Reflective Fault-Tolerant Systems: From Experience to Challenges

Juan Carlos Ruiz, Marc-Olivier Killijian, Jean-Charles Fabre and Pascale Thévenod-Fosse

LAAS-CNRS

7, Avenue du Colonel Roche

31077 Toulouse cedex – France

Contact: ruiz@laas.fr – Tel. +33 5 61 33 69 09 – Fax. +33 5 61 33 64 11

Abstract

This paper presents research work performed on the development and the verification of dependable reflective systems based on MetaObject Protocols (MOPs). We describe our experience, we draw the lessons learnt from both a design and a validation viewpoint, and we discuss some possible future trends on this topic.

The main originality of this work relies on the combination of both design and validation issues for the development of reflective systems, which has lead to the definition of a reflective framework for the next generation of fault-tolerant systems. This framework includes: (i) the specification of a MetaObject Protocol suited for the implementation of fault-tolerant systems, and (ii) the definition of a general test strategy to guide its verification. The proposed approach is generic and solves many issues related to the use and evolution of system platforms with dependability requirements. Two different instances of the specified MOP have been implemented in order to study the impact of different MOP implementations in the development of a reflective fault-tolerant system. Our test strategy is then illustrated on one of them. The results obtained from this work justify the interest of the proposed framework.

1. Introduction and problem statement

The development of fault-tolerant systems has to face nowadays to an increasing demand of reliability and availability, in many different types of system architectures and for various application domains. Most large fault-tolerant systems have a long lifetime and thus they are subject to a great number of evolutions of their requirements. The provision of means to handle this evolution is of great importance in the development of today's dependable systems. At the same time, the need for considering off-the-shelf executive supports (COTS operating systems, middleware and virtual machines), becoming crucial for economic reasons, involves defining solutions that are independent from the underlying layers of the system. Despite this initial economic interest, the use of COTS components may decrease the efficiency of a fault tolerance strategy, which often relies on a deep understanding and control of the structure and behavior of the executive support. In other words, more efficient fault tolerance solutions require more openness of the system underlying layers.

Our experience also shows the negative incidence that the participation of application programmers in the development of fault tolerance strategies may have on the global dependability of the system. In fact, the implementation of fault tolerance is often a complex and error prone task that requires very specific skills. Hence, fault tolerance transparency to application programmers is a desired property, rarely obtained in a dependable system. Adequate frameworks are thus necessary in order to enable the independent development of fault tolerance libraries and applications. Then, special care must be taken with the integration of these libraries and the application code. This is also a delicate issue that should be, in most cases, assisted by tools that avoid the participation of application programmers. The major problem is that existing tools (e.g., providing support for checkpointing [1]) are not well-suited for extension and customization, so their applicability remains limited in practice.

This paper handles the above issues taking into account not only the viewpoint of a system designer but also the considerations regarding the validation of the resulting solutions. More precisely, the discussion is focused on reflection, a promising emergent technology for the development of complex systems with a clear separation of concerns. From a conceptual viewpoint, *Reflection* [2] can be defined as the property by which a component enables observation and control of its own structure and behavior from outside itself. This means that a reflective component provides a *meta-model* of itself, including structural and behavioral aspects, which can be handled by an external component. This information is used as an input to perform appropriate actions for implementing non-functional properties (concerning, for instance, fault-tolerance or security strategies). The reflective systems that we consider are thus structured in two different levels of computation: the *base-level*, in charge of the execution of the application (functional) software and the *meta-level*, responsible for the implementation of observation and control (non-functional) software. The meta-level software has a runtime view (the meta-model) of the behavior and structure of its base-level. According to that view,

the meta-level can take decisions and apply corresponding actions on base-level components. The mechanisms providing such a meta-model are the *reflective mechanisms* of the system.

In systems mixing the object-oriented approach and the above reflective concepts, a so-called *MetaObject Protocol* (MOP) [3] handles the interactions between the base- and the meta-level software. This protocol governs the use of the reflective mechanisms to establish the meta-model. The efficiency of the software developed at the meta-level is highly dependent on the base-level details supplied, through the MOP, to the meta-level. Hence, the definition of this MOP is a key issue in the design and implementation of a reflective object-oriented system. For instance, the implementation at the meta-level of checkpointing and cloning strategies requires the MOP to provide deep observation and control over the internal state of a reflective component and its evolution.

Although the expected benefits of reflective technology have already been discussed in many works [4] [5] [6], as far as dependable computing is concerned, many open questions still remain from both a design and a validation viewpoint. The basic principle of reflective computing must be part of the design of software components and systems in order to provide appropriate and flexible frameworks for the development of dependable systems. Then, the resulting frameworks need the definition of systematic validation techniques in order to increase the confidence that we can place on their use. It is worth noting that a good design should not only simplify the implementation of the system but also its validation. Design for validation is a great challenge for current and future research.

From a design viewpoint, the separation of concerns promoted by the reflective approach has already shown significant effects on transparency for the application programmer, independence from the application software, reuse of core mechanisms and specialization for various contexts of use. These aspects are driving forces for reflection in many fields, not only regarding fault tolerance, but also regarding quality of service and adaptation. Today, reflection is used in a variety of software components: operating systems (e.g., Apertos [7] the basis for a Sony commercial OS called Aperios), middleware systems (e.g., open ORBs [8]) and virtual machines (e.g., the Reflective API of the JVM [9]). This notion also appears in standards for large-scale systems (e.g., emergent reflective features in CORBA [10]). These examples show that the notion has been recognized as a powerful paradigm not only in research projects but also in industrial products. It is worth noting however that reflective mechanisms are highly limited in the above examples. Extended reflective mechanisms are needed to handle fault tolerance at the meta-level of the system.

From a validation viewpoint, the use of reflection in dependable systems remains questionable. To the best of our knowledge, little work has been carried out on the verification of reflective systems, in general. This is also true in reflective object-oriented systems, in particular in MOP-based reflective architectures. We will discuss the problem here considering MOP-based dependable reflective architecture but most of these issues also apply to reflective system architectures in general. Solutions to this problem are essential for applying the reflective approach to systems with strong safety-critical constraints. Previous research work reports on the definition of formal models for verifying high-level

properties in reflective architectures [11] [12]. These models enable the reflective mechanisms provided by the MOP to be analyzed in terms of consistency, completeness, deadlocks and refinement checks. However, the level of description used in these models is too abstract to be helpful in finding problems associated with a particular MOP implementation.

Testing, which is a dynamic verification technique that consists in exercising implementations by supplying them with test case input values, is thus an essential complementary verification technique. Testing a MOP-based dependable reflective architecture is not an easy issue. The definition of a complete test strategy for such type of systems must handle the incremental verification of (1) the base-level, (2) the MOP, (3) the meta-level and finally (4) the composition of both the base- and meta-level through the MOP. As far as the MOP is concerned, the issues to be solved are numerous since, from the tester's viewpoint, a MOP gathers the well known problems related to both protocol testing (see e.g., [13]) and object-oriented testing (see e.g., [14] [15]). Four major questions must be taken into account in order to define a test strategy for a MOP: (i) what order should the reflective mechanisms of the MOP be tested in?, (ii) once a test order is defined, which test objectives should be associated with the successive testing levels defined by this order?, (iii) which conformance checks should be used in order to decide whether or not a MOP passes the test?, and (iv) given the test objectives and the conformance checks to be performed, how to design the test environments required for conducting the test experiments?

According to the above issues, we handle, all through the rest of this paper, two different perspectives for the study of a MOP-based fault-tolerant system. On one hand, we report on our experience on the design of architectures flexible enough for handling both the evolution of the system requirements and the transparency of fault tolerance for application programmers. On the other hand, we focus on the verification of reflective systems and more precisely on the definition of an incremental test strategy for the MOP of the considered reflective fault-tolerant systems.

This discussion is organized as follows. Section 2 provides the background on reflection required for understanding the rest of the paper. Section 3 describes the development of a fault-tolerant system using the reflective technology. First, we introduce the reflective framework that we have defined for the development of fault-tolerant systems. Then, we specify the requirements of a MOP suited for the implementation of such type of systems. Finally, a general model of the specified MOP is defined. Two different instantiations of this MOP model are considered in Section 4: the former provides an off-the-self MOP and the latter supplies a more elaborated CORBA-compliant MOP. Through these examples, we will study the pros and cons of each type of MOP and we will illustrate the benefits of providing different degrees of openness at the system application layer for the implementation of a fault-tolerant system. Section 5 presents our contribution to the verification of MOP-based fault-tolerant systems: the definition of a general strategy for testing the reflective mechanisms of a MOP. The usefulness of this strategy is then exemplified in Section 6 on one of the MOPs defined in Section 4 and some preliminary results are also supplied. Section 7 draws the main lessons we learnt from both

a design and a validation viewpoint. In Section 8, we then discuss the main open issues and challenges that must be addressed in future research work to make reflection a solid concept for the development of adaptable dependable software systems. Section 9 presents conclusions.

2. Basic concepts

2.1. Background on Reflection

"Computational reflection is the activity performed by a system when doing computation about (and by that possibly affecting) its own computation" [2]. Reflection enables a system to be structured in two layers: a lower layer, called *base-level*, executing the application components, and an upper layer, called *meta-level*, running the components devoted to the implementation of non-functional requirements (e.g., fault-tolerance or security).

The input for the meta-level is an image of the structural and behavioral features of the base-level. This image (called the *meta-model*) is *causally connected* to the base-level, which means that any change in one of them leads to the corresponding effect upon the other. As far as runtime reflection is concerned, the meta-model deals with two important base-level features:

- The behavior of the base-level components, which corresponds to a finite state machine indicating the possible states and transitions governing the base-level execution (notion of *execution model*);
- The structure and composition of the base-level components that depend on the base-level components considered, e.g., an object, a kernel, a compiler or a middleware (notion of *structural model*).

The above issues are addressed differently depending on the reflective component considered and the final objective of the reflective architecture. Let us take the example of the structural view defined for an application object. In Java [9], for instance, this view is a stream of information items generated and interpreted by the Java virtual machine. A more elaborated structural view of an object can be defined, as in [16], in terms of the object attributes and their types, the object methods and their respective signatures, the inheritance links, etc.

The meta-level may be considered as an interpreter of the meta-model, which can be customized according to the needs. The interest of this approach from a dependability viewpoint is that any standard action defined at the meta-level can be customized according to some non-functional objectives.

2.2. Reflective Architectures and MetaObject Protocols

From a design viewpoint (see Figure 1), one can distinguish four different processes in a reflective system to observe and control at the meta-level the features of the system's base-level. The *reification* process corresponds to the process of exhibiting to the meta-level the occurrence of base-level events. The *introspection* process provides means to the meta-level for retrieving base-level structural

information. Finally, the *intercession* process enables the meta-level to act on the base-level behavior (*behavioral intercession*) or structure (*structural intercession*). The term *behavioral reflection* will refer from here to both reification and behavioral intercession. Symmetrically, *structural reflection* will be used to designate both introspection and structural intercession mechanisms. The *reflective mechanisms* of the system are thus those mechanisms providing either behavioral or structural reflection.

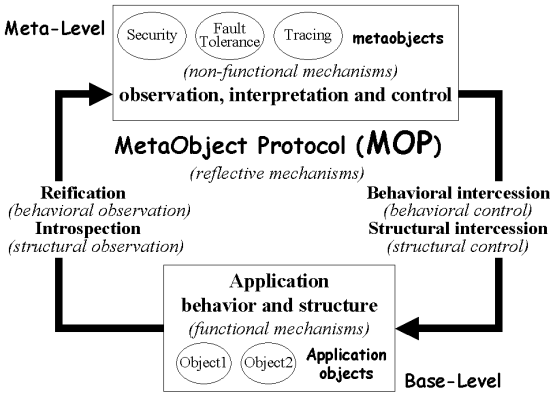


Figure 1. High-level view of a reflective architecture

In most systems mixing the object-oriented approach and the notion of reflection, a so-called MetaObject Protocol (MOP) handles the interaction between the base- and the meta-level entities, respectively called *objects* and *metaobjects*. In a sense, MOPs correspond to the rules that govern the use of the *reflective mechanisms* of the system. Hence, their definition is based on the level of observability and controllability required at the meta-level for interpreting and acting on the behavior and the structure of the base-level objects. Obviously, this identification of concerns is strongly dependent on the considered base-level component and must be handled on a case per case basis.

2.3. MOP-based Fault-Tolerant Architectures: Related Work

In early works, various MOPs have been defined and used for the implementation of fault-tolerant mechanisms at the meta-level. The MAUD [4] and GARF [5] architectures propose reflective mechanisms for intercepting base-level events at the meta-level (redirecting messages by renaming destinations in the first case, and making a tricky use of the Smalltalk exception handling mechanism in the second case). The reflective capabilities defined in these MOPs are, however, limited to behavioral reflection. This limitation avoids those systems to handle structural aspects of base-level entities that are essential, for instance, during checkpointing and cloning. A more sophisticated MOP enabling both behavioral and structural reflection was used in *FRIENDS* [6]. This MOP [17] supplies a meta-model expressed in terms of object method invocations and data containers defined for objects' states.

Despite their differences, the above MOP-based architectures have shown that meta-level strategies can be developed independently from the base-level mechanisms and selected according to the current set of system requirements. When these requirements change, the reflective mechanisms of the MOP enable non-functional mechanisms to be changed accordingly. From a non-functional viewpoint, this customization entails the specialization or the replacement of the metaobjects. The features of the MOP enable objects and metaobjects to be linked together either statically or dynamically. Despite these changes, the observation and control features provided by the MOP remain always the same. This is why the MOP is considered as the corner stone of such type of reflective architectures. In practice, the reflective mechanisms of the system must be implemented for each considered base-level software component.

As stated in the introduction, verification should complement design (and viceversa) in the development of any dependable architecture. Verification is useful for fixing problems in the design of a system. Symmetrically, designers must also keep verification in mind in order to ease the subsequent test of the resulting implementations. In our context, we think that this synergy is crucial for dependability: the reflective capabilities provided by the system must be checked in order to determine the degree of confidence that we can place in their use. Most research work focused on the verification of reflective architectures reports on the definition of high-level models for such architectures, using formal languages. In [11], the authors propose a model based on an architectural description language, called WRIGHT [18]. This language uses CSP [19] to describe a system as a set of architectural components linked by connectors (MOPs). The notion of MOP used in that work differs from the one defined in Figure 1 since it encapsulates both the reflective capabilities of the system and the system meta-level. As regards the MOP shown in Figure 1, a model based on π-calculus [20] is proposed in [12]. It describes a reflective system as composed of agents (objects and metaobjects), which communicate by exchanging messages. Both previous models can be analyzed in terms of consistency, completeness, deadlocks and refinement checks using tools specifically developed for the associated formal languages. The results obtained from these analyses are useful to increase the understanding of the MOP and ensure a certain number of high-level properties, for instance the absence of deadlocks. However, the level of description used in these models is too abstract to be helpful in finding problems associated with a particular MOP implementation. That is why testing is an essential complementary verification technique.

3. A reflective architecture for fault tolerance

This section elaborates on how a reflective architecture encompasses object-oriented techniques and reflection for the development of fault (physical faults) and intrusion (malicious interaction faults) tolerant distributed systems. This architecture is a MOP-based architecture providing metaobjects for fault tolerance, secure communications and group-based distributed applications. In practice, these metaobjects are developed using an object-oriented design method and implemented on top of basic

system services. Then, they are used recursively in order to add new properties to applications, as in MAUD [4] or in *FRIENDS* [6]. Flexibility is obtained through the provision of object-oriented libraries of metaobjects that can be easily reused and ported (subject to compiler availability) to various platforms. This reflective approach to fault tolerant system design provides new means to develop - as meta-level software - mechanisms that are traditionally implemented within the executive system layer[1].

It is worth noting that our objective here is not to describe in detail the founding principles of the architecture that has already been presented in [6]. Our aim is to study the impact of different MOP implementations in the practical provision of those principles. As a first step towards that goal, we introduce in this section the type of MOP that we consider. Section 3.1 reports on the specification of the global needs of observability and controllability required from a MOP well suited for the construction of fault-tolerant systems. A general model of the specified MOP will be then depicted in Section 3.2. Section 3.3 justifies, through a high-level view of the considered architecture, the central role played by the MOP in the provision of properties like flexibility, transparency and adaptation.

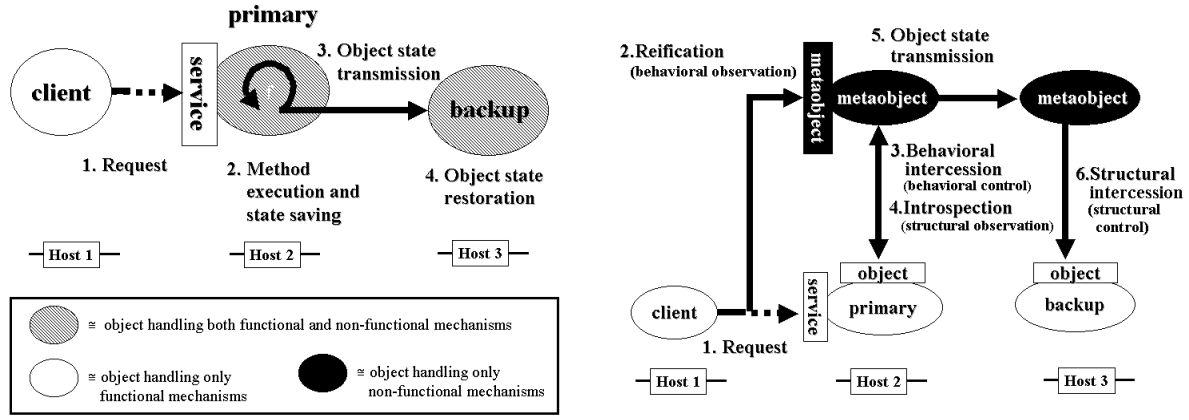
3.1. General requirements of a MOP for fault tolerance

Building a reflective fault-tolerant system requires means to control and adjust, at runtime, the behavior of the system in the presence of faults but also in normal operation. Clearly, the fault tolerance mechanisms have to be synchronized with the functional behavior of the system in order to react properly when an error is detected. The synchronization of these functional (base-level) operations with the fault tolerance (meta-level) mechanisms can take advantage of the reflective mechanisms depicted in Figure 1.

Let us consider a simple primary-backup strategy (see Figure 2) applied to a server object in order to increase its availability. In this example, the server object (the primary) has a passive replica (its backup). Only the primary computes and responds to client requests. The backup remains inactive (passive) and its state is updated periodically according to the primary's one. When the primary fails, the backup becomes primary and a new backup replica is created. This is how the strategy tolerates the faults of the primary object. Using a conventional object-oriented approach (Figure 2.a), server objects have to implement both the function of the servive they provide and the (non-funtionnal) replication strategy considered in our example. This results in a tightly coupled implementation mixing these functional and non-functional aspects of the problem. When the client sends a request to the server (step 1), the requested method is executed by the primary (step 2) that is responsible for retrieving and sending its state later to the backup object (step 3). This state is then used to restore the backup's state (step 4). This type of solution suffers from a lack of separation of concerns that leads to designs and implementations which are difficult to maintain and update. For instance, when the replication strategy

[1] The notion of executive system layer includes the middleware and the operating system.

changes, the implementation (and most of time also the design) must be revisited and changed. Using a MOP-based approach, each object uses the MOP reflective mechanisms to interact with its metaobject, which is a separate entity. This metaobject is responsible for the implementation of the associated replication strategy that can be modified without any consequence on the object implementation. Figure 2.b, provides a high level view of a MOP-based solution for our example. When a client object sends an invocation to the primary server (step 1), this invocation is intercepted (using the reification mechanisms of the MOP) at the meta-level (step 2). Then, the primary's meta-object triggers the execution of the requested method (step 3). After the method execution, the metaobject inspects the state of its primary object (step 4), which is then sent to the backup's meta-object (step 5). Finally, this state is used to restore the state of the backup (step 6).



(a) Non-Reflective version (b) Reflective version

Figure 2. Primary-Backup example

Despite its simplicity, the above example illustrates the need of behavioral observation and control for intercepting object invocations and trigger the execution of the adequate server methods. On the other hand, structural observation and control, mainly based on object serialization (as in Java [9]), are also needed on object states for both checkpointing (during normal operation) and cloning replicas (during error recovery). Consequently, both object behavioral and structural reflection is essential for implementing fault-tolerant systems.

Thus, the starting point towards the definition of a MOP well suited for fault tolerance regards the analysis of different types of fault tolerance strategies in order to provide a precise specification of the required reification, introspection and intercession features. This type of analysis has been performed in [21] on four different types of replication-based fault tolerance strategies called *active*, *semi-active* and *passive* replication and *stable-storage*. The conclusions of this analysis are summarized in Table 1 and can be synthetized in two major requirements:

1. Control of client invocations is needed for sending invocations to a group of replicas instead of a single server (active and semi-active replication).

2. At the server-side, controlling reception of the invocation enables a given strategy to be implemented between object replicas. This implies control over the execution of application methods and over the state of the application objects. These facilities are required to implement all aspects of the fault tolerance strategy (synchronization, checkpointing and cloning).

These are the minimal features needed from a reflective implementation of fault tolerance. The first bullet refers to the definition of the reification mechanisms; the last refers to the definition of the introspection and (structural and behavioral) intercession mechanisms of the MOP.

<table>
<tr><th rowspan="2">Reflective Mechanism</th><th colspan="3">Replication Strategy</th></tr>
<tr><th>Active / Semi-Active</th><th>Passive</th><th>Stable-Storage</th></tr>
<tr><td>Introspection</td><td>Required for retrieving the state of an existing replica when creating a new one</td><td>Required for maintaining passive replica’s state consistent with the state of the primary replica</td><td>Required for saving the state of the object in stable storage periodically</td></tr>
<tr><td>Reification</td><td rowspan="2">Enables forwarding client invocations to existing replicas and voting (active replication) over the received requests</td><td colspan="2" rowspan="2">Enables interception and control of the client invocations (mainly for synchronization with checkpointing and stable storage)</td></tr>
<tr><td>Behavioral Intercession</td></tr>
<tr><td>Structural Intercession</td><td>Needed for the creation of a new replica, which is made by cloning one of the existing replicas</td><td>Needed for the creation of a new passive replica by cloning the primary object</td><td>Needed for restoring the state saved on stable storage on a new object</td></tr>
</table>

Table 1. Reflective Behavioral and Structural features required for replication

3.2. A MOP model

According to the high-level view of a reflective architecture supplied in Figure 1, the base- and the meta-level layers of the architecture interact using a MOP. Figure 3 provides a general description of the type of interactions handled by the family of MOPs that we consider. Base-level objects communicate by method invocations, which are service requests sent by client objects to server objects. Each server object provides a public interface, called the *Service* interface. Every request received by the server object (step 1) is intercepted (using the reification mechanisms of the MOP) by its metaobject. This interception enables the metaobject to control the pre- and post-computation of the invoked method. The metaobject can, for instance, check specific access rights of the client and authorize or deny in consequence the execution of the concerned method. Using behavioral intercession (step 2), the metaobject acts on its object to trigger the execution of the intercepted method invocation. The considered MOP distinguishes between methods belonging to the *Service* interface of the object (public methods) and internal methods encapsulated in the object

implementation (non-public methods). The activity of these non-public methods (step 3) is transparent and thus not reified to the object's metaobject. Steps 4 and 5 show the path followed by the generated output values.

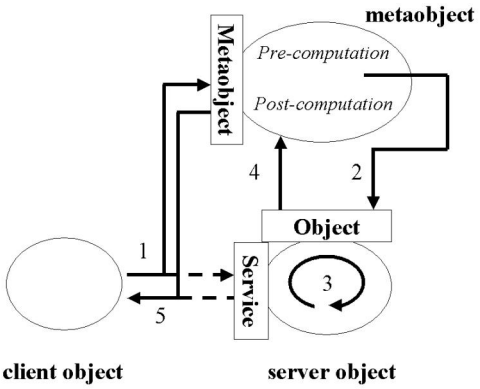


Figure 3. MOP interactions

The structural view of the base-level object handled by its metaobject includes the object identifier, its attributes and their types, and its methods and their respective signatures. Additionally, structural information regarding inheritance links and associations between classes is also included in this structural view in order to ease object state checkpointing. Metaobjects can inspect this information when necessary using the introspection mechanisms of the MOP. This enables meta-level computation to be performed in order to reason about the current configuration and state of a base-level object, which can be customized (according to the needs) using structural intercession[2].

The above interactions are defined in terms of method invocations exchanged between objects and metaobjects using two well-known interfaces, called *reflective interfaces*. The first one is the *Metaobject* interface through which metaobjects are notified about events governing the behavior of their objects. The second reflective interface is the *Object* interface, which is used by metaobjects to act on their objects. Through the *Object* interface, a metaobject can inspect the state of its object, modify that state and trigger the execution of the object's methods. It must be noted that both the introspection and the structural intercession mechanisms of the MOP are not controllable through the *Metaobject* interface. Metaobjects activate these mechanisms when necessary according to the needs associated with the non-functional requirements of the system, or upon request of other metaobjects.

It is worth noting that this model does not consider multi-threaded objects and other sources of non-determinism. This limitation concerns the reduced degree of observability and controllability over the executive support (the middleware and the operating system) available at the application layer. Handling threads at the metalevel implies the existence of reflective features providing access to the internal aspects of this support. This is also a mandatory requirement for the provision of more

2 The general approach considered to serialize objects can be found in [22][23][24].

complete descriptions of the internal structure of an object. Currently, the set of object variables handled by the operating system or the middleware, like the ones concerning the opened files or the threads running inside the object, are not considered in our model. These features constitute future challenges that require further openness of the underlying layers of the system.

3.3. The role of the MOP in the system architecture

The corner stone of the considered architecture is an instance of the general MOP presented in the previous section. The central place of such a MOP instance in the system architecture is shown in Figure 4.

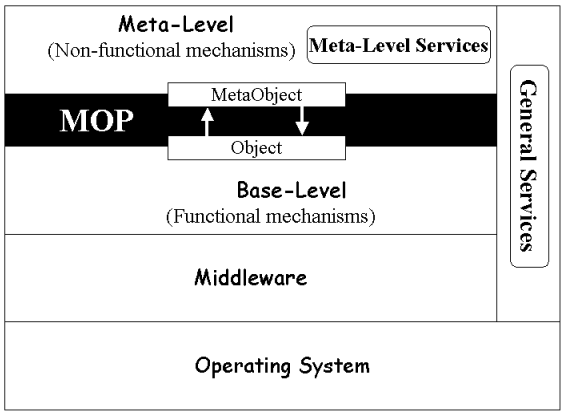


Figure 4. The MOP: corner stone of the architecture

The implementation of both the base- (functional) and the meta-level (non-functional) mechanisms is supported through the provision of two major types of basic services:

- The **meta-level services** provide the necessary support for implementing fault tolerance, secure communication, and group-based distributed mechanisms. For fault tolerance, the meta-level services must include a failure detector (based on a low-level group communication package) and a stable storage support. For secure communication, the meta-level requires an authentication server and cryptographic facilities. Finally, a group communication is also required to provide atomic multicast protocols and group membership. Other services, like a *MetaObject Factory*, are necessary for the provision and management of meta-level entities.
- The **general services** of the architecture are those shared by both the base- and the meta-level. For instance, let us quote the *Object Factories*, which are runtime services devoted to the creation and destruction of objects. It must be underlined that meatobjects are also objects and thus *MetaObject Factories* rely on Object Factories for the creation and destruction of metaobjects. The run-time language support and the libraries providing access to the executive layer (which includes the middleware and the operating system) are also considered as general services.

In this architecture we do not assume any openness of the executive layer of the system, which is considered as a COTS ("black box") component. This decision comes from the current reduced number of available "open-" executive layers suited for dependability. Ideally, the implementation of a fault tolerance strategy should have control over the concurrency model of the executive layer, e.g., scheduling policies and context switches. Similarly, the performance of state capture/restoration can be greatly improved by the knowledge of object mapping on memory unit at the meta-level. These are challenges for the next generation of dependable systems.

The role of the MOP in the architecture is thus crucial for the adequate composition of the functional and non-functional mechanisms of the system. Through the MOP, every base-level object included in an application can be linked to either a single metaobject or a stack of metaobjects. The principle is simple: each metaobject is an object and thus it can be linked through the MOP to another metaobject. This leads to the notion of stack of metaobjects, which enables the composition of several non-functional mechanisms. Figure 5 exemplifies how objects and metaobjects can be actually interconnected in order to provide fault-tolerance, security and distribution. When just distribution is required, only communication metaobjects are used at the meta-level. When fault tolerance is also required, then the base-level object is linked to a fault tolerance metaobject, the latter using group communication metaobject for the interaction between replicas. Finally, when authentification is additionally needed, interactions between fault tolerance metaobjects can be handled by security metaobjects. Such a recursive use of metaobjects leads to several meta-levels in the final application all of them linked through the same MOP, an instance of the MOP defined in Section 3.2.

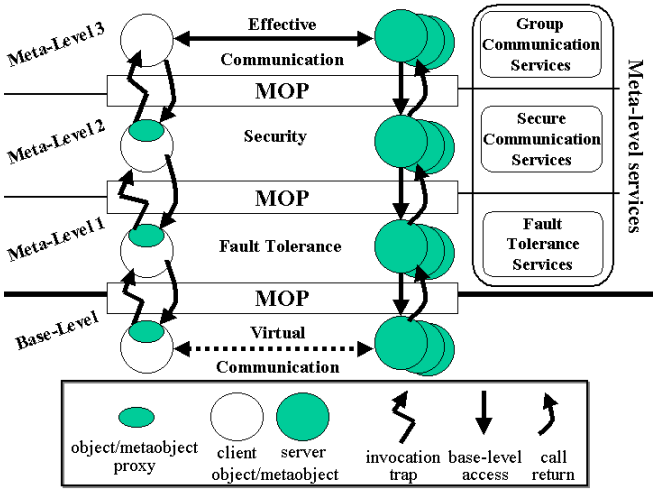


Figure 5. Composition of non-functional properties

4. Prototyping the architecture

Two prototypes of the presented architecture, respectively called *FRIENDS v1* [6] and *FRIENDS v2* [25], have already been developed. Through these prototypes, we will study the impact of different

MOP implementations in the practical provision of basic requirements for fault tolerance like flexibility, transparency and adaptation. Although based on the same model (the one defined in Section 3.2), the MOP considered in *FRIENDS v1* is different from the one implemented in *FRIENDS v2* ; an off-the-shelf MOP in the former case and a CORBA-compliant MOP in the latter. These differences have a direct incidence on the degree of details supplied by the base-level to the meta-level, and thus a direct impact on the pertinence of the non-functional strategies that may be provided by the meta-level.

4.1. *FRIENDS v1* - An off-the-shelf MOP

This first prototype was based on the use of an existing MOP written in C++, called OpenC++ v1 [17]. This runtime MOP enables C++ objets to be interconnected with metaobjects also written in C++. Both objects and metaobjects lie in the same runtime unit. During the development of the base-level classes, the application programmer statically defines this interconnection. Hence, the binding between an object and its metaobject is done on a class-by-class basis and cannot be changed at runtime.

Every metaobject inherits from (and thus specializes the implementation of) the pre-defined class *MetaObject*, whose interface is provided in Figure 6. This interface is the one known and thus used by application objects in order to reify (and thus activate) the computation of their metaobjects.

```
class MetaObject{
   public:
            void Meta_StartUp ();
            void Meta_CleanUp ();
            void Meta_MethodCall (int m_id, ArgPac args, ArgPac reply);
            void Meta_Read (int var_id, ArgPac value);
            void Meta_Assign (int var_id, ArgPac value);
   private:
            void Meta_HandleMethodCall (int m_id, ArgPac args, ArgPac reply);
            void Meta_HandleRead (int var_id, ArgPac value);
            void Meta_HandleAssign (int var_id, ArgPac value);
};
```

Figure 6. The MOP used in *FRIENDS v1*

Methods *Meta_StartUp* and *Meta_CleanUp* are called respectively after creation and before deletion of the base-level object. *Meta_StartUp* is, in our case, redefined to handle the creation of multiple replicas and their registration into a communication group. *Meta_MethodCall* intercepts a base-level method invocation: *m_id* identifies the method, *args*[3] packs its input arguments and *reply* packs the results when *Meta_MethodCall* returns. So, these three methods reify base-level invocations to the (fault tolerance) meta-level. This is how non-functional strategies are able to control the execution of base-level methods.

[3] *ArgPac* (Argument Packet) is a stack-like class that may contain any type of object (including *ArgPac* objects).

Meta_Read and *Meta_Assign* are the MOP mechanisms supplied for handling accesses to the object state. The former is called when a public attribute, identified by *var_id*, is read and *value* is supposed to contain the result of the read access. The latter is called when a public attribute, also identified by *var_id*, is modified and *value* is the new value with which that attribute should be updated. These facilities are used in our implementation to prevent external access to public attributes, i.e. to ensure a strong encapsulation.

Private methods implement the default behavior of the language: *Meta_HandleMethodCall, Meta_HandleRead, Meta_HandleAssign. Meta_HandleMethodCall* is used to activate base level methods, *Meta_HandleRead* and *Meta_HandleAssign* are respectively used to save (as shown in Figure 2.b, step 4) and restore (as shown in Figure 2.b, step 6) the state of the base level (primary and backup) objects.

This MOP enables simultaneously the following properties:

1. Ease of use and transparency of the non-functional mechanisms for the application programmer,
2. Seamless reuse and extensions of both application objects and metaobjects,
3. Composition of mechanisms.

The above properties have shown the interest of the reflective approach from both an architectural and an economic viewpoint. Regarding the performance of the solution, experiments performed and reported in [6] have shown that a certain overhead is introduced in the system due to the indirection between the base- and meta-level (200 percent for the creation and deletion of an object and between 40 percent for methods without arguments to 100 percent for methods with an average of 4 arguments). However, this overhead (associated to the MOP) is not significant with respect to the overhead introduced by distribution and fault tolerance.

The major drawbacks of the *FRIENDS v1* MOP lay in implementation weaknesses. Those identified so far essentially relate to (i) the static binding of application classes to metaobject classes, (ii) the programmer participation in the definition of this binding, (iii) the limited amount of meta-information supplied to the meta-level, and (iv) the impossibility to handle base-level inheritance.

However, these limits can be relaxed with a more elaborated MOP that provides better control over the object structure of the base-level object (inheritance links, associations and fine grain access to object state information) and enables a dynamic and transparent binding of objects and metaobjects (e.g., changing at runtime the fault tolerance mechanism being used). This is the main motivation leading to the MOP developed in the *FRIENDS v2*.

4.2. *FRIENDS v2* - A CORBA compliant MOP

The MOP of *FRIENDS v2* uses a CORBA platform [10], it solves the problems identified in the first prototype of the system and it conforms to the Fault Tolerance CORBA (FT-CORBA) specification [26]. This second prototype illustrates that reflective computing frameworks can be

compliant with the use of an off-the-shelf middleware. The use of CORBA was preferred with respect to other proprietary candidates due to the genericity, portability and interoperability properties supplied by this middleware platform.

The system model defined by *FRIENDS v2* considers both objects and metaobjects as being CORBA entities mapped to independent system processes for fault containment. These entities interact through well-known interfaces specified using the IDL language [26]. The interfaces of the MOP and the public (*Service*) interface exported by server objects are all defined using this language.

As shown in Figure 7, the execution model supplied by the MOP distinguishes three types of invocations: (i) constructor (*Start Up*) invocations, (ii) method invocations (related to the operations of the *Service* interface), and (iii) destructor (*Clean Up*) invocations. Consequently, the *MetaObject* interface contains three methods for the reification of base-level events, i.e., server object invocations. Symmetrically, the *Object* interface also provides three intercession methods that are used by metaobjects for triggering the execution of the reified invocations.

```
MethodIdentifier long;
typedef Arguments any;
typedef State any;

interface MetaObject{
        // Reification mechanisms
        void Meta_StartUp (MethodIdentifier constructorID, in Arguments args);
        void Meta_MethodCall (MethodIdentifier methodID, inout Arguments args);
        void Meta_CleanUp ( );
        // Link management mechanisms
        Object Meta_GetObject( );
        Void Meta_SetObject(Metaobject newMetaobject);
};

interface Object{
        // Introspection mechanism
        State GetFullState( );
        // Structural intercession mechanism
        void SetFullState(State new State);
        // Behavioral intercession mechanisms
        void Base_StartUp (MethodIdentifier constructorID, in Arguments args);
        void Base_MethodCall (MethodIdentifier methodID, inout Arguments args);
        void Base_CleanUp ( );
        // Link management mechanisms
        MetaObject Base_SetMetaobject(Metaobject newMetaobject);
        void Base_GetMetaobject(Metaobject newMetaobject);
};
```

Figure 7. The *FRIENDS v2* MOP

From a structural viewpoint, the MOP considers the state of an object as a list of attributes that can be saved and restored from the meta-level using respectively the *GetFullState* and *SetFullState* operations of the *Object* interface. This list contains for each attribute not only its value and its identifier, but also its type, the name of the class in which it is defined and the associations or

inheritance links that this class may have with other classes. Thus, this structural view is richer than the one supplied by the *FRIENDS v1* MOP.

The basic technology for the implementation of the above MOPs is open compilers. Open Compilers, like [27] [28], are macro systems providing means to perform source-to-source transformations. Figure 8 shows a high-level view of the open compilation process. The supplied facilities provide means to reason about and act on object-oriented programs, which are handled as a compound of classes with methods and attributes. However, these facilities do not apply by themselves any transformation to the input source code. This is the role of the *meta-program*, which uses the open compiler facilities for defining rules that (1) analyze the structure of the input source code, and (2) transform this structure according to the needs. Along this paper, these rules will be referred as *analysis and transformation rules*. It is worth noting that meta-programs may also generate error messages. When no error message is generated by the meta-program, the customized code finally produced can be compiled using a regular compiler.

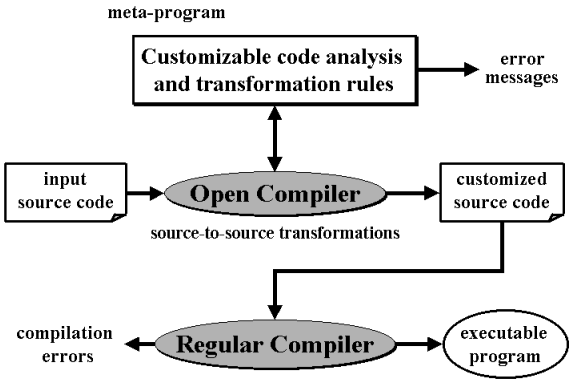


Figure 8. Open compilation process

In our prototype, the above analysis and transformation rules are specialized for the generation of reflective code from non-reflective one. The source code supplied to the open compiler is customized in order to provide an output code that (1) encapsulates the original source code, and (2) adds a specialization of the MOP reflective mechanisms for each class definition contained in the input program. The generation of an error message signals the violation of one of the programming conventions imposed by the meta-program. These programming conventions are filters applied to the input programs for checking whether or not these programs can be correctly managed by the generation rules. The benefits of such kind of approach are two-fold. On one hand, it minimizes the effort required for providing customized MOP implementations; the rules defining the MOP are defined only once and they can be later used on any program. On the other hand, the approach also provides transparency of the MOP implementation to the application programmer; this avoids the

participation of unskilled programmers in the development of such critical and error-prone reflective mechanisms.

The *FRIENDS v2* MOP has been implemented using both Java (Open Java [28]) and C++ (OpenC++ v2 [27]). In order to provide a taste of the solution, let us consider the example of a C++ class whose service consists in counting the number of requests received through its *Service* interface. The IDL definition of that service interface is shown in Figure 9.a and its C++ implementation in Figure 9.b. From that code, the meta-program implemented on top of OpenC++ v2 generates the reflective class shown in Figure 10. This class encapsulates the same capabilities provided by the original class and also supplies the reflective mechanisms of the MOP. These mechanisms are automatically generated according to the code transformation rules associated with the MOP definition. It is worth noting that although the code added to the original class might appear quite large, its size does not depend very much on the class size: for larger classes the size overhead is thus limited.

```
interface CountingInvocations{
        long count( );
};
```

(a) IDL Service interface

```
class CountingInvocations_Impl{
        private: int count;
        public:
          CountingInvocations_Impl(int i){ count = i; }
          int count( ){ ++count; }
          ~CountingInvocations_Impl{ }
};
```

(b) C++ implementation

Figure 9. Non-Reflective code

```
Class CountingInvocations_Impl{
  // ORIGINAL CODE
        private: int count;
  public:
    WrappedCountingInvocations_Impl (int i){count=i;}
    int Wrappedcount( ){ ++count; }
    WrappedDestructor( ){ }
  // REIFICATION
    CountingInvocations_Impl (int i){
      ConstructorArgs args;
      Base_SetMetaObject(MOFactory.CreateMO(object_CORBA_ref));
      args.i = i;
      Base_GetMetaObject()→Meta_StartUp(constructorID.args);
    }
    int count( ){
      CountArgs args;
      Base_GetMetaObject()→Meta_MethodCall(countID,args);
      return(args.return) ;
    }
    ~CountingInvocations_Impl ( ){
      Base_GetMetaObject()→Meta_CleanUp( );
    }
    // INTROSPECTION
    State Base_SaveState( ){
      CountingInvocationsState st;
      st.count = count;
      return st;
    }
    // STRUCTURAL INTERCESSION
    void Base_RestoreState(CountingInvocationsState st){
      count = st.count;
    }
    // BEHAVIORAL INTERCESSION
    void Base_StartUp(int methodID, any args){
      if (methodID == constructorID)
        WrappedCountingInvocations_Impl(args.i) ;
    }
    void Base_HandleCall(int methodID, any args){
      if (methodID == countID)
        args.return = Wrappedcount();
    }
    void Base_CleanUp( ){
        WrappedDestructor( ) ;
    }
    // LINK MANAGEMENT MECHANISMS
    MetaObject Base_GetMetaObject( ){
      Return mo;
    }
    void Base_SetMetaObject(MetaObject mobj){
      mobj = duplicate_reference(mobj);
    }
    private:
      MetaObject mo;
} ;
```

Figure 10. Reflective code

Beyond the properties already supplied by *FRIENDS v1*, *FRIENDS v2* provides the additional following features:

1. Fault tolerance mechanisms developed as independent CORBA objects (*reuse and adaptation*);
2. Automated customization of MOP implementations thanks to the use of Open Compiler facilities (*transparency for application programmers*);
3. An improved meta-model that enables a richer monitoring of base-level CORBA object interactions and states (*improved observability and controllability of the base-level layer*);
4. CORBA applications can dynamically select (and re-select when the system requirements change) the preferred non-functional strategy for fault tolerance *(evolution of the system)*;
5. Objects and metaobjets can be developed in different languages (currently, C++ and Java) and combined using any ORB *(interoperability, diversification and compliance with off-the-self components)*.

Performance experiments have been also performed and reported in [25]. The collected results agree with the ones obtained with the *FRIENDS v1* prototype, i.e., the runtime overhead introduced by system MOP is not significant with respect to the one introduced by distribution and fault tolerance. However, the open compiler approach used for the implementation of the MOP, imposes some compile-time overhead to the compilation of base-level code. Concretly, compilation time is (at least) increased by one order of magnitude, although the final overhead depends on the complexity of the analyses and transformation rules defining the MOP. It is worth noting that this additional overhead must be paid only at compile-time and not at runtime.

5. A Strategy for Testing MetaObject Protocols

While the design of a MOP mainly focuses on how to provide separation of concerns, the test of a MOP regards checking the conformance of the resulting MOP implementation to its specification. The test strategy presented in this section is part of a global strategy for the verification of MOP-based dependable reflective architectures that was introduced in [29]. The main concern of this global strategy is the incremental verification of:

1. The functional mechanisms defined by the system base-level;
2. The reflective mechanisms of the MOP used to compose the functional mechanisms and the non-functional mechanisms supplied by the system meta-level;
3. The non-functional mechanisms provided by the system meta-level;
4. The composition of functional and non-functional mechanisms using the reflective mechanisms of the MOP.

Phases 1, 3 and 4, which are highly dependent on the particular functional and non-functional mechanisms implemented in the target reflective architecture, are out of the scope of the work reported

here. As regards phase 2, emphasis was put on analyzing the MOP reflective mechanisms shown in Figure 3, in order to propose a test strategy generic enough to be used for different MOP-based architectures, and thus independently of a particular MOP implementation.

As a first step towards our goals, we concentrate on the following fundamental problems:

1. What order should the reflective mechanisms of the MOP be tested in? The goal is to define successive testing levels that fit with an incremental verification of the protocol mechanisms facilitating the reuse of the mechanisms that have already been tested for verifying the remaining ones.
2. Which test objectives should be associated with the successive testing levels? The objectives must focus on the verification of the properties expected from the reflective mechanism under test at each testing level.
3. Which conformance checks should be used in order to decide whether or not a MOP passes the tests, i.e., if it produces correct results in response to the test case input values?
4. Given the test objectives and the conformance checks to be performed, how to design the test environments required for conducting the test experiments? In particular, these environments must offer solutions to the observability and controllability problems generated by object encapsulation.

Other testing issues – e.g., the definition of test criteria to guide the selection of test case input values according to the test objectives – are not tackled in this section and will be commented in Section 8.

The testing strategy that we have defined identifies four different testing levels. Obviously, the strategy instantiation for the testing of a target MOP will have to comply with the MOP implementation. This concern will be exemplified on the *FRIENDS v2* MOP in Section 6.

5.1. Overview of the Strategy

According to the MOP model defined in Section 3.2, the activation of the MOP reflective mechanisms is based on the interaction channel used by objects and metaobjects. Hence, exercising (and thus testing) these mechanisms requires a high level of confidence in this interaction channel. This confidence may be obtained by testing the process followed to establish the interconnection. But the issues related to this very first testing level (called *Testing level 0*) are highly dependent on the MOP implementation and thus, they vary from one MOP to another. As a result, the strategy cannot provide general guidelines for this level although we assume that it is successfully achieved in the first place. Section 6 provides further details regarding these implementation related concerns.

Once objects and metaobjects are correctly linked, the reflective mechanisms of the MOP are exercised following a test order defined according to the dependencies existing among these mechanisms (see [6]). These dependencies are exploited in order to define an incremental test strategy

that reduces the testing effort to be spent. In accordance with this goal, we propose the following order to conduct the testing process of the four reflective mechanisms identified in Figure 1:

Testing level 1. Reification (behavioral observation) mechanisms;

Testing level 2. Behavioral intercession mechanisms;

Testing level 3. Introspection (structural observation) mechanisms;

Testing level 4. Structural intercession mechanisms.

The relevance of this order will appear all through the next sections that describe the testing levels. For each level, we give the test objectives, that is, the requirements to be met by (and thus tested on) the implementation of the target reflective mechanisms. Then, the necessary test environment is defined in terms of the entities participating in the test experiments (the server object and its metaobject, the test driver and the oracle objects), the interactions among these entities, and the conformance checks to be performed in order to decide whether or not the MOP passes the tests. The role of the *oracle object* is to verify that the test executions meet the requirements imposed by the MOP specification: it analyzes the test results according to an *oracle procedure* that implements conformance checks. The *test driver object* manages the test experiments: (i) it acts as a client object to exercise the MOP by supplying it with test case input values and, (ii) it provides the oracle object with (part of the) data that the oracle procedure uses to determine correctness during test execution.

5.2. Testing Level 1: Reification Mechanisms

As stated in Section 2.1, the base-level of the system (the server object in our case) must be *causally connected* to its meta-model, which means that any change in one of them leads to the corresponding effect upon the other. This means that the behavioral image supplied to the object's metaobject by the reification mechanisms must be complete and consistent with the real object activity. Hence, completeness and consistency are requirements to be verified.

1. *Completeness requirement* – Every message received by a server object through its *Service* interface must be reified (notified) to the server' s metaobjectAny invocation not observed at the meta-level invalidates the interpretation of the object's execution model performed by the metaobject.
2. *Consistency requirement* – Reification messages must provide the metaobject with a correct view of the events triggered at the base-level. In our case, these events are method invocations. Consequently, reification messages must identify the invoked method and give the parameter values used in the invocation. Then, (possible) output values produced in response to the activation of the invoked method have to be returned to the client object.

Figure 11 shows the test environment required for conducting test experiments. The object *Service* interface must be defined using operations with various types of method signatures in order to test the capacity of the MOP for handling a significant sample of signatures, e.g.: different directional parameter qualifiers (input, output, and input/output parameters), different parameter data types

(language built-in types, structures, arrays…), different return data types. It is also necessary to consider interfaces that inherit operations from other interfaces.

The test environment works as follows. Each time the test driver object sends a request (i.e., a test case input value) to the *Service* interface of the server object, the MOP should notify the method invocation to the server's metaobject through its *Metaobject* interface (step 1). The metaobject involved in this environment is simple: when receiving the reified message, it does not deliver it to its object; instead, a random value is generated for each output parameter of the invoked method (step 2). By this way, the metaobject simulates the execution of the reified method in order to avoid the use of the activation mechanisms that have not been tested yet. The MOP should return the output values to the test driver object (step 3). In addition, the metaobject sends the following data to the oracle object (step 4): (a) the reified method identifier, (b) the reified invocation parameters, and (c) the generated random values. Symmetrically, the test driver object communicates to the oracle object (step 5): (a) the name of the invoked method, (b) the values used in the invocation, and (c) the obtained output values. Finally, the oracle object executes the oracle procedure (step 6), which has to check the compliance between the data supplied by the metaobject and by the test driver object.

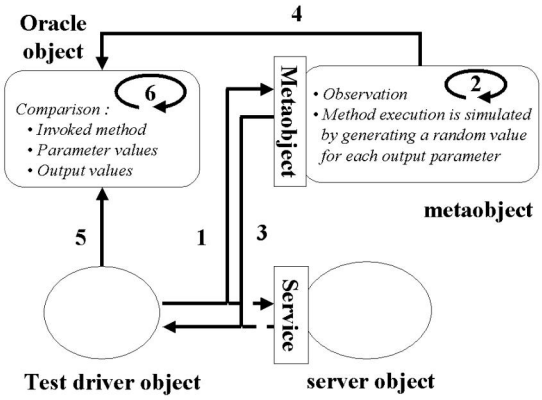


Figure 11. Testing of the reification mechanisms of the MOP

5.3. Testing Level 2: Behavioral Intercession Mechanisms

The behavioral intercession mechanisms allow a metaobject to animate the *execution model* of its base-level object, i.e, to act on its object in order to trigger the execution of an invoked method. Hence, the test objectives are to verify their capacity for (1) activating the right code of the base-level object with the correct input parameter values, and (2) returning to the metaobject the output values produced by the object's code. To decide whether or not a MOP passes the tests, the oracle procedure needs to get a report on the actual object activity in order to compare it with the image of the object activity obtained through the MOP. This report cannot be delivered through the MOP for two reasons: first, due to encapsulation, the metaobject may not know anything about the internal activity of its object, and second, even if we decide to break this encapsulation, the information would be supplied to

the metaobject through the behavioral intercession mechanisms which are under test, and thus, not yet reliable. As a result, the report must be supplied directly by the base-level object.

Figure 12 represents the test environment. The object *Service* interface must be defined using operations with various method signatures. As explained above, here the server object must report on its internal activity (notion of *verbose* object). As a result, in addition to the code related to the event treatment, every method – public, protected or private – has to generate an execution trace containing at least its name, the values of its parameters, and the output values got after completion of its execution.

Test experiments proceed as follows. The test driver object sends requests to the server object, which are reified to the metaobject (step 1). Both the format and the contents of every reified message may be considered as correct since they are built by the reification mechanisms that have already been tested according to the strategy. The metaobject involved in this environment delivers the reified information to its object (step 2). Then, the behavioral intercession mechanisms should trigger the execution of the server object code in charge of the initial request treatment (step 3), and return the output values to the metaobject (step 4). In addition, the server object produces the execution trace for the oracle object (step 5). The values returned to the metaobject are transmitted to the test driver object by way of the – presumably correct – reification mechanisms (step 6). Then, the driver object sends the following data to the oracle object (step 7): (a) the name of the invoked method, (b) the parameter values used in the invocation, and (c) the output values delivered by the metaobject. Finally, the oracle procedure checks the compliance between the data supplied by the server object and by the test driver object (step 8).

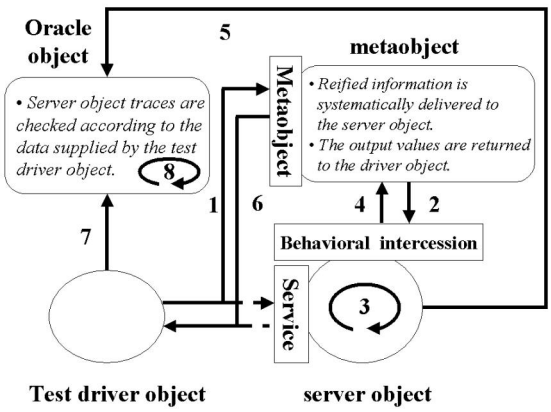


Figure 12. Testing of the behavioral intercession mechanisms of the MOP

5.4. Testing Level 3: Introspection Mechanisms

The introspection mechanisms provide means to a metaobject for retrieving structural data from its base-level object. These data are related to the object state defined by the current values of the object attributes. Depending on the non-functional mechanisms implemented at the meta-level, the

introspection mechanisms may have to supply all the attribute values, or only some of them. Testing these mechanisms without making assumption about their future use in a specific system, may be performed by considering the most general case, i.e., when all the attribute values are retrieved. The test objectives are then to verify that the full image of the object state got from the introspection mechanisms is conformed to the actual object state (same attribute values).

As mentioned in Section 3.2, the introspection mechanisms are not controllable through the *Metaobject* interface. They are triggered at the meta-level through a specific interface, called *MetaControl* (see Figure 13). The metaobject involved in the test environment must implement this second interface. In order to check the capacity of the MOP for handling a representative sample of attributes, the server object must possess attributes with various types and different levels of visibility (e.g., public, protected, private). Inheritance is another feature to be considered at this testing level.

First, the test driver object sends requests to the *Service* interface in order to initialize all the attributes of the server object to values supplied in the requests. The requests are transmitted to the server object by way of the reification and behavioral mechanisms that have already been tested (step 1), and the server object notifies the driver object of the end of the initialization process (step 2). Then, the driver object sends a request to the *MetaControl* interface for activating the introspection mechanisms (step 3). The metaobject should trigger the mechanisms (step 4) and return the observed attribute values (step 5) to the driver object. Finally (step 6), the driver object sends these values to the oracle object together with the values it supplied in the initialization requests. The oracle procedure compares both sets of values (step 7).

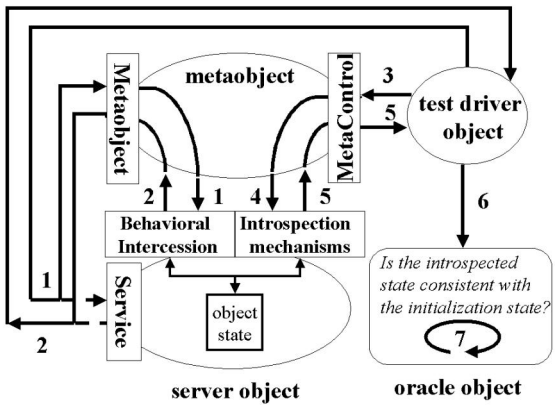


Figure 13. Testing of the introspection mechanisms of the MOP

The requests to be sent by the driver object to initialize the state of the server object (step 1) are strongly dependent on the level of encapsulation defined by the server object on its attributes. Public attributes may be directly initialized, while attributes with a more restricted visibility may cause a controllability problem. A simple solution consists in using the server object constructor, which in that

case must provide means for initializing the entire object state from the parameter values used in its activation[4].

5.5. Testing Level 4: Structural Intercession Mechanisms

Like the introspection mechanisms, the structural intercession mechanisms are controllable through the *MetaControl* interface. Their role is to allow a metaobject to update the state of its base-level object according the current needs of the non-functional requirements implemented at the meta-level: the attribute values have to be forced to the input parameter values used in the activation request sent to the *MetaControl* interface. The test objectives are then to verify that the object state is updated according to the input parameter values used in that activation.

Figure 14 shows the test environment, which makes use of all the MOP mechanisms already tested. The metaobject and the oracle object are identical to the ones shown in Figure 13, which may be reused. Here, the server object must offer appropriate method(s) for modifying the values of its attributes. Steps 1 and 2 correspond to the initialization process defined in Section 5.2. Then, the test driver object sends requests to execute method(s) of the server object that change the values of all its attributes (step 3). After notification of the end of the modification process (step 4), it sends a request to the *MetaControl* interface for activating the structural intercession mechanisms (step 5). The input parameter values supplied in the request are identical to the ones used in step 1 (initialization), and thus are different from the current state of the server object. The metaobject should trigger the mechanisms under test (step 6). Then, the test driver activates the introspection mechanisms (step 7) to get the object state after intercession (step 8). Finally (step 9), it sends these values to the oracle object together with the values it supplied in step 1. The oracle object compares both sets of values (step 10).

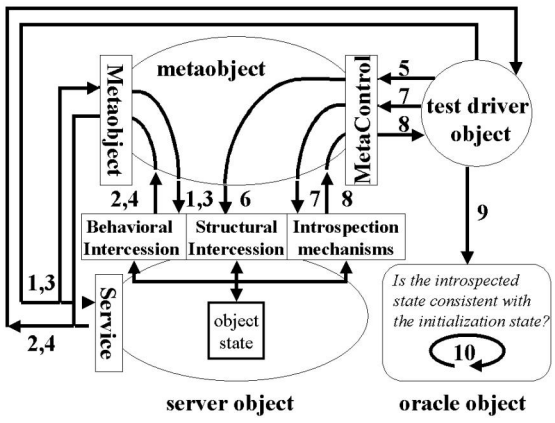


Figure 14. Testing of the structural intercession mechanisms of the MOP

[4] Another possibility consists in using specific implementation language features like the *friend functions* of C++ or the *default attributes* of Java to break the object encapsulation. Using these features, the test driver is able to control (and thus, initialize) the server object state directly, i.e., without using the MOP mechanisms in step 1. But this solution is language dependent while Figure 13 works for any MOP implementation.

6. Case Study: Testing the *FRIENDS v2* MOP

The feasibility of the strategy presented above is now showed using the *FRIENDS v2* MOP (defined in Section 4.2). Accordingly, Section 6.1 reports on the additional (and implementation dependent) testing that must precede the activation of the MOP reflective mechanisms. Then, Section 6.2 focuses on the test results obtained from the application of the test environments associated to the testing levels specified in the strategy.

6.1. Testing level 0: Testing preceding the activation of the MOP

As said in Section 5.1, the activation (and thus the test) of the reflective mechanisms is based on the interaction channel used by objects and metaobjects. Hence, the implementation of the reflective mechanisms defining the MOP can only be exercised (and thus tested) once a high confidence can be placed on the channel interconnecting objects to metaobjects.

This issue was reported in [30], where a strategy was defined for handling the implementation dependent testing concerns of MOPs generated by open compilers (like the *FRIENDS v2* one). The system model considered in that paper agrees the one defined in Section 3.2, which specifies that each server object must be handled by only one metaobject (*unity requirement*). Another requirement regards the capacity of an object to interact with its metaobject and vice versa (*interaction requirement*). From an abstract viewpoint, objects and metaobjects are able to interact if they have correctly exchanged their *references*. A reference can be defined as a compound of information unambiguously identifying a system process. For instance, a reference in *FRIENDS v2* corresponds to a set of four elements: an object identifier, the PID of the system process running the object, the IP address of the host executing that object and the host port number on which the object listens for incoming messages.

In the MOP of *FRIENDS v2*, an object is interconnected with its metaobject (and vice versa) during its instantiation. Conceptually, one may consider that objects are instantiated by runtime services called *Object factories*. Symmetrically, *Metaobject factories* are used to create metaobjects. Figure 15 depicts the test environment we have defined for this testing level. When the *Object factory* is asked by the test driver for the creation of a new base-level object (step 1), it creates an instance of the requested object class. This instantiation provokes the activation of the object constructor (step 2), whose execution leads to the creation of the object's metaobject (step 3). The object includes in that request of creation its own reference, which is delivered by the *Metaobject Factory* to the new instantiated metaobject (step 4). Once created, the metaobject stores that reference in order to get connection with its object (step 5). The *Metaobject Factory* sends to the oracle object, the reference of the metaobject and the reference of its associated object (step 6). The reference of the metaobject is also returned to the object (step 7), which stores it. This exchange of references leads both the object

and its metaobject to become interconnected. The activation of the class constructor can then be reified to the meta-level (step 8). Once the object has been instantiated, its reference is returned to the test driver object (step 9), which notifies the end of the instantiation process to the oracle object (step 10).

The oracle procedure checks the identity of the references supplied by the *Metaobject factory* (step 6) and the references finally stored by the object and its metaobject. It is worth noting that the procedure followed to obtain the second set of references may vary from one MOP to another. In general, MOPs designed for being easily tested should provide facilities for consulting that type of information. In *FRIENDS v2*, these references can be obtained, as shown in Figure 15, through the reflective interfaces of the MOP (steps 11 and 12). This is design feature of great interest for the test of this interconnection process.

The oracle procedure considers that an object and its metaobject are able to interact if they have correctly exchanged (and stored) their respective references. On the other hand, the unity requirement is respected when the notification performed in the step 6 of the described environment is only performed once during the instantiation the object. More than one notification implies the instantiation of more than one metaobject for the object, which violates the unity requirement.

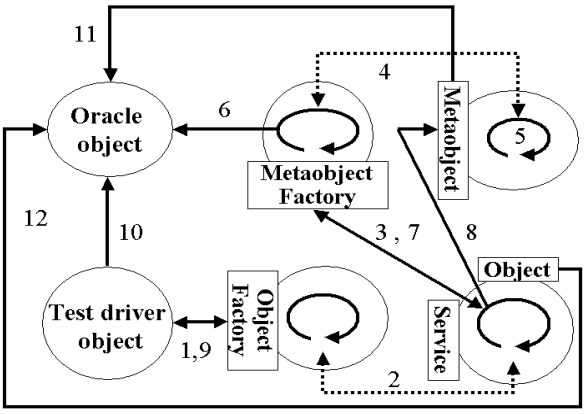


Figure 15: Testing the interconnection channel in *FRIENDS v2*

6.2. Testing experiments and results

In *FRIENDS v2*, the reflective mechanisms under test are generated by OpenC++v2 and are compiled using a regular C++ compiler (gcc version 2.95.2). The system platform (an Ultra-SPARC 5) runs on top of Solaris 7 and the associated release of Orbacus ORB, a commercial and free-source ORB providing the middleware infrastructure required for exercising the *FRIENDS v2* MOP.

Each one of the test environments included in our strategy has been implemented ontop of the above platform. We have used a probabilistic method, called statistical testing (see e.g., [31]) for generating the test case input values. However, the issue of how to select these test input values is out of the scope of our discussion. The following paragraphs report on the test experiments performed and

the type of feedback that they have provided to the system design. This discussion is conducted according to the levels identified in the test strategy.

Testing level 0: Testing preceding the activation of the MOP (Figure 15). The implementation of the test environment associated to this testing level requires a non-intrusive technique for intercepting object requests sent to the Metaobject Factory (step 3 in Figure 15). The retained solution was based on interposing *probe* objects between server objects and the Metaobject Factory. From a functional viewpoint, the probe object does not alter the exchanges between server objects and the Metaobject Factory. It delivers each trapped object request to the Metaobject Factory and then it returns the associated output. From a testing viewpoint, this probe object is responsible for transfering (step 6 in Figure 15) to the Oracle procedure the reference of each new created metaobject together with the reference of its associated object. This approach minimizes the level of intrusion introduced by the test environment.

During the test experiments, a violation of the unity requirement was detected when using inheritance for the definition of server objects. The oracle has observed that the instantiation of such servers led to the instantiation of more than one metaobject. Concretely, a different metaobject was created for each class of the inheritance hierarchy of the server. For instance, the instantiation of an inheritance hierarchy made of two classes leads to the creation of two metaobjects linked to the same object. This violates the unity requirement imposed by the MOP specification. The problem was fixed by systematically providing the type of the server object under instantiation as a parameter of each class constructor. Code must be also inserted in class constructors in order to compare that type to the one associated to each class. Thus, only the class representing the type of the object under instantiation asks the metaobject factory for the creation of the necessary metaobject. Then, a new analysis and transformation rule implementing that solution was included in the meta-program providing implementation to the considered MOP. Once this problem fixed, we have peformed testing level 1.

Testing level 1: Reification mechanisms (Figure 11). Different IDL data types were used to define the object *Service* interface: IDL built-in types, strings and CORBA object references. For instance, Figure 16 shows the interface defined for checking the capacity of the MOP for handling *long* data types in method signatures. Three different implementations were associated with each IDL interface. In the first one, a single class implements all the operations of the interface. In the other two implementations, simple and multiple inheritance are respectively used in order to reveal potential problems associated with the reification of inherited operations. Furthermore, different levels of visibility (public, protected and private) were affected to the methods included in the class implementation.

To implement the oracle procedure, one table was generated for each operation included in the *Service* interface of the server object. This table, generated using open compiler facilities, describes the signature of the operation, and gives the method identifier used by the reification mechanisms for

that operation. This information is required for checking the compliance between the data reified to the metaobject and the one handled by the test driver object.

```
interface ServiceForTestingHandlingOfType_LongParameters{
  long returnLong( );
  void InLong(in long p1);
  void OutLong(out long p1);
  void InOutLong(inout long p1);
  long AllLong(in long p1, out long p2, inout long p3)
};
```

Figure 16. IDL *Service* interface for reification and behavioral intercession

The test experiments performed revealed that different method invocations could be reified to the meta-level using the same method identifier. Hence, the metaobject cannot distinguish which method has actually been invoked. This problem has been observed using both simple and multiple inheritance. In the first case, it was easily fixed by modifying the code transformation rules that generate the MOP implementation. In practice, the algorithm used by these rules to assign method identifiers to IDL operations was slightly changed in order to avoid the reuse of method identifiers already assigned. However, in case of multiple inheritance, the problem is much more difficult to fix (and it has not been fixed yet).

The next testing levels were performed after having fixed the problem related to simple inheritance. Since multiple inheritance remains a restriction on the use of the new MOP release, it was not used in the test experiments carried out at the following testing levels.

Testing level 2: Behavioral intercession mechanisms (Figure 12). According to the observability constraints associated with the oracle procedure, the server object reports on its execution by generating execution traces. These traces are produced by every public, protected or private method included in the server object implementation. Let us remember that the activation of the code associated with a *Service* interface operation may lead to the execution of other object internal methods. As stated in Section 3.2, the object must encapsulate these internal invocations, which consequently must not be reified to its metaobject.

Two problems were revealed by the test experiments. The first one concerns the encapsulation of the internal object activity: public methods that do not belong to the *Service* interface of the server object were reified to the meta-level. This means that the behavioral intercession mechanisms may trigger reification mechanisms while they should not have to do so. The second problem was observed when the activated code performs an internal invocation to a method of a parent class, which is redefined in its child class. In that case, it is the method of the child class that is executed. This problem reveals an omission fault in the analysis and transformation rules defining the MOP. Although the MOP was supposed to handle this kind of invocation, no rules were defined for dealing with the associated transformations.

Both problems were fixed before going on with testing level 3. The first problem was fixed by restricting the application of the code transformation rules to the methods associated with the *Service* interface of the server object. Extending the existing rules in order to take account of internal invocations to methods redefined in child classes solved the second problem.

Testing level 3: Introspection mechanisms (Figure 13). The server object classes used in this level provides attributes with different data types: C++ built-in types, strings and C++ data types representing CORBA references. Figure 17 depicts the interface defined for checking the capacity of the MOP for handling *long* data type attributes. Each one of these IDL attributes maps to a public C++ accessor method that handles an associated (C++) attribute. While C++ accessors remain always public, the C++ definition of their associated attribute can be public, protected or private. Hence, these three levels of visibility have been used in order to check the ability of the MOP to handle the resulting attributes. In addition, inheritance has also been considered in the implementation of the *Service* interface. First, all attributes defined in the interface have been implemented in the same class. Then, simple class inheritance has been used for the implementation of these attributes. It is worth noting that due to dependability issues, the MOP under test restricts the set of data types that can be used for defining attributes. For instance, C++ multiple level pointers are not allowed.

The experiments performed on the considered server implementations have revealed a problem associated with the management of un-initialized string attributes. The MOP introspection mechanisms were not prepared for handling un-initialized strings so the resulting behavior was undefined: sometimes the object was freezed, sometimes a core-dump signal arised. The problem was fixed by extending the meta-program defining the MOP with a new analysis and transformation rule that forces the initialization of those string attributes that are not initialized during the execution of the object constructor. The default initialization value choosen was NULL.

```
interface ServiceForTestingHandlingOfType_LongAttributes{
  attribute long ReadWriteLong( );
  readonly attribute long ReadLong( );
};
```

Figure 17. IDL *Service* interface for introspection and structural intercession

Testing level 4: Structural intercession mechanisms (Figure 14). The server object classes used in this phase have been defined using the same features as in the previous testing level. The existence of accessor methods for each object attribute has been very useful in order to handle the initialization of the object state (see step 3 in Figure 14). No faults were revealed at this testing level during the experiments.

7. Lessons Learnt

Reflection is a concept that promotes separation of concerns in computer systems. A MetaObject Protocol is a design artifact that materializes separation of concerns in such a way that the system computation can be structured in different levels, each one devoted to the implementation of a different set of mechanisms. Various works in the field of fault tolerant computing [4] [5] [6] have shown the benefits of using MOPs at runtime for handling the behavior of base-level objects from the system meta-level. However, none of them has investigated the impact of different MOP implementations on system features like flexibility, adaptation and evolution.

As deduced from the analysis performed in Section 3.1, we can say that behavioral reflection is not sufficient for the implementation of fault tolerance strategies; structural reflection must be also supplied. The major impairment for the provision of structural reflection is the limited structural information provided by most language runtime supports. The off-the-shelf MOP presented in Section 4.1 falls into this category. Due to a limited visibility of object structural features, the *FRIENDS v1* MOP was not able to handle inheritance and imposed on programmers the static binding of objects and metaobjects. The conclusion is that in most cases, available runtime MOPs do not provide enough features to build flexible and adaptable fault-tolerant systems. This is mainly due to limitations regarding the observation and control of the structure of base-level components. So the question is, how can the above problem be handled? The solution we have considered consists in using compile-time reflective approaches based on open compilers. As explained in Section 4.2, open compilers enable the use of reflection at compile-time in order to provide customized runtime MOPs, i.e., reflection at runtime. The combined use of compile-time and runtime reflection is of high interest for the provision of Metaobject Protocols (like the *FRIENDS v2* one) with a large set of behavioral and structural reflective features. This is an important result since the degree of detail supplied by the base-level to the meta-level has a direct impact on the pertinence of the non-functional strategies that may be provided by the meta-level software.

The limits of the approach that we have identified concern the management of object features, like multi-threading, that cannot be handled because of the current lack of open executive supports. As a result, some programming conventions and restrictions have to be obeyed by base-level objects. In that context, open compilers are also attractive tools to automate the filtering and control of base-level object implementations. It is worth noting, however, that some restrictions cannot be statically verified at compile-time. In this case, the open compiler inserts assertions in order to check the restrictions at runtime. Although this approach may seem restrictive, our goal is to provide an adequate framework for the development of dependable systems, and thus components not observing the imposed restrictions cannot safely be integrated in the proposed framework. Advanced reflective features will enable these restictions to be relaxed (see Section 8).

From a testing viewpoint, the generic strategy presented in Section 5 deals with the issue of verifying the reflective mechanisms of the system. The main interest of the approach is the systematic and incremental order proposed for testing each of the reflective mechanisms supplied by a MOP. The proposed order reduces the test effort by enabling the reuse of the mechanisms already tested for testing the remaining ones, thus facilitating the implementation of the test environments. The test strategy must obviously be instantiated according to the set of reflective mechanisms supplied by a MOP (as shown in Section 6). The four testing levels defined in the strategy are required for checking the implementation of the family of MOPs specified in Section 3.2. However, other MOPs may be tested using only some of the testing levels. For instance, MOPs that only provide reification and behavioral intercession mechanisms (like those defined in MAUD and GARF) require only testing levels 1 and 2 to be instantiated; the CORBA portable interceptors [26] (a form of reification of CORBA requests) can be tested using testing level 1; the testing of the serialization features provided by a Java virtual machine [9] (a form of introspection and structural intercession mechanisms) only requires levels 3 and 4 to be instantiated.

8. Open issues and challenges

Clearly from the state of the art and our current experience, the use of MOPs in dependable systems is far from being a panacea. As motivated all through Sections 3 and 4, the design and implementation of a fault tolerant system requires a deep knowledge of information that does not belong to the application layer but to the executive support of the system. For instance, how can we ensure determinism for active replication when using multi-threading, or how can be saved and restored those site-dependent informations belonging to application objects? Solutions to these problems call for a disciplined control and access to the executive layers of the system (middleware and operating system). In other words, a radical reflective solution is thus required in order to handle the above problems. However, a reflective solution cannot be defined in general; it must be defined according to each particular aim, in our case the provision of open frameworks for the implementation of fault-tolerant systems. Accordingly, an overview of the various reflective solutions that must be investigated in the future is supplied in Figure 18. All system levels must provide adequate reflective features, but which are these features? This is a major track of our current research on the definition of architectural reflective frameworks.

As far as testing is concerned, we currently focus on the definition of rigorous test criteria to guide the automatic selection of test case input values according to the test objectives specific to each testing level of the strategy define in Section 5. This criteria definition benefits from existing approaches defined in [13] [15] [32] and considers both specification-based concerns – those that impats any MOP-based reflective architecture – and (more specific) implementation-based issues that are particular to each type of MOP. The long-term objective of this research focuses on the extension of

the proposed test strategy in order to consider the integration of base- and meta-level mechanisms using the reflective capabilities of the MOP. Then, further test work will be also required for eliminating problems associated to the recursive application of multiple metaobjects to a single object (as shown in Figure 5). Finally, the approach should be generalized for tackling the verification of other types of reflective systems not based on Metaobject Protocols.

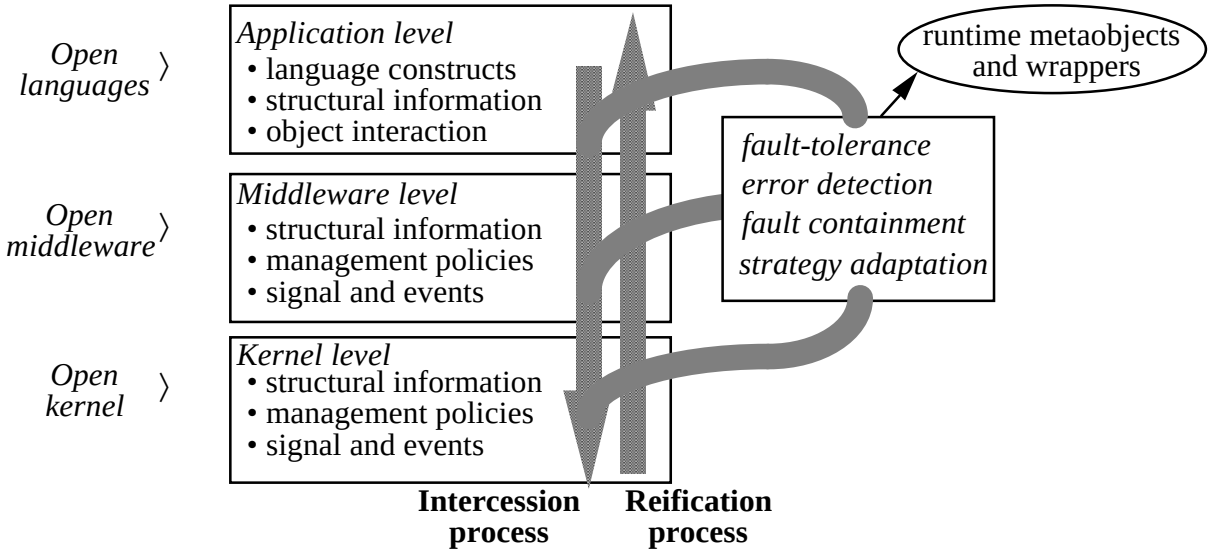


Figure 18. A radical reflective solution for dependable systems

9. Conclusions

This paper summarizes several years of research on development and verification of reflective architectures for building dependable systems. Developing a fault-tolerant system is rather expensive and its maintenance and evolution is a complex process. We have showed how reflection enhances the flexibility of such systems by providing properties like: ease of use and transparency of mechanisms for the application programmer, seamless reuse and extension of both functional and non-functional software and composition of mechanisms. Consequently, from both an architectural and an economic viewpoint, this approach constitutes a step forward in the development of fault-tolerant systems.

The work performed on reflective architectures has shown that implementing fault tolerance impacts all the layers of a system. When the underlying layers are not reflective, some aspects of the problem cannot be handled and some programming conventions and restrictions have to be obeyed. In this case, the use of open compiler facilities is both an attractive way to provide the expected properties, but also a very powerful tool to filter and control programming conventions.

Considering the underlying executive support as a “black box” is the core assumption of the work presented in this paper. Accordingly, we believe that the specification of the presented MOP still holds whatever the underlying executive support is. As we have seen, the implementation of this MOP can

be optimized and extended as the reflection paradigm progresses into language runtimes, middleware (CORBA interceptors and other serialization techniques) and operating system kernels. A radical solution based on multilevel reflection seems very attractive and is a real challenge today. We anticipate that this approach will solve many problems in building fault-tolerant systems with the current component technology.

Another originality of the proposed framework consists in the integration of validation concerns to the development of reflective fault-tolerant systems. The test strategy presented constitutes a step forward in the definition of a global strategy for the verification of reflective architectures. It focuses on the corner stone of the reflective architectures that we consider, i.e., the MOP. The main interest of the approach is the systematic and incremental order proposed for testing each one of the reflective mechanisms supplied by a MOP. This order reduces the test effort by enabling the reuse of the mechanisms already tested for testing the remaining ones, thus facilitating the implementation of the test environments. Furthermore, it allows the tester to progressively identify and fix potential problems related to the reflective mechanisms, one after the other.

The efficiency of such a progressive testing and debugging process was noticeable all along the experiments that we have conducted on real MOPs. Indeed, these experiments were performed in parallel with the development of the case study MOP presented in Section 6, i.e., with the definition of the code transformation rules used to automatically generate reflective classes from non reflective ones (see Figure 8). In that MOP, specific rules are associated with each of the four reflective mechanisms covered by the test strategy. As a result, each testing level only concentrates on a particular subset of the rules defining the MOP. Due to dependency between the reflective mechanisms, fixing potential faults in a given subset of rules before defining another subset avoids possible ripple effects of these corrections on the new defined rules. Following this approach, errors are early and incrementally detected and fixed (when possible). When a problem cannot be (easily) fixed, the test results allow at least the identification of restrictions on the use of the MOP.

Finally, we would like to point out that, beyond the various examples provided in this document, the major output of this work is the definition of a development and verification framework for a next generation of adaptable fault-tolerant systems. This framework considers the executive support as a black-box and can thus be easily ported to other implementation contexts. For instance, it can be used on top of real-time microkernels or on top of many different middleware supports (like CORBA - as shown in Section 4.2, DCOM and others). The defined framework solves many issues related to the reuse and evolution of system platforms with dependability requirements.

Acknowledgements

This work has been partially supported by the European ESPRIT project DEVA (n. 20072), by the European IST project DSoS (IST-1999-11585), by a contract with FRANCE-TELECOM (ST.CNET/DTL/ASR/97049/DT) and a grant from CNRS (National Center for Scientific Research in France) in the framework of the international agreements between the CNRS and the JSPS (Japan Society for the Promotion of Science).

10. References

1. Plank, J.S., M. Beck, and G. Kingsley, "Compiler-Assisted Memory Exclusion for Fast Checkpointing", in *IEEE Technical Committee on Operating Systems and Application Environments, Special Issue on Fault-Tolerance*. 1995.
2. Maes, P. "Concepts and Experiments in Computational Reflection" in *Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'87)*. 1987. pp. 147-155.
3. Kiczales, G., J.d. Rivières, and D.G. Bobrow, *The Art of the MetaObject Protocol*. 1992, Cambridge: The MIT Press. 335 pages, ISBN: 0-262-61074-4.
4. Agha, G., et al. "A Linguistic Framework for Dynamic Composition of Dependability Protocols" in *Dependable Computing for Critical Applications 3*. 1993. pp. 345-363.
5. Garbinato, B., R. Guerraoui, and K. Mazouni, "Implementation of the GARF Replicated Objects Platform". *Distributed Systems Engineering Journal*, 1995. 2: p. 14-27.
6. Fabre, J.C. and T. Pérennou, "A Metaobject Architecture for Fault-Tolerant Distributed Systems: the FRIENDS Approach". *IEEE Transactions on Computers, Special issue on Dependability of Computing Systems*, 1998. 47(1): p. 78-95.
7. Yokote, Y. "The Apertos Reflective Operating System: The Concept and Its Implementation" in *Object-Oriented Programming Systems, Languages and Applications (OOPSLA'92)*. 1992. pp. 414-434.
8. Costa, F.M., G.S. Blair, and G. Coulson, "Experiments with an architecture for reflective middleware", in *Integrated Computer-Aided Engineering, IOS Press*. 2000.
9. Sun, *Java Object Serialization Specification - Release 1.2*, ftp://ftp.java.sun.com/docs/j2se1.3/serial-spec.ps, 1996.
10. OMG, *CORBA 2.5 specification*, http://www.omg.org/cgi-bin/doc?formal/01-09-01, 2001.
11. Welch, I. and R. Stroud, *Adaptation of Connectors in Software Architectures*, in *ECOOP'98, Workshop on Reflective OO Programming Systems*. July 1998: Brussels, Belgium.
12. Marsden, E., J.-C. Ruiz, and J.-C. Fabre. "Towards validating reflective architectures: Formalization of a MOP" in *Workshop on Reflective Middleware*. 2000. New York. pp. 33-35.
13. Bochmann, G.V. and A. Petrenko. "Protocol testing: review of methods and relevance for software testing" in *International Symposium on Software Testing and Analysis*. 1994. Seattle (USA). pp. 109-124.
14. Barbey, S., M. Ammann, and A. Strohmeier, *Open Issues in Testing Object-Oriented Software*, in *European Conference on Software Quality (ECSQ'94)*, K. Frühauf, Editor. 1994: Basel, Switzerland. pp. 257-267.
15. Binder, R.V., *Testing Object-Oriented Systems*. 2000: Addison-Wesley. 1248 pages, ISBN: 0-201-80938-9.
16. Chiba, S., *OpenC++ 2.5 Reference Manual*, http://www.csg.is.titech.ac.jp/~chiba/opencxx/html/index.html, 1999.
17. Chiba, S. "A Metaobject Protocol for C++" in *Object-Oriented Programming Systems, Languages and Applications (OOPSLA'95)*. 1995. Austin, Texas. pp. 285-299.
18. Allen, J.R., "A Formal Approach to Software Architecture". 1997, PhD Thesis, Carnegie Mellon University.
19. Hoare, C.A.R., *Communicating Sequential Processes*. 1985: Prenctice Hall. 256 pages, ISBN: 0-13-153271-5.
20. Milner, A., ed. *A Calculus of Communicating Systems*. LNCS. Vol. 92. 1980, Springer Verlag.
21. Killijian, M.O., "Tolérance aux Fautes sur CORBA par Protocole à Métaobjets et Langages Réflexifs". 2000, LAAS/CNRS. PhD Thesis n. 00022 . 163 pages.
22. Ruiz, J.C., et al. "Optimized Object State Checkpointing using Compile-Time Reflection" in *IEEE Workshop on Embedded Fault Tolerant Systems (EFTS'98)*. 1998. Boston, USA. pp. 46-48.

23. Ruiz, J.C. and M.-O. Killijian, "Object Checkpointing: A Reflective Approach", LAAS-CNRS research report n. 99004.

24. Killijian, M.-O. and J.-C. Fabre, "A Reflective Fault-Tolerant CORBA System Based On Open Languages", LAAS/CNRS research report n. 99497.

25. Killijian, M.-O. and J.-C. Fabre. "Implementing a Reflective Fault-Tolerant CORBA System" in *19th Sysmposium on Reliable Distributed Systems*. 2000. Nürnberg (Germany). pp. 154-163.

26. OMG, *Fault-Tolerant CORBA Specification v 1.0*, http://cgi.omg.org/cgi-bin/doc?orbos/1998-05-04, 1998.

27. Chiba, S. "Macro processing in object-oriented languages" in *Technology of Object-Oriented Languages and Systems (TOOLS'98)*. November 1998. Australia. pp. 113-126.

28. Tatsubori, M., et al., *OpenJava: A Class-based Macro System for Java*, in *Reflection and Software Engineering*, Springer Verlag, Editor. 2000, LNCS 1826. pp. 119-135.

29. Ruiz, J.C., P. Thévenod-Fosse, and J.-C. Fabre. "Testing MOP-based reflective architectures" in *International Conference on Dependable Systems and Networks (DSN'2001)*. 2001. Göteborg (Sweden). pp. 327-336.

30. Ruiz, J.C., J.-C. Fabre, and P. Thévenod-Fosse. "Testing MetaObject Protocols Generated by Open-Compilers for Safety-Critical Systems" in *3rd International Conference on Metalevel Architectures and Separation of Crosscutting Concerns (Reflection'2001)*. 2001. Kyoto, Japan. pp. 134-152.

31. Thévenod-Fosse, P., H. Waeselynck, and Y. Crouzet. "Software Statistical Testing" in *Predictably Dependable Computing Systems (Randell, Kopetz, Littlewood eds.). Springer Verlag*. 1995. pp. 253-272.

32. Kung, D., P. Hsia, and J. Gao, *Testing Object-Oriented Software*. 1998. 280 pages, ISBN: 0-8186-8520-4.