# USING REFLECTION FOR SOFTWARE FAULT TOLERANCE

**Shahrokh Jalilian, Tofig Kazimov, Fatemeh Salar**

Institute of Information Technology of ANAS, Baku, Azerbaijan
*sh.jalilian@yahoo.com*

## 1 Introduction and Background

### 1.1 Software Fault Tolerance

*Software Fault Tolerance* represents a major challenge to designers of modern computing systems, in particular, in the development of critical applications. The construction of fault tolerant systems is not a simple task; it requires the use of appropriate techniques during the whole software development cycle. In general, these techniques are based on the provision of redundancy, both for error detection and error recovery. However, the provision of software redundancy implies: a cost increase of the software development, and a complexity increase of the system, caused by the addition of redundant components [2]. *Software fault tolerance* is concerned with techniques necessary to enable a system to tolerate software faults, that is, faults in the design and construction of the software itself. Strigini presented a comprehensive survey of software fault tolerance issues in [15]. Some of the software mechanisms used to support fault tolerant applications include check pointing facilities and replicated servers. Such fault tolerant behaviors can be implemented either by error processing protocols in the underlying runtime systems, or by using object oriented methodologies, so making nonfunctional characteristics inheritable. All these approaches have advantages and drawbacks: if the error processing mechanisms are provided by the underlying system, then the transparency and separation of concerns can be achieved, but this lacks of flexibility. If fault tolerance behavior is supported by predefined libraries then programmers can write their own error processing mechanisms but transparency and separation of concerns can not be achieved. In object oriented systems, when using inheritance, separation of concerns is achieved but transparency is not totally covered, because some programming conventions are required [10].

### 1.2 Reflection

*Computational reflection* is defined as the activity performed by an agent when doing computations about its own computation. Thus, a reflective system incorporates data structures representing itself in order to support actions on itself. A reflective object oriented system may: Monitor the behavior of its components and computations; Dynamically acquire methods from other objects; Make additional/deletion or changes to the set of its own methods [10]. From a conceptual viewpoint, *reflection* can be defined as the property by which a component enables observation and control of its own structure and behavior from outside itself. This means that a reflective component provides a metamodel of itself, including structural and behavioral aspects, which can be handled by an external component. This information is used as an input to perform appropriate actions for implementing nonfunctional properties (concerning, for instance, fault tolerance or security strategies). The reflective systems that we consider are structured in two different levels of computation: the baselevel, which executes the application (functional) software, and the metalevel, responsible for the implementation of observation and control (nonfunctional) software. The metalevel software has a runtime view (the metamodel) of the behavior and structure of its baselevel. According to this view, the metalevel can take decisions and apply corresponding actions on base-level components. The mechanisms providing such a metamodel are the reflective mechanisms of the system [1]. The use of metalevel programming permits transparent separation of functional components from non-functional components in a system [10]. Two major reflective approaches have been pointed out in [9]: *metaobject* and *communication reification*. The metaobject approach consists in linking each base entity - also termed *referent* - with one or more meta entities - also termed meta-objects - reifying it. The communication approach consists in reifying only the base-entities

interactions into specific meta-entities. For the former approach we analyze the *metaclass* and *metaobjects* models, while for the latter we analyze the *message reification* and *channel reification* models. The characteristics of these reflective models have been evaluated in [9] based on three categories named, generic measures, meta-entities features and type of reflection (Structural/Behavioral). Their analysis shows that each considered model has its own peculiarity. These diversities make different model suitable for different tasks. The models belonging to the communication reification approach are more suitable than the others to develop distributed reflective systems. Moreover, the models belonging to the metaobject approach are more suitable than the others to handle structural reflection, and they permit to extend reflective systems dynamically changing its structure. However poor flexibility and lack of continuity respectively make the metaclass model and the message reification model not well suited for building up software fault tolerant applications. Entering in details, metaobject and channel reification are the winners of their respective categories. In respect to the other, these models are adaptable to any requirement. The other models have limitations; the metaclass model is limited by language requirements and the message reification model is limited by the lack of information continuity [9].

## 2 MOP-Based Fault Tolerant Systems

### 2.1 MetaObjects Protocols

In systems mixing the object-oriented approach and the above reflective concepts, a so-called Metaobject Protocol (MOP) handles the interactions between the base and the metalevel software [1]. We will refer to the baselevel objects as "functional objects" and to metalevel objects as "nonfunctional objects". While functional objects model entities in the real world, nonfunctional objects model properties of functional objects (to reflect this, nonfunctional classes may have names that correspond to properties, e.g. fault_tolerant_object) [4].
The notion of *protocol* relates here to the interaction between object (functional objects) and metaobject (nonfunctional objects). From a design viewpoint, one can distinguish four different processes in a reflective system to observe and control at the metalevel the features of the system's base-level. The ***reification*** process corresponds to the process of exhibiting to the metalevel the occurrence of base-level events. The ***introspection*** process provides means to the metalevel for retrieving base-level structural information. Finally, the ***intercession*** process enables the metalevel to act on the baselevel behavior (***behavioral intercession***) or structure (***structural intercession***). The term behavioral reflection will refer from here to both reification and behavioral intercession. Symmetrically, structural reflection will be used to designate both introspection and structural intercession mechanisms [1].

### 2.2 Related Work

In early works, various MOPs have been defined and used for the implementation of fault tolerant mechanisms at the metalevel. The MAUD[11] and GARF[12] architectures propose reflective mechanisms for intercepting baselevel events at the metalevel. The reflective capabilities defined in these MOPs are, however, limited to behavioral reflection. This limitation means that these systems are not able to handle structural aspects of baselevel entities that are essential, for instance, during checkpointing. MOP enabling both behavioral and structural reflection was used in FRIENDS [13]. This MOP supplies a metamodel expressed in terms of object method invocations and data containers defined for objects' states [1]. Several Authors addressed the transparent addition of fault tolerance features to a software via reflection by means of: applying channel reification for communication fault tolerance, employing reflective N-version programming and recovery blocks, employing reflective server replication and also employing reflective checkpointing in concurrent object oriented systems [14]. In the next section of the paper, some of the implementations of using reflection in software fault tolerance is described. Using reflection in object-oriented languages was invested by B.Smith in the environment of 3-Lisp. P.Maes proposed a metaobject approach to implementing reflective systems in the framework of object-oriented computing. The metaobject approach has

been used in many application areas: debugging, concurrent programming and distributed systems [9]. A successful example is the metaobject protocol in CLOS [14].

## 3 Some reflective approaches for implementing Software fault tolerance

An abstraction model (architecture) is suggested in [8] to describe common characteristics of the existing software fault tolerance schemes. Extended model in this architecture suggests a coherent framework for enforcing fault tolerance in an object-orientated fashion. The abstraction architecture helps the separation of objectlevel and metalevel descriptions. The controllers that control the execution of object variants are naturally implemented as meta-objects. Since a metaobject is also an object, it can be controlled by a meta-metaobject.

The aim of this section is to describe several approaches and programming styles that *Reflection* has been used in them. These approaches are considered for programming fault tolerance in distributed and embedded applications. Finally the advantages and limits of the reflection are considered in these systems. In distributed systems, administrative requirements, such as recoverability and persistence, can be implemented by replication of components. In this case, the object is the smallest grain of distribution and fault tolerance [2]. In [7], the methodology using an object oriented metalevel technique in designing of an extensible language for distributed computing has been proposed. For the use of this methodology, OpenC++ (which is a C++ variant including a simple metaobject protocol) is presented. To obtain a new functionality that fits the application, the programmer can easily extend the implementation within OpenC++ itself [7]. FRIENDS *(Flexible & Reusable Implementation Environment for your Next Dependable System)* is a metalevel architecture that its advantages have been advocated in [5]. The objective of *FRIENDS* was to investigate the use of object-oriented techniques and a reflective language approach for the development of fault tolerant distributed systems. The idea is to handle dependability mechanisms at a separate abstraction level and to bind them to application objects according to their needs. The objective of the FRIENDS system was to provide mechanisms for building fault tolerant applications in a more flexible way. Flexibility is obtained through the provision of object oriented libraries of metaobjects and also through the provision of subsystems on a microkernel platform. The FRIENDS system today is very dependent on the language used (Open C++) and on its homemade object-oriented distributed support [5]. The most important result in one of the reflective architectures (FRIENDS) is that the *runtime execution overhead* due to the use of a metaobject protocol, is negligible with respect to the runtime execution cost of the mechanisms implementing metafunctional properties within the metaobjects. FRIENDS system is an platform that enables Object-Oriented and metalevel programming to be used for implementing meta functional properties. FRIENDS system is a very suitable platform for experimenting with object orientation and metalevel programming in various directions. Some experiments have already been done using OpenC++ V2. *Performance Overheads* are one of major issues in metalevel techniques, but they are not critical in domains such as distributed computing. Since the overhead of OpenC++ is negligible in comparison with the implemented functionalities, the proposed approach in [7] is applicable to actual problems.

## 4 Advantages and Problem Statements

The main advantage of using reflection from a design viewpoint is the recognized ability to adjust mechanisms according to system needs. The separation of concerns promoted by the reflective approach has already shown significant effects on transparency for the application programmer, independence from the application software, reuse of core mechanisms and specialization for various contexts of usage [1]. Some advantages for application programmers are the: (i) Separation of concerns, that is, separate the concerns related to the application domain from those related to the implementation of fault-tolerant mechanisms; (ii) it promotes code reuse of fault-tolerance mechanisms, it allows application programmers to use the most adequate fault tolerance strategy for his implementation, and (iii) it provides a design that is

more adaptable, flexible and easier to extend than traditional designs for developing fault tolerant software[2]. In traditional systems, the integration of mechanisms within applications still raises several problems, mainly related to *flexibility*. We understand flexibility in the following way: ease of use and transparency of the mechanisms for the programmer; reusability of existing mechanisms to derive new ones. None of the solutions traditionally used manages to ensure all these properties at the same time. Reflection aims at providing a good balance among these properties. The main benefit of reflective approaches with respect to more conventional solutions is that they provide means to customize nonfunctional mechanisms, even providing facilities to change them during the operational life of the system without any modification of the executive layers [5]. Comparison of using reflection in embedded and distributed systems shows that overhead in distributed systems is more acceptable. The overhead of distributed systems due to the existence of communication delays between different nodes is negligible. Therefore using reflection in embedded systems where performance is important should be considered carefully.

## 5 Conclusion

There have been a few papers about experimental evaluation of software fault tolerance schemes in the context reflection. These experiments show that the runtime overhead of software fault tolerance is generally acceptable while making a clear, structured separation of concerns in both design and operation stages. Furthermore, when the communication cost is considered, the overhead imposed by reflective operation calls will not be of major concern. The overhead of distributed systems due to the existence of communication delays between different nodes is negligible. Therefore using reflection in embedded systems where performance is important should be considered carefully. Channel reification model is an appropriate model for reflective fault tolerant software with respect to other models. This model is adaptable to any requirement while the metaclass model is limited by languages requirements and the message reification model is limited by the lack of information continuity.

### References

[1] Juan Carlos Ruiz, Marc-Olivier Killijian, "Reflective Fault Tolerant Systems: From Experience to Challenges," IEEE TRANSACTIONS ON COMPUTERS, Vol.52, NO.2, February 2003.
[2] Luiz E.Buzato, "A Reflective Object-Oriented Architecture for Developing Fault-Tolerant Software," Institute Of Computing Scienec, University of Compainas, Brazil.
[3] Marc- Olivier Killijian and Jean-Charles Fabre, "Adaptive Fault Tolerant Systems: Reflective Design and Validation,", University of Compainas, Brazil.
[4] Walter Cazzola, Andrea Sosio, "Reflection and Object-Oriented Analysis," DISCO, pages95-106, November 1999.
[5] J.C. Fabre and T. Pe´rennou, "A Metaobject Architecture for Fault-Tolerant Distributed Systems: The FRIENDS Approach," IEEE Trans. Computers, special issue on dependability of computing systems, vol. 47, no. 1, pp. 78-95, Jan. 1998.
[6] J. Xu, B. Randell, and A.F. Zorzo, "Implementing Software Fault Tolerance in C++ and Open C++: An Object Oriented and Reflective Approach," *Proc. CADTED '96* , pp. 224-229. Beijing, China: Int'l Academic Publishing, 1996.
[7] Dr. Stéphane Ducasse, "Reflective Programming and Open Implementations," http://www.iam.unibe.ch/~ducasse/ University of Bern,2000/2001.
[8] Xu, B. Randell, C.M.F. Rubira & R.J. Stroud, "Towards an Object-Oriented Approach to Software Fault Tolerance," In: *Fault-Tolerant Parallel and Distributed Systems*, D.R. Avresky (Ed.), IEEE Computer Society Press 1994.
[9] Walter Cazzola, "Evaluation of Object- Oriented Reflective Models," DSI-University of Milano, July 4,1998.
[10] M.Ancona,W.Cazzola, "Channel Reification: a reflective approach to fault-tolerant software development," DISI-University of Genoa, October 9,1995.
[11] G. Agha et al., "A Linguistic Framework for Dynamic Composition of Dependability Protocols," Proc. Dependable Computing for Critical Applications 3, pp. 345-363, 1993.
[12] B. Garbinato, R. Guerraoui, and K. Mazouni, "Implementation of the GARF Replicated Objects Platform," Distributed Systems Eng. J., vol. 2, pp. 14-27, 1995.
[13] J.C. Fabre and T. Pe´rennou, "A Metaobject Architecture for Fault- Tolerant Distributed Systems: The FRIENDS Approach," IEEE Trans. Computers, special issue on dependability of computingsystems, vol. 47, no. 1, pp. 78-95, Jan. 1998.
[14] Jean-Charles Fabre, Vincent Nicomette, " Implementing Fault Tolerant Applications using Reflective Object-Oriented Programming," Department of Computing Science University of Newcastle upon Tyne,1998.
[15] L.Strigini, " Software Fault Tolerance," PDCSI 1st year Report, vol.2, Newcastle, 1990.