Implementing a Reflective Fault-Tolerant CORBA System

Marc-Olivier Killijian and Jean Charles Fabre
LAAS-CNRS, 7, Avenue du Colonel Roche, 31077 Toulouse cedex 04, France
{killijian, fabre}@laas.fr

Abstract

The use of reflection becomes today popular for the implementation of non-functional mechanisms such as for fault-tolerance. The main benefits of reflection are separation of concerns between the application and the mechanisms and transparency from the application programmer point of view. Unfortunately, metaobject protocols available today are not satisfactory with respect to necessary features needed for implementing fault tolerance mechanisms. In previous papers, we proposed a specialised MOP, based on Corba, well adapted for such mechanisms. We deliberately focus in this paper on the implementation of this metaobject protocol using compile-time reflection and its use for implementing distributed fault tolerance. We present the design and the implementation of a fault-tolerant Corba system using this metaobject together with some preliminary experimental results. From the lessons learnt from this work, we briefly address the benefits of reflection in other layers of a system for dependability issues.

1. Introduction and Related Work

Reflection is the property of a system by which its structure and operation can be controlled and updated from outside itself [12]. This definition introduces the notion of metainformation that correspond to a self-representation of a component that is visible and manageable from outside the component. This information can be used to model the component operation at an upper layer, called *metalevel*. A reflective component also provides external means to act upon the component operation (the *metalevel* activate actions at the so-called *baselevel*, i.e. the application level). Reflection encompasses two different processes: *reification* provides the needed information to the metalevel and *intercession* provides control over baselevel actions. Defining a reflective component involves defining precisely these two processes: (i) what is the information needed to observe the internal structure and operation of the component? (ii) what are the actions that can be applied to control the component operation? Answering these questions depends very much on the application context. Clearly, these processes lead to interactions between baselevel objects and metalevel objects (i.e. metaobjects), called *MetaObject Protocol (MOP)*. The specification of a MOP may vary considerably depending on the complexity of the model that is required at the metalevel for implementing the desired mechanisms. A reflective approach offers, in principle, very interesting properties (transparency, adaptation, composition of mechanisms) for implementing non-functional mechanisms.

In many research projects advocating the use of reflection for fault-tolerance, very simple reflective mechanisms have been used, only interception mechanisms (redirection of messages by renaming destinations as in Maud [1], tricky use of exception handling of Smalltalk as in Garf [7]). The Friends [5] project was based on a metaobject protocol provided by Open C++ v1 [4]. Although this MOP offers limited reflective capabilities, it provides not only interception, but also intercession in the form of access to baselevel methods and object attributes. Friends showed that a runtime MOP is very appealing for the design and the implementation of fault tolerant systems.

The need for reflective features is currently finding its way into industrial products and standards. For instance, the reflective API provided by the Java Virtual Machine (JVM) allows introspection into the language runtime support to get access to the state of Java objects (*object serialization* [18]). While this mechanism was provided primarily for transmitting objects as remote call parameters, it can be used in fault tolerance protocols to checkpoint object replicas or to clone a new object when necessary. Limited reflective features have also been specified in CORBA in the form of interceptors [16] that provide means to control object interactions, and *object-by-value* facilities [15] that provide object serialization. The MetaObject Facility (MOF) [14] is rather different from the notion of MOP but provides an external description of the structure of CORBA objects (structural metainformation). All these features, when implemented, will be useful for the implementation of non-functional mechanisms.

Today, many papers have explained the benefits of reflection for fault-tolerant computing. It is worth noting, however, that few information is given about implementation issues. The specification and the design of a reflective Corba fault tolerant system have already been described in [10]. We focus in this paper on implementation issues of such a system, on top of Corba considered as a COTS.

The remainder of this paper is organised as follows. First, Section 2 presents an overview of the metaobject protocol we designed, reviewing the key interfaces and interactions of the various components. Section 3 describes how this metaobject protocol has been implemented using compile-time reflection. Section 4 then describes the implementation of a reflective Corba fault tolerant system based on the use of this protocol, presents some services necessary for fault tolerance, illustrates the use of the metaobject protocol on a primary-backup mechanism and gives an overview of some preliminary results. Section 5 draws the lessons learnt from this work and Section 6 concludes this paper.

2. Definition of a Metaobject Protocol for Fault Tolerance

2.1. Overview of the MOP

In object-oriented systems, method invocation corresponds to the emission and the reception of messages. In fault tolerant systems, the implementation of replication strategies rely on intercepting both the emission and the reception invocation requests between CORBA objects. The second important aspect to be controlled is the creation and destruction of replicated entities. Control over the creation of an object is needed to control the creation of multiple replicas. This aspect covers both the creation of new objects (installation of a CORBA object) but also the cloning of a new replica when one has failed (reconfiguration of the service provided by a CORBA object). Reification and intercession of object destruction is also necessary, for shutting down a given CORBA object, i.e. the various replicas are deleted by the metalevel mechanisms. The above aspects correspond to behavioural reflection.

In traditional fault-tolerant systems, including current fault tolerant object oriented systems, the state management functions are usually user-defined. This is obviously a major drawback since a wrong definition of these checkpoint functions makes the fault tolerance mechanisms ineffective. In contrast, runtime reflection can provide basic functions for state capture and restoration. In our work, compile-time reflection is used to implement object serialization in C++ [11]. This approach is similar to compiler based checkpointing as in Porch [17], but the object model considerably simplifies the analysis of application source code. Compile-time reflection is also used in companion work [9] for obtaining a complete state image of C++ objects but this work concentrates on specific language issues whereas, we focus here on the notion of a CORBA object.

Using an object-oriented model, the complete internal state of an object can be defined as the set of its attributes. We consider local checkpoints as the whole set of object attributes values. Nevertheless, in practice, it can be useful to identify the part of the object state that has been modified since the last checkpoint. Clearly, between two successive checkpoints, only a subset of the attributes is subject to change. This subset of modified attributes is called the partial internal state of the object. Structural reflection is thus interesting to automate the object state capture.

The last kind of information that must be reified is the link between objects and metaobjects. We aim here at providing a mechanism for changing this connection when necessary, for example to change the fault tolerance strategy by replacing the metaobject at runtime. The metaobject protocol must therefore provide a way of controlling the link between objects and metaobjects, i.e. the references that these entities own on each other. The MOP must enable replacing both objects (on-line updates) and metaobjects (configuration, e.g. new fault tolerance strategy) at each end of this link.

2.2. Definition and Design of the MOP

The metaobject protocol is defined as a set of interfaces belonging to the various entities involved in this protocol. On CORBA, the protocol entities (see Fig. 1) and their corresponding role are the following:

- the **client** uses the server's services through a stub that is reified;
- the **metastub** controls the behaviour and information of the stub and forwards invocations to the metaobject;
- the **metaobject** receives invocations from the metastub, controls the behaviour and monitors the state of its server;
- the **server** implements services via a standard IDL interface;

- the **metaobject factory** is responsible for the creation of the metaobjects and metastubs on demand from the stubs and the objects.

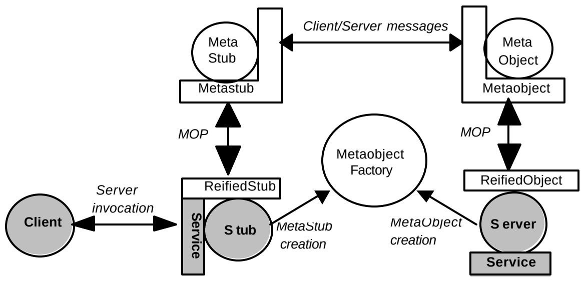

Fig. 1. Entities co-operating using the MOP

The *ReifiedObject* interface (see Fig. 2) implemented by the server object, is used by the metaobject to control the execution of baselevel methods and to obtain or restore the state of the object. Using this interface, the metaobject can activate the service methods, or the constructor/destructor of the baselevel object. This interface is composed of the following methods:

- *Base_StartUp/Base_CleanUp* is used to activate the constructor/destructor of the base object and
- *Base_MethodCall* is used to activate service methods.
- *Base_SaveState* and *Base_RestoreState* are used, respectively, to obtain or to restore the state of the base object.
- *Base_GetMetaobject* and *Base_SetMetaobject* are used to obtain and change the metaobject reference of the base object.

```
interface ReifiedObject {

 // State handling methods
 any Base_SaveState();
 void Base_RestoreState(in any newState);
 // Intercession methods
 void Base_StartUp (in long constructorID,
                   inout any arguments);
 void Base_MethodCall(in long methodID,
                     inout any arguments);
 void Base_CleanUp();
 // Link handling methods
 Metaobject Base_GetMetaobject();
 void Base_SetMetaobject
              (in Metaobject newMetaobjet);
};
```

Fig. 2. The ReifiedObject IDL interface

The Metaobject interface (see Fig. 3.), implemented by the metaobject, is used by the server to reflect actions and also to handle the object-metaobject link. This interface is composed of the following methods:

- *Meta_StartUp/Meta_CleanUp* reifies a constructor/destructor invocation,
- *Meta_MethodCall* reifies method invocation,
- *Meta_GetObject* and *Meta_SetObject*, respectively, return or update the reference of the baselevel object controlled by the metaobject.

```
interface Metaobject {

 // Reification methods
 void Meta_StartUp (in long constuctorID,
                   inout any arguments);
 void Meta_MethodCall (in long methodID,
                      inout any
arguments);
 void Meta_CleanUp();
 // Link handling methods
 Object Meta_GetObject();
 void Meta_SetObject(in Object newObject);
};
```

Fig. 3. The Metaobject IDL interface

The stub is a proxy of the server in the client's address space. The client invokes stub methods that are forwarded to the real server. Controlling invocation requests at the source is necessary in some replication mechanisms, thus the stub's behaviour must be reified. We thus introduce the notion of *ReifiedStub* interface, which is similar to the *ReifiedObject* interface. However, the state of a stub corresponds only to the reference of the server, and thus the stub can be considered as a stateless entity. The last entity in this MOP is the metastub. Its interface, *Metastub* (quite similar to *MetaObject*), defines methods for the reification of the stub behaviour and two other methods for handling the stub-metastub link.

It is worth noting that the arguments and return values of the methods as well as the state of objects are all encapsulated into the IDL type Any. The main reason for this is, of course, genericity so as to be able to apply this MOP to any CORBA object. The IDL compilers provide the marshalling and unmarshalling functions from/to real language types. This is a real benefit of the combination of the open language approach and the features provided by the underlying runtime support, in fact from the tools available with the middleware.

As illustrated in Section 4, this MOP is very suitable for the implementation of various fault tolerance strategies in a very adaptable way. However, its use is not limited to fault tolerant computing but this aspect is beyond the scope of this paper.

3. Implementation of the Metaobject Protocol

The proposed implementation of this metaobject protocol is based on the use of two open-compilers, namely OpenC++ v2 [3] and OpenJava [19]. In this section, we first give a brief overview of compile-time reflection and open compilers and, then, describe the main aspects of the implementation.

3.1. Introduction to Open Compilers

When reflection is applied to compilers, it is possible to observe and control the behaviour of the compiler, i.e. of the compilation process. This means that it is possible to customise the compile-time analysis and the code generation. Figure 4 illustrates the compilation process proposed by an open-compiler. The open compiler associates a metaclass to each input class during the compilation process. This metaclass is responsible for the customisation of the compiler, hence the name "open-compiler". A metaobject protocol thus takes place at compile-time between the open-compiler and the metaclass. The latter has the following role: (i) provide structural information about the input class C and (ii) enable modifications of the translation, thus resulting in a transformed class C'.

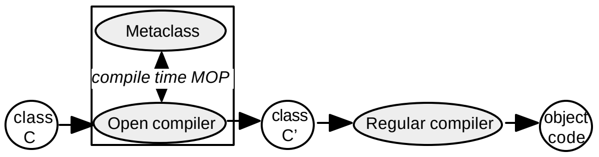

Fig. 4. Open compilation process

The role of the open compilation process is to implement the reification and intercession processes of the MOP, so that, the objects can be controlled from outside by their attached metaobjects. This customised compilation process consists in :

- renaming and wrapping methods for behavioural control,
- structural analysis and creation of new methods for object serialization (state capture), and
- enforcement of a strong encapsulation paradigm (code filtering).

3.2. Behaviour control

Reification of the object behaviour, i.e. method invocation, has been done using *method wrapping* [2]. This technique has been preferred to *class wrapping* because the latter has many drawbacks. Class wrapping (i.e. encapsulation of the original class) can be done either using inheritance or delegation; both approaches lead to serious drawbacks. The most important drawback is that the wrapper cannot access private members of the original class.

With method wrapping, each original method *foo* of the class is renamed as *real_foo* and a new method using the same signature, i.e. *foo(parameters*), which traps the invocation. This method also reifies the invocation to the metalevel by calling *Meta_MethodCall* belonging to the metaobject interface. This technique involves more modifications within the original class but the new methods are identical to the original ones from a behavioural viewpoint. This approach is also used for reifying object constructors and destructors (by calling *Meta_StartUp*, and *Meta_CleanUp*, respectively). It is worth noting that the approach for method wrapping is identical with OpenC++ or OpenJava.

The result of these transformations is that object creation, object deletion and all method invocations are reified to the metalevel. The metaobject can then activate the object methods using *Base_HandleCall*. The later unmarshalls the invocation parameters and calls the corresponding object method using its real name (e.g. *real_foo*). The control of the constructors and destructors using *Base_StartUp* and *Base_CleanUp*, respectively, is performed in the same way.

3.3. State capture

One key feature of the proposed metaobject protocol is the ability to automatically provide methods to obtain and restore the internal state of CORBA objects, i.e. the set of classes corresponding to the implementation of the application. Two checkpointing techniques are proposed: complete state or partial state techniques. It is worth noting that in Java, the complete state of objects can be obtained using the object serialization feature of the JVM. Nevertheless, *Base_SaveState* and *Base_RestoreState* methods are also generated for Java classes. Their goal is simply to obtain (or restore) the state of the object using serialization and to marshal (or unmarshal) that state information into (from) an *Any* typed variable. The proposed technique for obtaining and restoring partial state is however very useful for Java objects.

During the compilation process, the metaclass defines two new methods added to the input class: *Base_SaveState* and *Base_RestoreState*. These methods use an intermediate data structure to store the object state, and the IDL compiler generates methods for marshalling this structure into an *Any* typed variable. These structures contain a slot for each attribute and for each first level parent class in the inheritance hierarchy. The type of each slot depends on the type of the attribute: for basic types, the corresponding type is defined by the IDL-to-C++ mapping, for objects, the *Any* type is used.

The *Base_SaveState* method calls recursively each *Base_SaveState* method of any first level parent class. The resulting values are stored into the corresponding slot of the structure. When the attribute is an object, the value is obtained by calling the *Base_SaveState* method of that object. However, when objects can be accessed through several pointers, each object must be saved only once and re-created at restoration time. To tackle this problem we use a referencing technique: each saved object is assigned a reference. When the object state is saved for the first time, both the reference and the object state is saved. When the object state has already been saved, only the reference must be saved in the checkpoint. For efficiency, the information about references and pointers is saved in a hashtable indexed by pointers.

For restoration, the *Base_RestoreState* unmarshalls the received Any typed information and recursively calls each *Base_RestoreState* method of the first level parent classes. The value of each attribute is restored using the value in the corresponding slot of the received data structure. When the attribute is an object, *Base_RestoreState* is called on that object. For pointed (or referenced) objects, the checkpointed references are used to re-construct the same object hierarchy as in the original object. For restoration, the hashtable is indexed by references.

The optimisation of the state capture, i.e. obtaining a partial state for incremental checkpointing, relies on the notion of *accessors*. Accessors provide a simple means to detect which attribute has been modified since the last checkpoint. They maintain a flag for each attribute, indicating whether this attribute has been modified or not. When an attribute is checkpointed, its flag is reset. A new data structure is defined for holding partial states and handling them dynamically. In practice, it is a list of attribute slots holding the attribute identifier and its corresponding value. Concerning arrays, accessors maintain a list of modified elements instead of a simple flag (i.e. a list of indexes). Only array elements that have been modified are saved. The algorithms for saving (restoring) the partial states are similar to those for the complete state, except that only those attributes whose flag is set are saved (restored).

It is worth noting that accessors are defined using the same visibility as the original attribute (public, protected and private, Java defines also default). In the source code of the class, attribute access is replaced by accessors invocations. This is also the case for attributes used as method invocation parameters. Thanks to this simple notion, every type of attribute access can successfully be controlled. Three different types of accessors are provided:

- a read accessor that returns the current value of the attribute,
- a postfixed write accessor that updates the value and returns the previous value,
- a prefixed write accessor that updates the attribute and returns the new value.

3.4. Code Filtering

During the analysis of the source code of the input class, the metaclass can return errors to the user. These errors can either correspond to a wrong usage of the standard programming language, or new errors identified by the metaclass. These new errors are related to the analysis and to the properties the metaclass wants to enforce in the input class. As far as dependability is concerned, this technique enables some additional verification to be performed such as enforcing programming conventions.

Some object-oriented programming languages are designed as a compromise between a strong object model and programming flexibility, these are hybrid languages. The C++ language, for instance, provides a weak object-model and a very lazy access to attributes. Violating encapsulation in C++ is not very difficult. Java, in contrast, proposes a stronger object-model. However, neither of these languages ensures a strong encapsulation of object attributes. Enforcing encapsulation has many benefits in our context: the state of an object must only be modified in a controlled manner, i.e. though accessors. This is a very important issue, especially to tackle the problem of identifying the partial state of an object.

For the above reasons, in C++, the use of the following features cannot be accepted:

- Direct access to attributes using public or protected attributes, or with friend classes or functions.
- Use of global or class variables because such variables are shared among several C++ objects; it is not possible to correctly determine the state of objects that use such variables.
- Pointer arithmetic that enables uncontrolled access to any memory location is very error-prone.
- Pointers as method parameters because the entity which is given the pointer can save and reuse it later; this feature also breaks the encapsulation principle.

Since Java provides blind references instead of pointers, only direct access to attributes and class variables must be excluded from Java applications.

Compile-time reflection is used here for code filtering, in order to forbid the use of these features. Global and class variables, pointer arithmetic and pointers as parameters are simply forbidden by the metaclass. This illustrates a first benefit of compile-time reflection: enforcing programming conventions that are mandatory for the implementation of fault tolerance strategies. When such features are used in application classes, the compilation process stops and throws an error to the user. Control over direct access to attributes is done by accessors.

4. Implementation of a Reflective Fault Tolerant System

The objective of this section is to illustrate the use of this metaobject protocol in the design and the implementation of a fault tolerant CORBA system. The MOP is the corner stone of the architecture in which fault-tolerance mechanisms are developed as metaobjects.

4.1. Host Architecture

The basic elements of this architecture are thus the following: an operating system, an ORB, a group communication system and the runtime support for programming languages. In this architecture, the CORBA middleware, the underlying operating system and group communication system can be changed easily, without requiring any modification of either in the application or in the mechanisms. For this purpose, on top of this platform various services are provided: object and metaobject factories and a group communication service. These services are used by the fault tolerance metaobjects. The MOP links the application and the metalevel objects. In this section, we describe these various layers (Fig. 8).

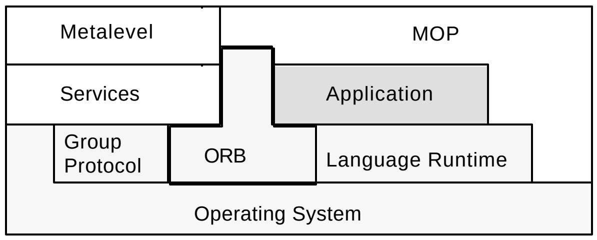

Fig. 8: Block overview of the Architecture

Object Factory: This service is responsible for the creation of object replicas on any node of the system, either during the initialisation or during recovery. The CORBA objects are created as new Unix processes, firstly for fault-containment between the objects themselves and secondly, for separation of concerns between the factory and the created objects. The links between the objects classes and the executable files to be launched by the factory are expressed into a configuration file.

Metaobject Factory: This service is responsible for the management of metaobjects and the dynamic handling of the link between objects and metaobjects. The metaobject factory manages the whole runtime life-cycle of metaobjects: creation, replacement and destruction. The metaobject factory selects the appropriate metaobject for each CORBA object according to the fault-tolerance strategy specified in a configuration file. Depending on the fault assumptions, the fault tolerance strategy may impose some fault-containment requirements. The metaobject factory can first create metaobjects as separate runtime entities. Clearly, improving fault-containment introduces performance overheads due to the creation and interactions between distinct processes. When only crash faults are considered, the metaobject factory can create metaobjects into its own address space for performance reasons.

Group Communication Service: This service provides an abstraction of the underlying group communication system. It is designed as OGS [6] but is implemented with the xAMp package [20] providing multicast communication protocols and group membership. Changing the group communication system does not require any modification of the clients.

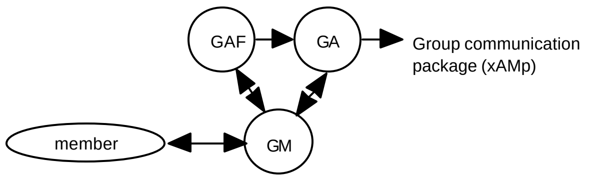

Fig. 9: Group Service Entities

Figure 9 shows the various entities implementing this service. The *GroupMembers (GM)* maintain the relationship between a group member and the group service. Clients only interact with *GroupMembers* and thus are totally independent from implementation issues of this service. This has a very positive effect on portability. The *GroupAdmins (GA)* are responsible for the management of a group on a node, i.e. of all *GroupMembers* located on this node. The *GroupAdminFactory (GAF)* creates new groups and the corresponding *GroupAdmins* on demand.

4.2. Fault-Tolerance Mechanisms as Metaobjects

A metaobject implementing a primary-backup strategy has been developed to illustrate the use of the metaobject protocol. Metaobjects were designed using Statecharts and UML tools. We give in Fig. 10 an example of a statechart for the primary metaobject. A statechart is like a simple finite state machine, excepted that states can be nested and concurrent (this later point is represented by dotted lines).

In the modelling of the primary behaviour, we identified three main roles, represented here as three different states:

- the Replication state for performing the main computation of the strategy,
- the Crash Handler state for handling view changes and crash detection,
- the Deliver state for handling messages reception.

To simplify the handling of the various events into this mechanism, these three states are running concurrently and communicate through a FIFO message queue.

The Replication state performs the replication strategy: when an invocation is received (from state Deliver), it invokes the method (state Invoke), build a checkpoint (state SaveState), send the checkpoint to backups (state Update) and then reply to the client (state Reply). When a crash occurs (received by the state Crash Handler), it creates a new clone using the ObjectFactory (state Clone), get the state of its base object (state SaveState) and send this state to the new backup (state Recover).

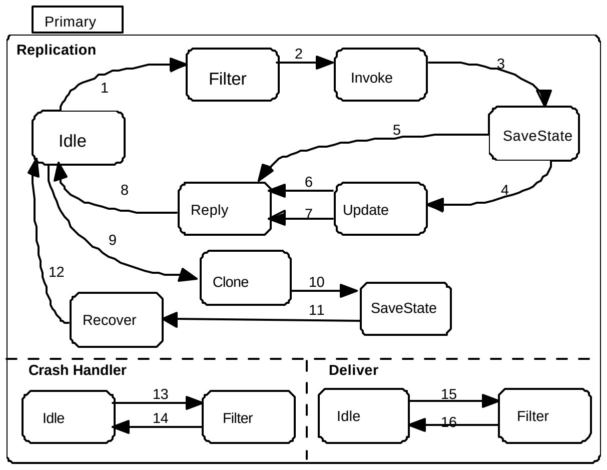

Fig 10. Statechart of the primary metaobject

Clearly, our objective here is not to describe the mechanism in detail but to illustrate that the metaobject protocol is appropriate for the design of such mechanisms. The metaobject protocol activates the mechanism at the metalevel, i.e. provides the input messages received (that trigger transitions) and activates the base-level actions, i.e. corresponding to output messages produced.

- Reification of the object methods invocations (*Meta_MethodCall*) and inter replica protocol messages are received by the Deliver state. These messages are filtered and passed to the Replication state through the message queue.
- The Replication state handles these messages according to the replication strategy. It uses intercession for invoking methods on the base object (*Base_MethodCall*) and also uses introspection for obtaining its state (*Base_SaveState*).
- The backup, not shown here, also uses intercession for applying checkpoints to the base object (*Base_RestoreState*).

It is worth noting that the activation of the crash handler is not performed by the MOP, but by the error signal coming from the group communication service.

The tools we used for the design of the statecharts are able to animate the model: running step-by-step the various communicating automatons, enabling to produce events, messages, or to look to internal variables and states, as a real debugger. Actually, this helped us to firstly debug the design of the mechanisms and secondly, to incrementally improve the design of the fault-tolerance mechanism: handling multiple clients and nested calls, for instance.

This detailed analysis using Statecharts, not only facilitates the design of the mechanisms, but greatly reduced the cost of their implementation. Indeed, it was relatively easy to implement these mechanisms, although the translation from statecharts to real-code is not straightforward. Actually statecharts communicates using messages and not method invocation, and an automaton was used to simulate the MOP, as well as group communications. Furthermore, the state machine is currently used in the implementation and the statechart model can then be used for debugging.

It is worth noting that the separation of concerns between objects and metaobjects greatly simplified the implementation of the inter-replica protocol. The clear interface provided by the MOP plus the development of metaobjects using UML and statecharts greatly facilitates the reuse of the software produced for implementing other strategies.

4.3. Preliminary Results

The CORBA architecture and the fault-tolerant mechanisms have been recently implemented on top of the Orbacus ORB running on Solaris. Preliminary results are thus provided.

Figure 11 shows some behaviour-related results : object and metaobject creation, replica creation and remote method invocations. Three methods have been used:

- Method_1 is very simple and just increments an attribute,
- Method_2 is more complex and writes 1 kilobyte into a file, and
- Method_3 has many complex parameters and thus requires heavy marshalling and unmarshalling of the arguments.

The figures have been obtained using two kind of metaobject factories, *Factory_1* which creates metaobjects in its own address space, *Factory_2* which uses the object factory service for creating independent metaobjects.

	Not Reified	*Factory_1*	*Factory_2*
Object and metaobject creation	4 ms	11 ms	1 s
Replica creation	0.9 s	1 s	2 s
Method_1 invocation	0.5 ms	2.5 ms	5.1 ms
Method_2 invocation	4.3 ms	6.3 ms	9 ms
Method_3 invocation	3.1 ms	5.7 ms	9.4 ms

Fig. 11: Creation and invocation performance

Indeed, the cost of reflection can appear to be excessively high for an empty method (10 times higher for Method_1/Factory_2), however, for regular methods it is more suitable (2 times for Method_2/Factory_2). It is worth noting however that, as shown in [5], this cost (approximately 5 ms.) must be compared to the runtime cost of fault tolerance protocols, which is actually very high (approx. 100 ms. for a primary-backup strategy).

	Class_1	Class_2	Class_3	Class_4
Full	0.02/0.01	0.08/0.03	1.4/0.4	1359/390
Partial 0%	0.02/0.17	0.25/1.9	1.6/34	20/509
Partial 20%	0.02/0.17	0.8/1.9	16.7/34	75/532
Partial 40%	0.05/0.17	1.5/1.9	32/34	230/543
Partial 60%	0.05/0.17	2.3/1.9	40/34	499/576
Partial 80%	0.2/0.17	2.7/1.9	62/34	838/625
Partial 100%	0.2/0.17	4/1.9	70/34	1370/673
Java	2/1	3/4	4/20	157/204

Fig. 12: Full and partial state (preliminary) performance

Figure 12 provides preliminary results for the state management of CORBA objects. Four different examples have been used: Class_1 contains only simple attributes (int, float, string), Class_2 contains arrays of small simple typed elements, Class_3 also contains arrays but each array has hundreds of elements, and finally Class_4 is a linked list of hundreds Class_1 objects. It is worth noting that the latter implies recursive calls to the state handling methods. For each example, we give the average time (in ms) for saving/restoring the full state of the corresponding CORBA object, but also for the partial state assuming that a given percentage of the state was modified during a method execution. The provided figures are also to be compared with related services provided by Java Serialization.

These results clearly indicate that partial state techniques are interesting for complex data structures that involve dynamic creation or destruction of objects. For simple objects, the full state technique is more efficient. For complex objects (i.e. Class_4), the partial state technique is very useful, especially when the state of the object is slightly modified. This is also true when comparing with Java Serialization, the latter being more efficient to get/restore the full state of complex objects (i.e. Class_4). This is an indication that reflection at underlying layers can improve the performance of these techniques. Performance measures of the fault tolerance mechanisms are currently performed.

5. Lessons learnt

The implementation of behavioural and state management using compile time reflection has many merits. Clearly, the first benefit is to be able to control and filter programming conventions in order to guarantee a satisfactory object model, i.e. enforcing the encapsulation principle, mandatory for dependability. The programming conventions obviously reduce some of the features of the language but they must be obeyed to improve dependability. It is worth noting also that these conventions enforce the basic model required for the fault tolerance strategies. Because the compilation process can be mastered, the necessary hooks can be easily inserted to control the behaviour of CORBA objects. This feature is essential to design and implement a runtime metaobject protocol on a “black-box” runtime layer. The design philosophy can be thus summarised as follows: (i) identify the necessary information that must be observed and controlled, (ii) design the compile time metaclasses responsible for the corresponding code transformation.

Clearly, all necessary features cannot be handled at the language level, and some openness of the underlying layers is required. Handling context switches is more difficult, and reflection at the lower layers of the system is highly recommended. However, scheduling decisions can be handled at the metalevel as in [13] but this may not be consistent with the concurrency models provided by CORBA. However, it is clear that some information can only be obtained at the language level. Open languages thus provide an appropriate technique to adjust the program structure and behaviour according to the requirements of the non-functional mechanisms. It is a good approach on top of a COTS middleware.

Unfortunately, an object’s state may also be composed of site-dependent information, such as file descriptor identifiers in Unix environment. We call such information local variables because their semantics and value is local to a given site. Since local variables are not within the application objects but are, nevertheless, part of their state, this information is called the external state of the object. Clearly, our serialization technique does not presently tackle this problem. Local variables could be handled at compile time, provided the user declares the corresponding system calls; these calls can be intercepted and reified to the metalevel. Declarative reflection is thus helpful in this case, as in [8] .

In practice, our technique requires access to the source code of each class used in the application. This assumption cannot always be met for class libraries. It is thus difficult in this case to implement the partial state technique because we cannot know whether an attribute (or object) passed as parameter has been modified. To tackle this problem, we assume it is modified by default, and thus partial state information is not minimal. Also, when a library class is used or derived (a class inherits from a library class), the MOP cannot get the state of the library class. This can be avoided and filtered by the metaclass or we can let the user do the job.

Finally, the system architecture developed in this paper is a framework providing interesting properties for system designers in many application fields. Fault tolerance strategies are implemented as independent CORBA objects. They can thus be reused and customised according to the needs. The approach also provides means to adapt the strategies at runtime, with respect to the evolution of the system configuration and its surrounding environment. In the next future, various implementation strategies and extensions could be applied to this generic MOP. Interceptors could be used instead of method wrapping, state serialization instead of compile time reflection, the external state could also be included in state data structures. The openness of the runtime layers could also provide new reified information and hooks (context information, thread switching, etc.) for fine grain synchronisation of the fault tolerance metaobjects.

6. Conclusion

Although reflection has been recognised as a powerful concept for the implementation of non-functional mechanisms, very few work address the real challenge of implementing a reflective fault tolerant system. This paper, which focuses on implementation issues, shows that implementing fault tolerance ideally requires extensive reflective features. Considering the underlying support as a black box involves using reflective features at the upper layer. Runtime reflection implemented with a metaobject protocol enables distributed fault tolerance to be implemented with attractive properties (adaptation and reuse). Building a runtime MOP is thus the difficult issue to be solved. Compile time reflection provides means to this aim, although all aspects cannot be controlled and some programming convention must be obeyed. It is worth noting however that this solution enables a reflective approach to be implemented on top of standard middleware, such as CORBA, enabling fault tolerance strategies to be developed as CORBA software. This is a first positive aspect of the work reported in this paper. In addition, the specifications of this MOP are now stable and their implementation can be optimised very much using reflective features of the underlying layers, now appearing in many executive supports and standards. Controlling some fine grain aspects of the underlying layers, i.e. extending the current specifications, will also be possible for the same reason, in particular using Open ORBs. This is the main track for our future investigations.

7. References

[1] G. Agha, S. Frolund, R. Panwar, and D. Sturman, “A Linguistic Framework for Dynamic Composition of Dependability Protocols,” presented at DCCA-3, pp. 197-207, 1993.

[2] J. Brant, B. Foote, R. E. Johnson, and D. Roberts, “Wrappers to the Rescue,” presented at ECOOP'98, Brussels, Belgium, pp. 396-417, 1998.

[3] S. Chiba, “A Metaobject Protocol for C++,” presented at ACM OOPSLA'95, Austin, Texas, USA, pp. 285-299, 1995.

[4] S. Chiba and T. Masuda , “Designing an Extensible Distributed Language with a Meta-Level Architecture,” presented at ECOOP'93, pp. 482-501, 1993.

[5] J.-C. Fabre 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*, vol. 47, pp. 78-95, 1998.

[6] P. Felber, B. Garbinato, and R. Guerraoui, “The Design of a CORBA Group Communication Service,” presented at IEEE Symposium on Reliable Distributed Systems, pp. 150-159, 1996.

[7] B. Garbinato, R. Guerraoui, and K. Mazouni, “Implementation of the GARF Replicated Objects Platform,” *Distibuted Systems Engineering Journal*, vol. 2, pp. 14-27, 1995.

[8] Y. Honda and M. Tokoro, “Soft Real-Time Programming through Reflection,” presented at Intl. Workshop on New Models for Software Architecture: Reflection and Meta-Level Architecture, Tokyo, Japan, pp. 12-23, 1992.

[9] M. Kasbekar, C. R. Das, S. Yajnik, R. Klemm, and Y. Huang, “Issues in the Design of a Reflective Library for Checkpointing C++ Objects,” presented at IEEE SRDS'99, 1999.

[10] M.-O. Killijian, J.-C. Fabre, J.-C. Ruiz-García, and S. Chiba, “A Metaobject Protocol for Fault-Tolerant CORBA Applications,” presented at IEEE SRDS’98, West Lafayette, Indiana, USA, pp. 127-134, 1998.

[11] M.-O. Killijian, J.-C. Ruiz-García, and J.-C. Fabre, “Using Compile Time Reflection for Object State Capture,” presented at Reflection'99, Saint-Malo, France, pp. 150-152, 1999.

[12] P. Maes and D. Nardi, “Meta-Level Architectures and Reflection,” , Elsevier Science Pub., 1988.

[13] L. E. Moser, P. M. Melliar-Smith, and P. Narasimhan, “A Fault Tolerant Framework for CORBA,” presented at FTCS’29, Madison, Wisconsin, USA, 1999.

[14] OMG, “Meta Object Factory (MOF) Specification” OMG, ad/97-08-14, 1997.

[15] OMG, “CORBA/IIOP 2.2 Specification” , 98-07-01, 1998.

[16] OMG, “Portable Interceptors” Eternal Systems Inc., Expersoft Corporation , Sun Microsystems Inc., Initial RFP Submission, orbos/99-04-07, April, 26 1999.

[17] V. Strumpen and B. Ramkumar , “Portable Checkpointing for Heterogeneous Architectures,” in *Fault-Tolerant Parallel and Distributed Systems*, D. Avresky, R. and D. Keli, R., Eds.: Kluwer Academic Press, pp. 73-92, 1998.

[18] Sun, “Java Object Serialization Specification” Sun Microsystems, Technical Report November 1998.

[19] M. Tatsubori, S. Chiba, M.-O. Killijian, and K. Itano, “OpenJava: A Class-based Macro System for Java,” in *Reflection and Software Engineering*, *Lecture Notes in Computer Science 1826*, W. Cazzola, R. J. Stroud, and F. Tisato, Eds. Heidelberg, Germany, pp. 119-135, 2000.

[20] P. Veríssimo and J. Marques, “Reliable Broadcast for Fault-Tolerance on Local Computer Networks,” presented at IEEE SRDS-9, pp. 54-63, 1990.