exchanged between two base-entities, but not all message are trapped by the same channel, on the contrary a channel traps only those message exchanged by its referents and which in that moment require a meta-computation of its kind. Defining and changing dynamically the kind appropriately it is possible to assign a different channel for each method call. So we can state that the granularity of this model is at method call level. Channels are created in a lazy manner, ie. at the moment of their first use and are destroyed when one of their referents is destroyed. So channels lifecycle is shorter than the lifecycle of it referents. In this model there is the risk of an uncontrolled demographic explosion of the meta-entities due to a continuous request of meta-computations with new kind.

This model is very suitable for distributed systems, as seen in [2]. The possibility to establish several channels between the same pair of objects originates the problem of synchronizing the referents updates in order to avoid inconsistencies.

5 Conclusions

As shown in table 1, from our analysis it results that each considered model has its own peculiarity. These diversities make different model suitable for different tasks.

The models belonging to the communication reification approach are more suitable than the others to develop distributed reflective systems, with fine-grained parallelism and loosely coupled entities.

Moreover, the models belonging to the meta-object approach are more suitable than the others to handle structural reflection, and they permit to extend reflective systems dynamically changing its structure.

Entering in details, meta-object and channel reification are the winners of their respective categories. In respect to the other, these models are adaptable to any requirement. The others models have limitations; the meta-class model is limited by languages requirements and the message reification model is limited by the lack of information continuity.

Acknowledgments

I wish to thank M. N. Bouraqadi-Saâdani for the bootstrap discussion which originated the idea for this paper and for the suggestions about layout and contents of the paper, and Andrea Sosio for helpful comments during paper revision.

References

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

[2] Massimo Ancona, Walter Cazzola, Gabriella Dodero, and Vittoria Gianuzzi. Channel Reification: A Reflective Model for Distributed Computation. In *proceedings of IEEE International Performance Computing, and Communication Confer-*

	MCM	MOM	MRM	CRM
Behavioral Reflection	Yes	Yes	Yes	Yes
Structural Reflection	Yes	Separated	No	Separated
Reflexivity	Always or Never on message exchange Instances	Depends on message exchange Referent	Always on message exchange Message	On Request on message exchange Communication
Transparency	Very Good	Good	Poor	Average
Extensibility	Good	Good	Poor	Good
Visibility	its instances	its referent	only action	global view
Concurrency	Low	Complex	High	High
Reflection Granularity	class	object	method call	method call
Meta-Entity Lifecycle	program	referent	method	shorter than referents
Meta-Entity Proliferation	Low	Average	High	Average - High

Table 1: **Models Evaluation Summary**

ence (IPCCC'98), 98CH36191, pages 32–36, Phoenix, Arizona, USA, on 16th-18th February 1998. IEEE.

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

[4] Mohammed N. Bouraqadi-Saâdani, Thomas Ledoux, and Fred Rivard. Metaclass Composability. In *proceedings of Composability Workshop*, in 10th European Conference on Object Oriented Programming (ECOOP'96), Linz, Austria, July 1996. Springer-Verlag.

[5] Mohammed N. Bouraqadi-Saâdani, Thomas Ledoux, and Fred Rivard. Composition de Métaclasses. In *Journées francophones des langages applicatifs, JFLA'98*, Como, Italy, February 1998.

[6] Jean-Pierre Briot and Pierre Cointe. Programming with Explicit Metaclasses in SmallTalk-80. In *proceedings of OOPSLA'89*, volume 24 of *Sigplan Notices*, pages 419–431. ACM, October 1989.

[7] 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).

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

[9] Shigeru Chiba. A Meta-Object Protocol for C++. In *proceedings of OOPSLA'95*, volume 30 of *Sigplan Notices*, pages 285–299, Austin, Texas, October 1995. ACM.

[10] Shigeru Chiba. Implementation Techniques for Efficient Reflective Languages. Technical Report TR-97-06, Department of Information Science, University of Tokyo, 1997.

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

[12] Pierre Cointe. MetaClasses are first class objects: the ObjVLisp model. In *proceedings of OOPSLA'87*, volume 22 of *Sigplan Notices*, Orlando, Florida, October 1987. ACM.

[13] François-Nicola Demers and Jacques Malenfant. Reflection in Logic, Functional and Object-Oriented Programming: a Short Comparative Study. In *proceedings of the workshop section*, in IJCAI'95 (International Join Conference on AI), Montréal, Canada, August 1995.

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

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

[16] Stéphane Ducasse and Tamar Richner. Executable Connectors: Towards Reusable Design Elements. In *proceedings of ESEC'97*, LNCS 1301, pages 483–500. Springer-Verlag, 1997.

[17] 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.

[18] Jacques Ferber. Computational Reflection in Class Based Object Oriented Languages. In *proceedings of OOPSLA'89*, volume 24 of *Sigplan Notices*, pages 317–326. ACM, October 1989.

[19] Brian Foote and Ralph E. Johnson. Reflective Facilities in SmallTalk-80. In *proceedings of OOPSLA'89*, volume 24 of *Sigplan Notices*. ACM, October 1989.

[20] Benoît Garbinato, Rachid Guerraoui, and Karim R. Mazouni. Distributed Programming in GARF. In Rachid Guerraoui, Oscar Nierstrasz, and M. Riveil, editors, *Object-Based Distributed Programming*, LNCS(901). Springer-Verlag, 1994.

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

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

[23] Nicolas Graube. Metaclass Compatibility. In *proceedings of OOPSLA'89*, volume 24 of *Sigplan Notices*, pages 305–316, New Orleans, Louisiana, October 1989. ACM.

[24] Gregor Kickzales, Jim des Rivières, and Daniel G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, Cambridge, Massachusetts, 1991.

[25] Gregor Kickzales, John Lamping, and Shigeru Chiba. Avoiding Confusion in Metacircularity: The Meta-Helix. In *proceedings of the 2nd International Symposium on Object Technologies for Advanced Software (ISOTAS'96)*, LNCS 1049, pages 157–172, Kanazawa, Japan, March 1996. Springer Verlag.

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

[27] John Lamping, Gregor Kickzales, L. Rodriguez, and E. Ruf. An Architecture for an Open Compiler. In Akinori Yonezawa and Brian C. Smith, editors, *proceedings of the Int'l Workshop on Reflection and Meta-Level Architecture*, pages 95–106, 1992.

[28] Thomas Ledoux. Implementing Proxy Objects in a Reflective ORB. In *proceedings of CORBA: Implementation, Use and Evaluation Workshop*, in 11th European Conference on Object Oriented Programming (ECOOP'97), Jyvasky, Finland, June 1997. Springer Verlag.

[29] Arthur H. Lee and Joseph L. Zachary. Using Meta Programming to Add Persistence to CLOS. In *International Conference on Computer Languages*, Los Alamitos, California, 1994. IEEE.

[30] Pattie Maes. *Computational Reflection*. PhD thesis, Artificial Intelligence Laboratory, Vrije Universiteit, Brussel, Belgium, 1987.

[31] Pattie Maes. Concepts and Experiments in Computational Reflection. In *proceedings of OOPSLA'87*, volume 22 of *Sigplan Notices*, pages 147–156. ACM, October 1987.

[32] Hidehiko Masuhara, Satoshi Matsuoka, Takuo Watanabe, and Akinori Yonezawa. Object-Oriented Concurrent Reflective Languages Can Be Implemented Efficiently. In *proceedings of OOPSLA'92*, volume 22 of *Sigplan Notices*, Vancouver, Canada, October 1992. ACM.

[33] Jeff McAffer. Meta-Level Programming with CodA. In *proceedings of the 9th Conference on Object-Oriented Programming (ECOOP'95)*, LNCS 952, pages 190–214. Springer-Verlag, 1995.

[34] Ramana Rao. Implementational Reflection in Silica. In *proceedings of ECOOP'91*, pages 251–266, Switzerland, July 1991. Springer-Verlag.

[35] Frédéric Rivard. *Evolution du Comportement dans les Langages Réflexifs Dynamiquement Typés*. PhD thesis, Université de Nantes, 1997.

[36] Brian C. Smith. Reflection and Semantics in a Procedural Language. Technical Report 272, MIT Laboratory of Computer Science, 1982.

[37] Christoph Steindl. Reflection in Oberon. Technical Report CS-SSW-P96-11, Department of Computer Science (System Software), Johannes Kepler University, Linz, Austria, November 1996.

[38] Robert J. Stroud. Transparency and Reflection in Distributed Systems. *ACM Operating System Review*, 22:99–103, April 1992.

[39] Robert J. Stroud and Zhixue Wu. Using Meta-Object Protocol to Implement Atomic Data Types. In *proceedings of the 9th Conference on Object-Oriented Programming (ECOOP'95)*, LNCS 952, pages 168–189, Aarhus, Denmark, August 1995. Springer-Verlag.

[40] Don Vines and Zen Kishimoto. SmallTalk's Run-Time Type Support for C++: Emulation of SmallTalk Typing in C++. *C++ Report*, pages 44–52, January 1993.

[41] 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.

[42] Takuo Watanabe and Akinori Yonezawa. An Actor-Based Metalevel Architecture for Group-Wider Reflection. In J. W. de Bakker, W. P. de Rover, and G. Rozenberg, editors, *Fondations of Object-Oriented Languages*, pages 405–425. Springer-Verlag, 1990.

[43] Yasuhiko Yokote. The ApertOS Reflective Operating System: The Concept and Its Implementation. In *proceedings of OOPSLA'92*, volume 27 of *Sigplan Notices*, pages 414–434. ACM, October 1992.