A Metaobject Protocol for Fault-Tolerant CORBA Applications

Marc-Olivier Killijian[*], Jean-Charles Fabre[*], Juan-Carlos Ruiz-Garcia[*], Shigeru Chiba[**]

[*] *LAAS-CNRS, 7 Avenue du Colonel Roche 31077 Toulouse cedex, France*

[**] *Institute of Information Science and Electronics, University of Tsukuba, Tennodai, Tsukuba, Ibaraki 305-8573, Japan*

Abstract

The use of metalevel architectures for the implementation of fault-tolerant systems is today very appealing. Nevertheless, all such fault-tolerant systems have used a general-purpose metaobject protocol (MOP) or are based on restricted reflective features of some object-oriented language. According to our past experience, we define in this paper a suitable metaobject protocol, called FT-MOP for building fault-tolerant systems. We explain how to realize a specialized runtime MOP using compile-time reflection. This MOP is CORBA compliant: it enables the execution and the state evolution of CORBA objects to be controlled and enables the fault tolerance metalevel to be developed as CORBA software.

1. Introduction

The implementation of fault tolerance in object-oriented systems has been investigated using various approaches, either based on the use of inheritance [1,2] and reflection [3-5] or investigating their implementation on/within CORBA runtime supports [6-8]. Reflective architectures have been developed in various research areas [9], in particular regarding dependability issues. The use of reflective ideas [10] and metaobject protocols (MOPs) [11] provides many interesting properties, such as transparency and separation of concerns. Reflection enables an object-oriented application to be controlled at a higher level of abstraction, the metalevel. Mechanisms previously developed as standard and object-oriented libraries or else integrated into an operating system and into an ORB are then developed at the user level as metalevel software. Beyond transparency and separation of concerns, this approach also provides visibility of the mechanisms that can be easily customized for a given application or system configuration using object-oriented techniques. Visibility enables the fault-tolerance software to be manipulated at the user level (i.e. not embedded in the operating system layers, including the ORB). Nevertheless, to our knowledge, most of the examples are based on existing general-purpose metaobject protocols or restricted reflective features of some object-oriented languages. They have thus to cope with their limits.

The *corner stone* of a fault-tolerant reflective architecture is the MOP. We thus propose a special purpose MOP to address the problems of general-purpose ones. We show that compile-time reflection is a good approach for developing a specialized runtime MOP.

The definition and the implementation of an appropriate runtime metaobject protocol for implementing fault tolerance into CORBA applications is the main contribution of the work reported in this paper. This MOP, called FT-MOP (*Fault Tolerance - MetaObject Protocol*), is sufficiently general to be used for other aims (mobility, adaptability, migration, security). FT-MOP provides a mean to attach dynamically fault tolerance strategies to CORBA objects as CORBA metaobjects, enabling thus these strategies to be implemented as CORBA software. It also aims at handling the state of objects automatically to solve the problem of user-defined inherited functions (e.g. *save_state* and *restore_state*); a wrong definition of these state handling functions (state information missing) prevents the fault tolerance software to perform its recovery actions correctly.

However this MOP cannot cope with all non-functional requirements; hard real-time aspects for instance, as discussed in [12]. Although, FT_MOP is very language dependent (C++ today) it can be adapted to other object-oriented languages, e.g. Java. In practice, the fact that such a MOP is language-dependent is mandatory in fault-tolerant computing as explained in the paper.

In Section 2, we describe the basic concepts for implementing a given runtime MOP using compile-time reflection. We describe in section 3 the distributed object model we consider and the requirements for implementing fault tolerance using a runtime metaobject protocol. Section 4 describes the design and the implementation of FT_MOP. Section 5 briefly describes some related works.

2. Building a Runtime MOP

The main objective of this section is to describe how to build a specialized runtime MOP for fault-tolerant systems. We describe first our motivations according to past experience in building fault-tolerant systems using metalevel architectures and we introduce both compile-time and runtime reflection for implementing FT-MOP.

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

2.1. Motivations

The implementation of fault tolerance previously done in FRIENDS [5], using a general purpose runtime MOP, showed that the use of a metalevel architecture is very attractive for implementing flexible fault-tolerant systems. The FRIENDS system provides very interesting properties such as transparency, separation of concerns, composition, etc. Because of these properties, application objects can gain fault tolerance (and/or communication security) according to the needs and metaobjects (i.e. fault tolerance strategies) can be developed independently and easily customized according to different fault assumptions and system configurations. Although runtime reflection is often costly in terms of performance, it has been shown in previous experiments reported in [13] that the inherent cost of fault tolerance mechanisms by replication is several orders of magnitude higher than the cost of handling the object-metaobject interaction at runtime.

However a general purpose MOP is not always satisfactory. With the general purpose runtime MOP used in FRIENDS (OpenC++ V1), we have been faced to some limits: handling composition and inherited methods in application objects is not possible, some important meta-information is missing for handling the state of objects, etc. Additionally, the link between objects and metaobjects is defined at compile-time and thus cannot be changed at runtime. The dynamic adaptation of the fault tolerance strategy according to different operational conditions and fault assumptions was not possible with this MOP. Additionally, another drawback of the FRIENDS system is that it is not CORBA compliant. Similar limits with general purpose MOPs or reflective features can be observed in other reflective fault-tolerant systems, e.g. GARF [4] based on reflective features of Smalltalk and MAUD [3] based on the actor model and the Hal language. These are the main reasons for the development of a customized runtime MOP for fault tolerance, compliant with CORBA and able to handle the internal state of objects (variables and objects).

In summary, this work is a big step forward with respect to previous work done in FRIENDS. It enables:

- full use of object-oriented programming features to develop complex applications,
- dynamic link object-metaobject (adaptativity),
- use of CORBA compliant runtime supports,
- automatic handling of the object state (optimized when possible),
- development of FT strategies as CORBA software.

2.2. Compile-time and runtime MOPs

Metaobject protocols are a reflection-based technique to make a compiler (or an interpreter) extensible. For example, a C++ compiler with a MOP takes two C++ programs to produce a single object file. One is an ordinary source program, and the other is a C++ program that customizes the compiler and makes extended language mechanisms available from the source program. To distinguish the two programs, we call the former a base-level program, and the latter a meta-level program.

The meta-level program customizes the compiler through an object-oriented interface called a metaobject protocol. In general, this interface is categorized into runtime MOPs (e.g. OpenC++ V1 [14]) and compile-time MOPs (e.g. OpenC++ V2 [15]), depending on how to describe the customization.

A basic idea of runtime MOPs is to provide hooks to trap a method call, object creation, and so forth, at runtime and to let the programmers alter the behavior of the trapped actions. The trap handlers are defined as an object called a metaobject since the MOPs are object oriented. The programmers can associate a base-level object with a different metaobject so that they can control the behavior of the base-level object as they want.

Compile-time MOPs provide hooks to transform specific expressions and declarations in the base-level program during the compilation. Like runtime MOPs, the trap handlers are defined by a metaobject although the metaobject does not exist at runtime but it is part of the compiler. Although compile-time MOPs can be regarded as a sort of macro system, they provide more semantic information of the program than macro systems and enable the scope control where the transformation is applied.

Both runtime and compile-time MOPs have pros and cons. A drawback of runtime MOPs is their runtime penalties[1]. Since all the meta computation is executed at runtime, the metaobjects need to execute even the computation that can be done using static information at compile-time. Although compile-time MOPs include almost no runtime penalties, the metaobject of those MOPs cannot access runtime information to determine the behavior of the base-level program.

To avoid those problems, this paper proposes the use of a general-purpose compile-time MOP to implement a runtime MOP specialized for a specific application domain, in our case, fault tolerance. The integration of runtime and compile-time MOPs enables more efficient functionality for fault tolerance than we have done by only a runtime MOP in the FRIENDS system. The compile-time metaobjects transform the base-level program so that minimal runtime hooks are embedded in the program. Then, the base-level program is linked to another meta-level program including runtime metaobjects, associated with the base-level objects through the hooks inserted by the compile-time metaobjects (see. Fig. 1).

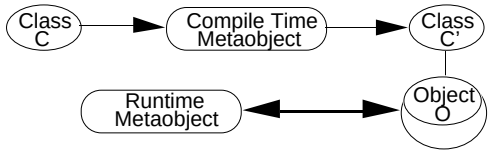

Fig.1. Compile-Time vs. Runtime Metaobjects

[1] Not significant for distributed fault tolerance strategies.

The compile-time metaobjects execute meta computation as much as possible and insert the results in the base-level program.

3. Requirements for FT-MOP

In order to define an appropriate metaobject protocol for implementing fault-tolerant applications, we firstly define the computational model that we consider: what is a distributed object-oriented applications and what is an object for us. These definitions are quite general and comply with the CORBA model. Then, we comment on what is needed for implementing fault-tolerance.

3.1. Application model

A distributed object-oriented application is a collection of active objects interacting by messages (see. Fig. 2).

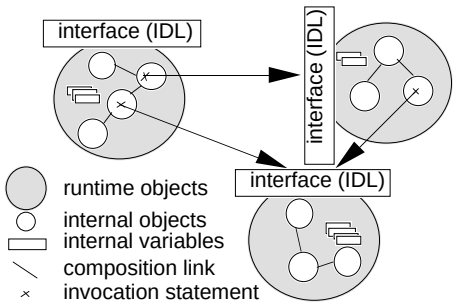

Fig. 2. Object-oriented application model

Any object is defined by an interface and encapsulates a state composed of various attributes, either objects (sub-objects) or variables defined using standards types of an object-oriented programming language. The interface of any object is composed of several methods that have typed parameters and are the only mean to manipulate internal attributes. The implementation of the interface can be done using a conventional class-based object-oriented language. The implementation may use internal variables and objects which are not visible to the outside world, i.e. that do not have an external interface. The latter objects do not correspond to separate runtime entities as the former. The brief definition given here complies with several models including the CORBA model, i.e. we see a CORBA application through the model presented here. Internal concurrency among object methods within a runtime object is not considered for the moment. Extending FT-MOP to handle intra-concurrency is a future objective.

3.2. Fault tolerance requirements

Implementing fault tolerance using replication strategies in a distributed system imposes being able to control the following aspects of the runtime objects:

- *object creation/deletion*: depending on the fault tolerance strategy, the object has to be created into multiple copies (the replicas) and all copies have to register in a communication group. Deleting an object implies removing all copies from this group.
- *object invocation*: any invocation to replicated objects involves broadcasting messages through a group communication system which ensures, at least, that all correct replicas will receive all messages in the same order (this is required for replica consistency). In case of replicated calling objects, all replicas must be able to control multiple copies of request messages (filtering). After execution of the method, state information transfer (checkpoints) or synchronization among the replicas must be performed.
- *object state evolution*: whatever the strategy is, the object state must be maintained consistent among all replicas. For active replication, the basic assumption required is determinism of objects. In this case, the same inputs lead to the same outputs and the same transformation of the internal state. For strategies either based on stable storage or passive replication, then the object state has to be sent periodically to the stable storage service or to the backup replica, respectively. Being able to handle the object state (variables and internal objects) is always required for cloning a new object replica during the reconfiguration process after a failure. This involves being able to handle the state of an object-oriented program.

As in similar works, we also assume that the failure unit is a process and that processes are fail-silent. However, arbitrary faults can be tolerated when the communication system is running on a fail-silent NAC (Network Attachment Controller) as shown in [16].

3.3. Controlling object execution

Object methods are executed sequentially and update the internal object attributes, its state. The objective is here to be able to control the behavior and the state of an object O within a metaobject ^O, a runtime object. The behavior can be observed as follows:

- control is given to ^O for the creation of O,
- control is given to ^O for each invocation to O,
- control is given to ^O for the deletion of O,
- ^O can activate base-level methods of O
- ^O can get and manage a consistent copy of O state

Handling the object state requires very fine grain information of the object program since it needs structuring information which are: identification of attributes and methods, type information for attributes and method parameters, internal classes used. When this can be reified, the complete object state can be obtained through a recursive process handling all sub-objects down to simple typed attributes. The object state information can then be packed into a predefined data structure and delivered to the runtime metaobject ^O on demand. The reified information needed here can only be obtained through compile-time reflection where all the structuring and type information is available. Clearly, handling the object state is very language dependent.

4. A CORBA compliant MOP

4.1. Overall architecture and MOP definition

We briefly describe here the overall architecture which is considered. The design decisions regarding the organization of the fault tolerance metalevel and the use of this MOP for implementing mechanisms are left open today. This means that several solutions can be investigated using this MOP.

Our reference architecture, given in Fig. 3, complies with other domain-specific meta-level architecture [17]; it is composed of a meta-level architecture for structuring fault-tolerant applications, a collection of components implementing the fault-tolerant strategies (the metaobjects) and a metaobject protocol providing a nice interface between base and meta level(s), this is FT-MOP.

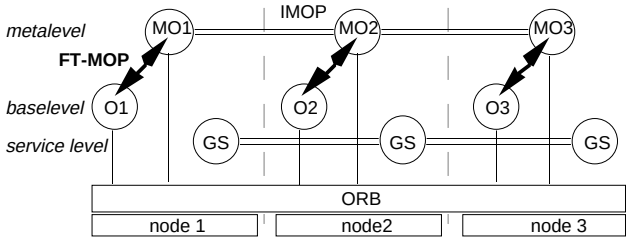

Fig. 3. Overall architecture

Mandatory services such as a group communication services (GS), replication domains management, error detection and object factories constitute the basic layer. Above this basic layer of CORBA services, metaobjects implement fault tolerance strategies. The FT-MOP controls the behavior and the state of base level CORBA objects (application level). Since metaobjects are CORBA objects they have also access to the ORB. The Inter MetaObject Protocol (IMOP) concerns the combination of several metaobjects at the metalevel; the composition of such metaobjects can be achieved either recursively (as in FRIENDS, or MAUD since a metaobject is an object) or based on a meta-scheduler triggering several metaobjects within the same metalevel.

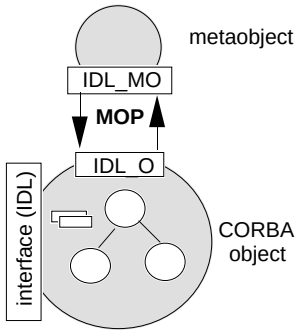

Fig. 4. Object-metaobject interaction with a MOP

In this architecture, FT-MOP is defined by two IDL interfaces (see Fig. 4): the object side and the metaobject side. The behavior observation consists in reifying the different operations of our object model: instance creation and deletion, methods invocations and the object state. Both IDL interfaces (IDL_O for the object side, IDL_MO for the metaobject) are given in Table 1.

IDL_O

```
Interface Object {
//Base_HandleCall dispatchs methods invocation
void Base_Call(in long m_id, inout ArgPac args,
               out ArgPac reply);
// Base_StartUp calls the original constructor
void Base_StartUp(in ArgPac args);
// Base_Cleanup calls the original destructor
void Base_CleanUp();
// Base_State gets updated attributes
void Base_State(out sequence<ID> attributes);
// Base_GetValue gets the value of an attribute
void Base_GetValue(in ID id,out Any val);
};
```

IDL_MO

```
Interface MetaObject{
  // Implementation of trapped invocation.
   void Meta_MethodCall (in ID method_id,
                         inout ArgPac args,
                         out ArgPac reply);
  // Implementation of trapped destructor.
   void Meta_CleanUp();
};
```

Table 1. IDL interfaces for FT-MOP

Although objects and metaobjects are separated runtime entities, they run on the same site and are linked by a synchronous communication protocol (procedure calls). Accordingly, atomicity of the pair (object-metaobject) is ensured; a failure of the metaobject involves leaving the communication group (failure of a group member); a failure of the object is either detected by the local communication protocol (raised exceptions) or by a timer handler also raising an exception. These exceptions are handled by the metaobject that leaves the group.

The following sections describe how these features are realized using the Open C++ V2 meta compiler. For the ease of explanation, we have switched the order given in section 3 and start with the handling of method execution in section 4.2, then of the object creation-deletion in section 4.3 and of the object state in section 4.4. Some aspects for handling method calls given in section 4.2 (i.e. renaming) apply also to constructors and destructors.

4.2. Handling methods invocation

The trapping of base level method calls is done by renaming these methods: (i) any original method `foo` is renamed by adding "real" to its name (i.e. `real_foo` now), (ii) and the hook which traps the operation call uses the original name (i.e. `foo`). This means that the initial object method `foo` now forwards the call to a runtime metaobject. The real body of the method is now `real_foo`. This renaming also applies to the access of public attributes and also to constructors and destructors.

Let's consider a simple class C which has a constructor, a destructor and two method; class C is translated into class C' as follows:

Initial Class	Translated Class
class C { C (); ~C(); void foo1 (); int foo2 (long, char); }	class C' { C (); ..... //Trap ~C(); ..... //Trap void foo1 (); ..... //Trap int foo2 (long,char);//Trap //Real bodies void real_Startup(); void real_Cleanup(); void real_foo1 (); int real_foo2 (long,char); }

Table 2. Renaming method name and invocation trap

We mean by public method, any method which has been defined in the IDL interface of a class, this definition consists in the method's name, its return type and a list of typed parameters.

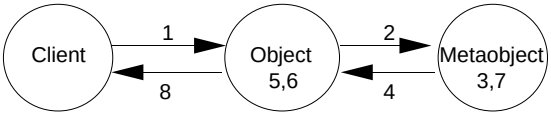

Fig. 5. Method invocation

In Fig. 5, the client invokes a method of the object, say `foo`, using the required parameters (1). The MOP traps this call, packs the parameters and call the `Meta_MethodCall` method of the metaobject (2) with the packed parameters. The metaobject may perform some actions before calling the base level method (3). Then, the metaobject calls the `Base_HandleCall` method of the object (4). This method, according to the method number, unpacks the parameters and call the original method (5), say `real_foo`. The return value is packed and returned to the metaobject (6). The metaobject may execute some action before returning to the object (7) which in turns returns to the client (8). Table 3 gives an example of a simple method translation:

Initial Source code	Translated Source Code
void C::foo1() { cout << "foo1 was called" << endl; }	void C::foo1() { mo->MethodCall(2,a,r); // mo is a ref to the MO // foo1 method id is 2 // a is foo1 args // r is return value } void C::real_foo1() { cout << "foo1 was called" << endl; }

Table 3. Method translation

The MOP operations need to have a stable interface, so arguments and return value of the methods are packed into a common data structure, called *ArgPac*. Hopefully, CORBA defines the type *Any* which can store data of any type which is used in IDL interfaces. Moreover, it also defines the abstract type *Sequence* which can be instantiated to store a sequence of any given type. These two features are combined for defining *ArgPac* as a *Sequence* of *Any*, providing a clean and portable solution for the arguments packing. The IDL code to define an *ArgPac* is: `typedef sequence<any> ArgPac;` . Table 4 gives an example of a hook-method which receive an invocation, packs the arguments and call the metaobject.

Translated Source Code (for foo2)
int C::foo2(long a, char b) {   argPac* reply;.... // To pack the return value   argPac pac; ..... // To pack the arguments   int fake; ..... // To unpack the return value   pac.length(2);..... // There are two arguments   pac[0] <<= (CORBA_Long) a;..... // Pack arg #1   pac[1] <<= CORBA_Any::from_char(c);..... //#2   mo->MethodCall(3,pac,reply);... // Call the MO   (*reply)[0] >>= (CORBA_ULong&)fake;   return(fake); ..... //Unpacks and replies }

Table 4. Method trapping with parameters packing

Using the Interface Definition Language (IDL) of CORBA, a public attribute can be accessed with two methods: one for reading (`Get`), one for writing (`Set`). Handling these two methods at the metalevel is done in the same way as for any other user-defined method

4.3. Handling creation-deletion

The instantiation process creates an object of a given class. Usually this creation process is initiated by another object or by a program (the main procedure in C++); the initiator gets a CORBA reference on the new instance.

During the creation process, a new runtime metaobject has to be created for each new object. An instance of metaobject is created using a Metaobject Factory.

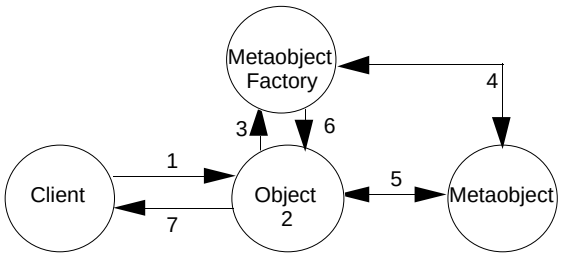

Fig. 6. Creation of participants

As illustrated in Fig. 6, the client requests the creation of an object, i.e. it calls the constructor (1). The constructor is executed but it has been translated in order to trap this call. It initialises the CORBA environment and calls the naming service for a reference to the Metaobject Factory (2). Then, the object constructor calls the Metaobject Factory to create a new metaobject of a given metaclass (3). The Metaobject Factory invokes the metaobject constructor (4). The metaobject constructors calls the original object constructor (5), i.e. `Base_Startup` which in turns activates `real_Startup` of

the base level object. A reference to the metaobject is then returned to the object (6). Finally, a reference to the object is returned to the client.

The implementation involves translating the constructor and copying the original one into a `real_Startup` method; the translation of a simple constructor is illustrated in Table 5.

Initial Source code	Translated Source Code
`C::C()` `{` `A=0;` `}`	`C::C() {` `// 1. CORBA INIT.` `// 2. Get MOFactory ref.` `// Create a MO of a given type with` `// object ref., MO type and args.` `mo=MOFac->create_metaobject(o_r,t,a);` `}` `void C::real_Startup() {` `A=0;` `}`

Table 5. Constructor translation

During the deletion process, the metaobject must also be deleted when the object is deleted.

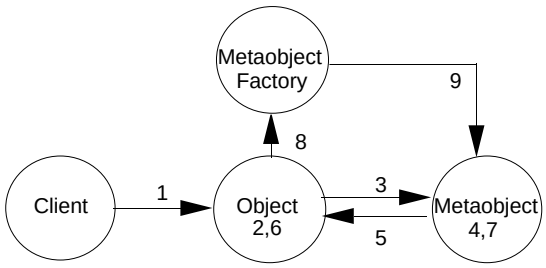

Fig. 7. Deletion of participants

The client requests the deletion, i.e. it invokes the destructor (1) of the object (see. Fig. 7). As previously explained, the destructor has been translated in order to trap this operation and thus behaves differently (2). The destructor calls the `Meta_Cleanup` method of the metaobject (3). The metaobject may execute some "pre-deletion" actions (4) and calls the original destructor, `real_Cleanup`, using the `Base_Cleanup` method of the base level object (5). The metaobject may now execute some "post-deletion" actions (7). Finally, the object invokes the Metaobject Factory to delete the metaobject just before to destroy itself (completion of the object destructor). This process is illustrated with the translation of a simple destructor (see Table 6.):

Initial Source code	Translated Source Code
`C::~C()` `{` `cout << "End" <<` `endl;` `}`	`C::~C()` `{` `mo->MetaCleanUp();` `MOF->destroy(mo);` `CORBA_release(mo);` `}` `void C::real_Cleanup()` `{` `cout << "End" <<` `endl;` `}`

Table 6. Destructor translation

4.4. Handling the object state

Our first objective is to handle the object state automatically and secondly to minimize, as far as possible, the state information to be transferred to object replicas. This is language dependent and involves a deep analysis of the target source code using compile-time reflection. The runtime metaobject is responsible for handling a consistent copy of the base level object state. The *object state* corresponds to the whole set of attributes. From the initial state, the runtime metaobject can get the part of the state which has been updated after the execution of every method. This is called *delta_state*. The metaobject triggers method execution using `Base_Call` and gets the delta-state of the object using `Base_State` as shown in Table 1. `Base_State` returns the set of modified attributes IDs to the metaobject. When necessary (checkpointing, cloning), the metaobject gets the corresponding values of these attributes using `Base_GetValue`.

The metaobject thus monitors the state evolution of the object it controls. The way the state information is handled by the metaobject is open and may vary according to the various ways of implementing a given fault tolerance strategy: the whole state can be sent to replicas after every method execution or only a sequence of delta-states can be sent from the initial state (notion of *incremental checkpointing*). The second approach is far more efficient in practice.

A metaobject builds the object state by getting the initial state S_I at object creation and then delta-states ΔS_i after each method invocation. The main problem is to obtain the delta-state ΔS_i after the i^{th} method invocation. To tackle this problem, we use compile-time reflection and statically analyze the program. Depending on the complexity of the program, this static analysis enables the appropriate technique to be selected for each pair (method, attribute) of the object. Three techniques have been investigated and are described in the next sections.

In any case, at the completion of every method execution, the delta-state ΔS_i must be available for the metaobject and must include all modified attributes. It is worth noting that a coverage of 100% of the updated object state is mandatory, even if it is not minimal.

CTO - Compile-Time-Only. This technique aims at determining the delta-state at compile-time. Clearly, the delta-state cannot be determined statically, except for very simple method code. Mainly for didactic reasons, we illustrate here the use of compile-time reflection to detect a write access to an attribute. A compile-time metaobject insert some code in the method to produce the set of modified attributes, ΔS_i. For example, considering a simple `class C` containing a simple method `foo` which modifies directly[2] only one attribute `index`, the compile-time metaobject generates the following code of `real_foo`

[2] Write access to pointed attributes can also be detected.

(see Table 7. - *boldface* characters indicate inserted code by compile-time metaobjects for handling the delta-state):

Initial Source code	Translated Source Code
class C { public: void foo(int i) { index=i; } private: *// S=all attributes* int index,j,k,l,m,n; };	void C::real_foo(int i) { *// Detect write access* index=i; *// Insert index in ΔSi* **Δ-State.Insert(index);** }

Table 7. CTO simple example

CRT - Compile-Run-Time. In most case, the delta-state must be determined dynamically, i.e. which are the attributes that will change at runtime. Indeed, attribute modifications can be included in conditional or iterative blocks of code. Thus, these modifications depend on input parameters. The objective of the Compile-Run-Time technique is to compute in two phases (at compile-time and at runtime) the proper set of attributes ΔS_i for any method. The first phase, taking place at compile-time, consists in:

- determining the set of attributes which are likely to change at runtime during a method invocation;
- inserting just after each write access an instruction to flag the attribute (see. Table 8.).

Instrumenting the code in this way enables, during the second phase (at runtime), the exact set ΔS_i to be determined at the end of an invocation by checking the flags. Here is an example of the use of the CRT technique on a new method `foo`:

Initial Source code	Translated Source Code
class C { public: void foo(int i) { if (i == 0) index=i; else if (i < 0) j=i; else k=i; } private: int index,j,k,l,m; };	void C::real_foo(int i) { **boolean flag[2];** **for (int a=0;a<=2;a++)** **flag[a]=False;** if(i==0) { index=i; **flag[0]=True;** } else if (i<0) { j=i; **flag[1]=True;** } else { k=i; **flag[2]=True;** } *// Insert attributes* *// in ΔSi* **if (flag[0]=True)** **Δ-State.Insert(index);** **if (flag[1]=True)** **Δ-State.Insert(j);** **if (flag[2]=True)** **Δ-State.Insert(k);** }

Table 8. CRT simple example

CWS/CCS – Copy-Whole/Compare-State. Obviously, C++ is a permissive language: the programmer is able to use pointer arithmetic or to pass arguments by reference as function parameters (either using a pointer or a C++ reference). Such programming techniques generate "unpredictable modifications" of attributes and can lead to situations where the delta-state cannot be determined. For instance, when pointer arithmetic is used, any attribute can be accessed without compile-time reification of such a write access. For safety reasons, we may assume here that all attributes have changed.

Thanks to compile-time reflection such situation can be identified and computing the delta-state is done automatically at runtime, thus providing safe checkpointing for any recovery strategy. The whole state can be obtained automatically after every method execution – CWS. CCS involves saving all attributes before a method call and comparing their values to the new one when the method completes. Compile-time reflection is used here to insert code during code analysis for getting the before-state and after-state, also for computing the delta-state. For attributes defined using object classes, the handling of delta-state is performed recursively.

Clearly, these techniques are very costly in performance terms, but this is the price to pay for a language like C++. The use of a "*clean*" object-oriented language with a strong typing system will make these techniques useless.

5. Related work

Different approaches have been used for the implementation of fault-tolerant CORBA-based systems. Some consist in developing a specific ORB able to handle group communications and fault tolerance strategies (the *integration approach*) [18] or consider off-the-shelf ORBs (the *interception approach*) [7] and implement fault tolerance by diverting messages to a group communication system and other components responsible for handling replication. In either case group management and fault tolerance strategies are not always visible at the application level and, as far as we can understand from available documents, not easy to customize. Another approach consists in providing a group communication service (the *service approach*) to applications [8]. In this case, the use of groups and fault tolerance strategies is left open to application programmers which are responsible for using these basic services directly within CORBA applications. Another major drawback of the implementation of fault tolerance using object-oriented languages and also with CORBA-based solutions is that the definition of the object state is left open to application programmers. Again, a wrong definition of these functions (state information missing) prevents the fault-tolerance software to perform its recovery actions correctly.

6. Conclusion and future work

The reflective approach described in this paper enables strategies to be visible and easily customized at the application level, although their use is kept transparent for CORBA application programmers. The use of basic services, such as group communication, is devoted to metaobject programmers, specialized in a non-functional domain (here fault-tolerance). The implementation of fault tolerance using a metaobject protocol enables off-the-shelf ORBs to be used. Additionally, this approach provides a mean to develop fault tolerance software as any CORBA software with different object-oriented languages. This is a very good point for their reuse in many application domains. Being able to handle the object state automatically is the last benefit of the FT_MOP approach.

This work also illustrates the benefits of compile-time reflection for building a specialized runtime MOP. The same approach could be used to develop a specific runtime MOP in other application domains. Because, a compile-time MOP is a clever source-to-source translator, any recommended version of a compiler (as required in the industry) can be used to produce final binary code.

The use of the metaobject protocol described in this paper is in fact not limited to the implementation of fault-tolerant applications. Actually, this MOP may be used for handling other non-functional requirements, such as security (authentication) as done in the FRIENDS system. The automatic handling of the object state in combination with the runtime aspects of the MOP could also allow the migration of objects according to operational conditions. The dynamic linking between object and metaobjects enables as well mobile objects to gain fault tolerance and change strategies depending on the failure assumptions made on the underlying node.

FT-MOP has been partially implemented on COOL-ORB, a CORBA-compliant system available on UNIX also on the CHORUS real-time microkernel. Handling the creation, deletion and invocation of CORBA object using FT-MOP has been implemented today. Handling the object state of complex objects and performance measurements (benchmarking) are now carried out.

7. Reference

[1] Detlefs D., Herlihy M.P., Wing J.M., "Inheritance of Synchronization and Recovery Properties in Avalon/C++", *Computer*, 21 (12), Dec. 1988, pp. 57-69.

[2] Shrivastava S.K., Dixon G.N., Parrington G.D., "An Overview of the Arjuna Distributed Programming System", *IEEE Software*, 8 (1), 1991, pp. 66-73.

[3] Agha G., Frølund S., Panwar R., Sturman D., "A Linguistic Framework for Dynamic Composition of Dependability Protocols", in *Proc. of DCCA-3*, 1993, pp. 197-207.

[4] Garbinato B., Guerraoui R., Mazouni K., "Implementation of the GARF Replicated Objects Plateform", *Distributed Systems Engineering Journal*, (2), March 1995, pp. 14-27.

[5] Fabre J.C., Pérennou T., "A Metaobject Architecture for Fault Tolerant Distributed Systems: The *FRIENDS* Approach", *IEEE Transactions on Computers*, Special Issue on Dependability of Computing Systems, Jan. 1998, pp. 78-95.

[6] Landis S., Maffeis S., "Building Reliable Distributed Systems with Corba", Theory and Practice of Object Systems, (special issue on the future of Corba), vol. 3 (1), 1997, pp. 59-66.

[7] Moser L.E., Melliar-Smith P.M., "The Interception Approach to Reliable Distributed CORBA Objects," P. Narasimhan, L. E. Moser and P. M. Melliar-Smith, Panel on Reliable Distributed Objects, *in 3rd USENIX Conference on Object-Oriented Technologies and Systems*, Portland, (Or, USA), June 1997, pp 245-248.

[8] P. Felber, B. Garbinato,, R. Guerraoui. "Towards Reliable CORBA: Integration vs. Service Approach", *in Special Issues in Object-Oriented Programming*, Springer-Verlag, 1997.

[9] Stroud R.J., "Transparency and Reflection in Distributed Systems", *ACM Operating Systems Review*, 22 (2), April 1993, pp. 99-103.

[10] Maes P., "Concepts and Experiments in Computational Reflection", in *Proc. of OOPSLA'87*, Orlando, USA, 1987, pp. 147-155.

[11] Kiczales G., des Rivières J., Bobrow D.G., *The Art of the Metaobject Protocol*, MIT Press, 1991.

[12] Mitchell S. E., Wellings A. J., Burns A., "Developping a Real-Time Metaobject Protocol", in *Proc. of the IEEE Workshop on Object-Oriented Real-Time Dependable Systems*, 1997.

[13] Fabre J.C., "Design and Implementation of the *FRIENDS* System", in *Proc. of the IEEE FTPDS'98 Workshop*, LNCS 1388, Orlando, USA, April 1998, pp. 604-622.

[14] Chiba S., Masuda T., "Designing an Extensible Distributed Language with Metalevel Architecture", in *Proc. of ECOOP'93*, LNCS 707, Springer-Verlag, Kaiserslautern, Germany, 1993, pp. 482-501.

[15] Chiba S., "A Metaobject Protocol for C++", in *Proc. of OOPSLA'95* (ACM Conference on Object-Oriented Programming, Systems, Languages and Applications), Austin (TX-USA), Oct. 1995, pp. 285-299.

[16] Powell, D., "Distributed Fault Tolerance: Lessons from Delta-4", IEEE Micro, Feb. 1994, pp. 36-47.

[17] Blanck Lisboa M. L., "A New Trend on the Development of Fault-Tolerant Applications: Software Meta-Level Architectures", *in Proc of the 1998 Int. Work. on Dependable Computing and its Application*, (DCIA'98) Johannesburg, South-Africa, Jan. 1998, pp. 148-157.

[18] Maffeis S., Schmidt D.C., "Constructing Reliable Distributed Communication Systems with Corba", in IEEE Communications Magazine, Vol. 14(2), Feb. 1997, 6p.