Reflective architectures
for reusable fault-tolerant software

M. Ancona
G. Dodero
V. Gianuzzi
DISI, Univ. Genova

A. Clematis
IMA,
CNR

M.L. Lisboa
Inst. Informatica,
UFRGS

Authors' addresses:

Massimo Ancona, Gabriella Dodero, Vittoria Gianuzzi
DISI, Università di Genova
Via L.B. Alberti, 4 - I 16132 Genova, Italy
Fax: + 39 10 353 8769
Email: ancona@xenon.dima.unige.it

Andrea Clematis
IMA, CNR - Via De Marini, 6
I 16149 Genova, Italy
Fax: + 39 10 647 5660
Email: clematis@ima.ge.cnr.it

Maria Lucia Lisboa
Instituto de Informatica, UFRGS
Porto Alegre, RS, Brazil
Email: llisboa@inf.ufrgs.br

Abstract

We describe an architecture supporting the definition of reusable and fault tolerant software. The architecture is reflective and consists on three levels, the application level, the monitoring meta level and the meta-meta level. Various types of reuse are applicable at each level, in accordance with requirements for fault tolerance arising from application needs.
1 Introduction

The object-oriented paradigm credits its success to reusability. The old macro-instruction concept as a reusable component attained perfection in this paradigm by gluing together the data and the actions to be executed over the data, namely, objects and their associated behaviors.

A second issue springs from the complexity of abstraction. The procedural abstraction, originating from the imperative model to divide the programs into small modules, has become more abstract due to the encapsulation concept. Experience has shown that it is not a trivial task to design object-oriented applications and it is even harder to design reusable components. Fault tolerance introduces an additional degree of complexity, since failures and resulting invalid computational states must also be taken into account by designers.

A third issue arises from the current pressure to produce reliable systems in a short term. Reuse of code enhances reliability because residual faults on software tend to decrease in the long term. On the other hand, the design of fault tolerant policies to be inserted in a given application is a very specific task, which takes into account application dependent issues. The reuse of such efforts at code level is usually not possible, and only general design principles can be adapted from one system to another.

The purpose of this paper is to show how reusable fault tolerant software can be obtained by means of computational reflection. Reflection is especially important in the design of extensible languages. However, such a powerful tool is widely applicable, and the field of fault tolerant software can benefit from it.

The paper is organized as follows: the next section recalls the basic definitions about software and system fault tolerance. Reflective architectures are then introduced, showing the relationships with previous architectural approaches for fault tolerance, and motivating the choice of the metaobject model for our architecture.

Section 4 details reuse aspects at each level in the proposed architecture, and provides specific examples of reusable fault tolerant software. Implementation of the proposed architecture on distributed systems requires defining suitable grouping mechanisms, dealt with in section 5.

2 Fault tolerant software

The aim of this section is to give a brief introduction to fault tolerance concepts, and commonly used software mechanisms for making systems and software fault tolerant. There are two levels at which software techniques can be applied to help systems increasing dependability: the first that we call system fault tolerance includes mechanisms capable of executing an application program in spite of transient or permanent hardware faults. The
management of software bugs is not considered. System fault tolerance can be obtained by replication or by state saving-restoring. In the latter case, state changes can be handled as atomic actions that are committed or aborted-restarted like transactions. In the former case, fault tolerance is achieved by means of parallel identical computations (replicas) running on different sites. If a site crash, the surviving computations can resume the lost computation. An additional goal of system fault tolerance is transparency, that is, the application programmer has little or no concern about replication and distribution of his/her programs.

The second category called software fault tolerance is specifically related to, and deals with, software faults: in this case operation restarting and replication of identical subsystems cannot solve the problem of bypassing software bugs persistently installed into the software. Successful approaches to such problems exploit diversity at code level: different versions of critical software parts are developed by different teams, using different designs and implementations, in order to minimize the probability of a coincident software bug. A comparison/selection mechanism, called acceptance test or voter, must be provided in order to discriminate nonfaulty results. The diverse software can be executed sequentially on the same machine, or in parallel on a distributed hardware.

Most of the algorithms intended to support system fault tolerance have been implemented as reusable tools (e.g. ISIS [Bir90]). They provide at the operating system level, or just above it, a set of services that can be invoked by applications, by linking to specific libraries. Specialization of requirements for fault tolerance inside each application is thus performed within groups of related processes.

Application specific features in software fault tolerance imply an even tighter coupling between mechanisms for fault tolerance and application code: there have been proposals for software fault tolerance that assume the use of new programming language constructs, such as the recovery block. Diversity is often achieved by developing software in different environments and languages: in fact, diversity and reuse may even appear as conflicting issues.

We have already remarked how complex can fault tolerant software be. Similar to what happens for any other intrinsically complex system, the definition of clean interfaces between application components is the key for their independent development and reuse. In the case of fault tolerant object-oriented software, interfaces isolate the behavior of an object from that of other objects and thus may prevent fault propagation. At the same time, the designer of the whole application need not be concerned with a "global" property of fault tolerance of the application; he/she may decompose such issue into that of making individual critical objects fault tolerant, and that of controlling interactions among them.

Transparent interfacing of fault tolerant and non fault tolerant objects is then the first step leading to reuse. This way, the best mechanism for achieving fault tolerance within an object can be independently designed from the rest of the application, once
the interfaces are kept stable. The second step lies in the possibility of "reasoning about objects behaviour". This feature, as it will be shown in the next section, is achieved by computational reflection.

3 Using Reflection for Fault Tolerance

Computational reflection is defined as the activity performed by a computational agent when doing computations about its own computation [Mae87, Yon87, Wat88]. Thus, a reflective architecture is a system which incorporates data structures representing itself. That representation of self makes it possible for the system to support actions on itself. As a result, a reflective object-oriented system may monitor the behavior of its components and relative computations, acquire methods from other objects dynamically and make addition/deletion or change the set of its own methods.

In a reflective architecture there is a reflective tower of computational domains \( D_i \). Each domain \( D_i \) is called the basic level of domain \( D_{i+1} \) that, in turn, is called the meta level. Each domain \( D_i \) is both a basic domain for the upper level domain \( D_{i+1} \) called the meta level, and a meta domain for the basic domain \( D_{i-1} \), excepted \( D_0 \) made of referents, which can be used only as a base level. A reflective architecture thus defines a layered system where the base level represents the application domain and the other levels, the meta levels, represent the system domain.

The reflective tower is analogous to previous work [Anc90] where a layered architecture is presented, for implementing fault tolerant concurrent software systems. Such an architecture postulates the existence of a separate level for recovery software, called the Recovery Meta Program. This recovery level monitors the execution of the application programs and implements the recovery algorithms, accessing operating system primitives for fault tolerance support, while the controlled application level is only required to specify the fault tolerance technique to be used, to indicate the critical regions, and to supply the redundant code through minor languages’ extensions.

The advantage given by the Recovery Meta Program architecture over specific systems for fault tolerant software lies in the transparency of interfacing between the application and the recovery levels. The two levels can be independently developed and tested, They can also be reused independently from one another. As an example, let us consider system fault tolerance implemented by replication of critical processes on different sites:

- at the application level, the user just defines code for all application processes (including non critical ones);
- at the recovery level, additional processes may monitor replication of critical application software and support their transparent interfacing within the system.

A reflective object-oriented model thus appears to be well suited for designing and implementing fault-tolerant software.
There are three basic approaches to computational reflection in class based languages [Fer89, Mae87, Mat91, Chi93]:

- The **metaclass model**, in which the class of an object plays the role of the metaobject controlling the execution of all its instance objects. This approach is considered the canonical one for its structural computational model, but it lacks of flexibility, in that all objects in a given class are controlled by their own metaclass.

- The **metaobject model**, in which each object may be controlled by a corresponding metaobject. This approach is more flexible since it specializes metaobject behaviour in accordance with the individual object, thus providing a good encapsulation.

- The **metacommunication model**, which is based on the reification of messages sent to objects running at the base level. Instead of sending a message \( ttM \) to object \( O \), \( M \) is reified into an object \( M_0 \) and the special message "send" is sent to it. This model is the most flexible one, but its disadvantage is that the system is almost always operating at meta level. This fact obstructs encapsulation of application objects.

These considerations have suggested us to consider the metaobject model for computational reflection as the most natural model upon which fault tolerance can be built. Each object \( O \) has a metaobject \( M_O \) which represents \( O \). To represent means that every object is the representation of something. Base level objects represent entities of the real word, while metaobjects represent other objects of the base level. The metaobject describes the behavior of the related object (i.e., how the object handles messages, changes its state and so on). A metaobject is an object which reflects the structural, and possibly also the computational aspect of a single object. Note that a metaobject is a meta level object, while the converse is not necessarily true: there could be multiple metaobjects representing a single object.

The metaobject model is based on a specific class **metaobject**, whose instances are objects, called **metaobjects**. Each metaobject \( M_O \) can control the behavior of a specific referent object \( O \) of the base level. The latter one belongs to another class, while \( M_O \) encodes information concerning to its referent object. Each message \( M \) sent to object \( O \) is intercepted and the meta message \( \text{dispatch-method}(O,M) \) is sent to metaobject \( M_O \). This feature preserves transparency of interfacing between those objects that are controlled by a corresponding metaobject, and those that are not. There is no way by which the sender object may know whether the message is intercepted by a metaobject or is directly delivered to the referent object. This message passing mechanism represents the key for transparent interfacing between base level objects and metaobjects, or between metaobjects. As we can view an object-oriented application as a collection of interacting objects, we can distribute the property of fault tolerance to each critical object [Cle95]. Furthermore, adopting a reflective architecture can organize activities relevant to fault tolerance within the meta level, without interfering with the structure of the remaining application domain objects. Following [Anc90] we propose three levels of execution supported by reflection:
• The application level is composed of referent objects operating at the $D_0$ base execution domain. Portions of code relevant to software fault tolerance, such as versions code and voter, should be defined in $D_0$.

• The monitoring level is composed by metaobjects operating at the upper execution domain $D_1$. This level provides the fault tolerance policies specifically designed for the application.

• The reflective tower regression is closed by the meta-meta level $D_2$. The meta-meta level contains the mechanisms needed to support policies of $D_1$. They may be operating system services up to primitive operations for distributed computations (e.g., Isis [Bir90]).

Another feature of object-oriented systems which is relevant to fault tolerance is encapsulation: by its use, unified management of replicas and versions can be supported. Due to the encapsulation property, if two objects $O_i$ and $O_k$ have the same protocol and the same external behavior, we can consider $O_i$ and $O_k$ as equivalents objects independently of their internals. Substituting $O_i$ any place $O_k$ is expected will cause no problem to the application, as far as the expected service is delivered. Given a set of equivalent objects $O_1, O_2, ..., O_n$ implementing the same semantics, each object $O_i$ might be either a version or a replica of $O_k$. Thus, different versions of an object can be implemented and later modified, without affecting the rest of the system.

4 Reuse in the reflective architecture

The three-layered architecture of the previous section can support various forms of reusable software, which we analyze in detail in the following.

The application level

The application level $D_0$ contains all referent objects, which may be critical (then associated to a metaobject in $D_1$) or non critical (therefore not associated to a corresponding metaobject). Reusability criteria for non critical objects are guaranteed by the transparency of interfacing to critical objects, and follow the same criteria as for any object-oriented software [Mey88, Boo94]. As an example, we may reuse

• classes and objects manipulating non critical systems portions, where a simple exception handling /error signaling mechanism is sufficient;

• classes and objects whose correctness has been formally proven, or thoroughly tested;

• classes, objects and functions that may be used for implementing certain policies for fault tolerance, providing application independent services. Among them we may consider supports for diversity, such as majority voters, procedures checking for the equivalence of different data structures which contain the same information, procedures for compacting process status (thus optimizing the amount of information to be checkpointed) and so on.
The monitoring level
The monitoring level $D_1$ is composed by metaobjects responsible for the policies for fault tolerance useful for the application. There are both reusable objects (i.e. those being reusable as such), and reusable abstractions, that must be specialized according to the application, providing an abstract specification of policies for fault tolerance.

Examples of reusable objects include:

- objects used for the manipulation of underlying hardware for purposes of fault tolerance, such as checkpointing services (when such mechanisms are not implementable as libraries, or when they are reused from similar architectures);

- objects used for interfacing with software systems, at the $D_2$ level, such as those capable of controlling allocation of replicas.

Reusable abstractions at this level follow the well known policies in the literature [Fer89, Car90, Bir90, Cre91, Chi93] and variations of such policies may be reused as well. The next section shall illustrate two of them.

The meta-meta level
At the meta-meta level $D_2$ we find mechanisms to support the monitoring level. By definition, mechanisms are intended to be reusable and they are usually provided as libraries of primitives.

Here reuse is exploited at the system level. It follows the view of *implementational reflection* considered in Silica [Rao91].

"*Implementational Reflection*. Reflection involves inspecting and/or manipulating the implementational structures of other systems used by a program."

The meta-meta level supports an *open-ended* view of a system architecture by interfacing levels $D_0$ and $D_1$ to other systems supporting distributed computations and mechanisms like object groups implementation, broadcasting primitives and so on. The meta-meta level $D_2$ interfaces the fault tolerant application composed by critical and non-critical objects with the fault tolerance system’s services. Thus it is possible, for instance, to replicate metaobject groups to achieve system fault tolerance.

$D_2$ provides reusability at the system level instead of programming language level. It frees $D_0$ from supporting mechanisms for controlling concurrency and process synchronization: they are used in $D_1$ and implemented in $D_2$. This offers the advantage of decoupling application software from problems related to concurrency and encapsulation [Bri87, Car90, Cre91].

On the other hand, this approach limits concurrency by excluding intra-object concurrency. But this is exactly what a reliable system requires: an encapsulated use of critical
operations for preventing the smuggling of inconsistent states through large parts of the system.

4.1 Examples

Let us now examine typical situations of system and software fault tolerance, in order to better understand the proposed architecture and its features for reusability.

Example 1: Coordinated Checkpointing

Let us examine first a typical situation for system fault tolerance: the problem of taking a checkpoint of a distributed computation, to tolerate transient failures. Each object must be capable of taking a local checkpoint on a stable storage. Moreover, these checkpoints must be taken in a coordinated way, in order to bring the distributed system to a consistent state, avoiding the so called "domino effect", that is, the uncontrolled roll-back of all the objects to the starting point. Several algorithms have been proposed to prevent the domino effect and to allow the correct resumption of the execution after a failure. One of the best known is the algorithm presented by Koo and Toueg [Koo87]. Taking a local checkpoint is a reflective operation, to be performed by each metaobject upon the corresponding object. The coordination algorithm is then executed by the metaobjects. Reusability can be achieved in two ways:

- support for checkpointing is often provided by specialized hardware: the servers of such devices are then reusable;

- the coordination algorithm for metaobjects can be reused (possibly by subtyping) from other applications providing such policy.

Note that in this example there is a communication among metaobjects. One metaobject sends messages to other metaobjects. When this happens, we have a computation at the meta-level only and the system is then reasoning about itself.

Example 2: N-Version Programming (NVP)

N-Version Programming is one of the best known policies for software fault tolerance. It is based on the assumption that it is most unlikely that different teams, independently developing a given portion of software, will introduce the same errors. Results yielded by various versions of the same software are then compared, in order to check if they agree. If they disagree, some version must have made an error, and the choice of what to yield as result is usually taken as a majority vote. Note however that voting is not always possible, and that there are several voting criteria, of which two-out-of-three majority is only the simplest. The problem of defining voters, or better adjudication functions, has been widely investigated in the literature, and a survey of them is given by Di Giandomenico and Strigini in [DiG90].

Let us assume that in our application there is an object Obj which we make fault tolerant by developing three versions respectively named Vrs1, Vrs2, Vrs3. To minimize the probability of hardware faults, as well as for software faults, the three versions must be executed on different processors, and to simplify the discussion let us assume that they are
not invoking additional services from other objects. The three versions must support the same external interface, that is the external specification of the behaviour of Obj. At the application level $D_0$ another object, User, is invoking a method of Obj. The metaobject associated to it, M.Obj, intercepts the invocation and is responsible for

- instantiating the three versions on different sites;
- invoking the corresponding method in each version and collecting the answers;
- communicating such multiple results to the M.Voter metaobject. The task of M.Voter is to select the relevant voting object between V1 and V2, and to dispatch results to/from such voter;
- answering to User with the correct result computed by the voting object.

![Diagram](image)

*Figure 1: N-Version Programming*

This situation is illustrated in Figure 1. Note that the metaobject M.Obj is conceptually associated with the single object Obj, which actually no longer exists in the system, having been replaced by its three versions. In this case the rule of associating a single object with one or more metaobjects is not violated. In this example, let us consider what kinds of reuse are possible:

- at level $D_0$ we substitute the non fault tolerant object Obj with the fault tolerant above described three-versioned subsystem, without affecting User, as well as other portions in the application software;
- again at level $D_0$, code for the voters V1, V2 can be reused from other fault tolerant subsystems;
- at level $D_1$, the metaobjects M.Obj and M.Voter can be reused from similar fault tolerant subsystems, with minor specializations achieved by subtyping.
5 Implementation of metaobjects on distributed systems

The mechanism shown in the above example can be suitably used only to implement the software fault tolerant mechanism of N-Version Programming. In fact, considering system fault tolerant aspects, that architecture can tolerate a site crash, if the crashed site is that of one version, but it does not consider what happens if the crashed site is where the metaobject M.0bj is executing. We can overcome this problem (as in any other similar situation where there is a centralized decision) by replicating metaobjects as well. The replicated metaobjects shall reside at the same site as the respective referent objects.

As explained by [Mat91] a metaobject is an object reflecting the structure and behavior of a single object. The converse is not necessarily true: there could be a set of metaobjects collectively controlling the behavior of a single (or of a set) of referent objects. This is necessary when a high degree of reliability is required: metaobjects themselves need to be fault tolerant. In this case, the behavior of a set of object is governed by the coordinated efforts of a group of metaobjects. The concept of (meta) object group is very important in distributed fault tolerant system, because it is a mechanism for encapsulating a distributed redundant computation into a single entity, that can be seen, from the outside, as a single non replicated or versioned base level or meta level object.

Thus, any distributed implementation of our architecture should be based on object groups. The object group has been widely studied both by the reliability and the object-oriented communities [Yon87, Mat91]. Object groups are dynamic entities. New groups can be created at runtime, an object can join an existing one or leave a group at any time, or after the completion of some event. Object groups require few manipulation primitives, as suggested by Chiba and Masuda [Chi93]: a primitive for group creation, destruction, a primitive for entering an object into a group, and for leaving a group. In our reflective architecture, these primitives are implemented at the meta-meta level $D_2$. The object group concept requires also a way for identifying groups and some form of message broadcasting between the participants to a group.

Recalling the last example given in section 4, Figure 2 illustrates the situation of the versioned object Obj, where each version Vrs1, Vrs2, Vrs3 is respectively monitored by M.0bj1, M.0bj2, M.0bj3. The voting system is the same as in Figure 1. Each one of three metaobjects monitors the related version, but only one out of them, called group leader, is responsible for collecting the invocations to Obj, to contact the other metaobjects and the voter, and, finally, to return to the caller the correct result. In Figure 2, the current leader is M.0bj1. In case of failure of the site where M.0bj1 is residing, the other metaobjects execute an election algorithm to replace the lost leader.

6 Conclusion

The paper has illustrated a reflective architecture upon which fault tolerant software can be developed. This architecture is based on a composition of three levels, and it has been
shown how different kinds of reuse can be applicable at each level:

- reuse of interfaces between fault tolerant and non fault tolerant objects, due to transparency preserved by the metaoject model for reflection;
- reuse of mechanisms for fault tolerance, through implementational reflection;
- reuse of policies from similar fault tolerant applications, by use of inheritance and subtyping.

Implementation of the proposed architecture on distributed systems must be based on groups of objects and groups of metaobjects, since centralising critical decisions makes tolerance to site crashes impossible.

References


