Adaptive Fault Tolerant Systems: Reflective Design and Validation

Marc-Olivier Killijian and Jean-Charles Fabre
LAAS-CNRS, 7, avenue du Colonel Roche, 31077 Toulouse Cedex 4, France
{marco.killijian, jean-charles.fabre}@laas.fr

Abstract

Reflection has been used with some success, since quite a few years now, for dealing with separation of concerns and transparency of fault-tolerance mechanisms for the application. Nevertheless, it has also shown some concerning the control of fine-grain information such as thread control or other deep aspects of the platform. We propose here the use of a new concept, called multi-level reflection, for firstly solving these issues, but also for introducing more adaptation into fault-tolerant reflective architectures. We also discuss some essential validation issues of reflective systems, which are still a challenge for future research.

1. Problem statement

Flexibility, reuse, and adaptation are becoming key aspects of today's large embedded systems (satellite systems, transport, automotive), and explain the increasing use of off-the-shelf components in the concerned industries. This trend raises challenges when considering the dependability of the resulting systems: How can we build dependable systems from components that don't specifically target dependability concerns? For these reasons, integrators are looking for sound and principled approaches that help them separate functional development from fault-tolerance concerns, within large projects, over long life-time.

Computational Reflection, an architectural paradigm that appeared in the late eighties, and related technologies such as aspect oriented programming, appear as very promising and powerful approaches to tackle this issue. Reflective architectures are centered on a key element, their meta-model, that insures the separation of concerns between the "base" system (here the system resulting from component integration) and the mechanisms (in our case, fault-tolerance) that are added to the base system. To be effective, this meta-model must take into account both the multi-component nature of the systems and the requirements of fault-tolerance that it should help implement. Within this work, we address this dual issue and propose a methodology to help design meta-models that specifically target the implementation of fault-tolerance into systems made of third party components. To some extent, we also address validation issues.

2. An introduction to multi-level reflection

A reflective system is basically structured around a representation of itself (its self-representation or meta-model) that is causally connected to the real system [1]. This meta-model divides the system into two distinct parts: a base-level where normal computation takes place, and a meta-level where the system computes about itself (meta-computation or meta-level software).

The design of a reflective system mainly consists in providing reflective mechanisms to establish meta-models. The reflective mechanisms provide observation and control features that can be divided into four classes:

- *reification mechanisms* by which the base-level exhibits information about its own computation;
- *introspection mechanisms* by which the meta-level can obtain (on-demand) structural information about the base-level;
- *behavioral intercession* leading the meta-level to control base-level computation;
- *structural intercession* enabling the meta-level to update base-level entities.

This kind of mechanisms is the corner stone of any reflective system or component. In object-oriented system, this is often provided by a so-called meta-object protocol (MOP), for which base-level entities are objects and meta-level entities are metaobjects.

In a reflective object-oriented application meta-objects populate the meta-level and use the meta-model to control the behavior of normal application objects (i.e. based-level object). The meta-model is structured around notions that are "constitutive" of the base level; i.e. these notions are common to all applications that share the same programming model.

The systems we are interested in are made of third-party components that are most often organized in a layered architecture: OS kernel, system libraries, compilers, virtual machines, middleware, etc. These

layers introduce different abstraction levels that each provide different sets of "constitutive" elements from which applications can be built to run on top of these levels. For instance, at the language level, the meta-model of an object-oriented application considered would typically contain entities and events such as "Class", "Method", "Instanciation", "Invocation", or "Attributes". On the other hand, at the OS-level, it would other contain entities, e.g. concerning memory paging, or task scheduling. In our case, an ideal meta-model should provide all the reflective features that are required to implement correctly and efficiently fault-tolerant mechanisms. In the meta-model, all the different entities from the various levels are represented and linked together (cf. Figure 1).

For the design of this first part of the meta-model, we had to study various fault-tolerance mechanisms, to extract key features that must be observed and controlled [2]. The second important aspect of the meta-model concerns the multi-level nature of the considered architectures, i.e the relationship between the various entities and information found within each level.

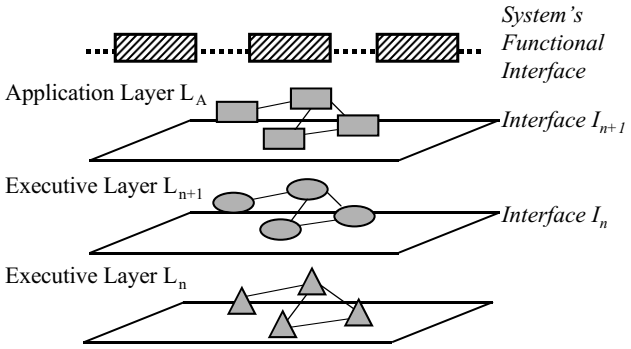

Figure 1: A Multilevel Reflective Architecture

In [3], we introduce the notion of mapping and projection to support the analysis of interlevel coupling from a reflective perspective. A mapping describes the various possible representations of a given entity at a given abstraction level *i* by entities available at a (lower) abstraction level *i-1* (cf. Figure 2).

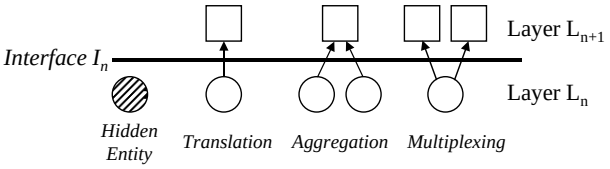

Figure 2: Simple Mapping Taxonomy

A projection is the transitive closure of mapping relations that maps a top-level entity to lower level entities (useful for state handling). Reverse projections map low-level entities to higher level ones (useful for defining error confinement regions). Projections allow the tracing of high level requirements regarding fault-tolerance (in term of data-consistency, expected QoS, performance) to lower-level concerns (platform state, communication management). Reverse-projection permits the mapping of low-level fault-assumptions to their impact on the high-level functional entities of the system (cf. Figure 3).

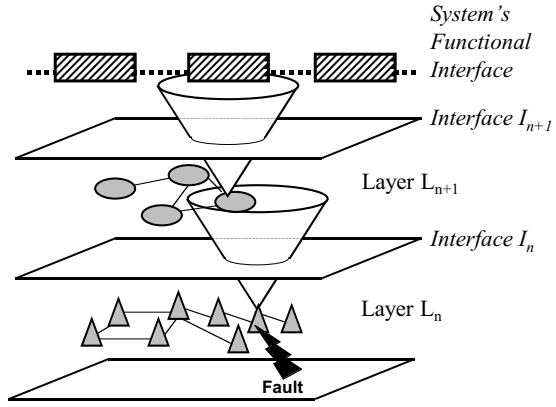

Figure 3: Error Detection across system layers

3. Adaptation using multi-level reflection

A multi-level relective architecture is an ideal platform for studying adaptation of fault-tolerant strategies. Firstly, all the necessary information is available for implementing a lot of different strategies. Secondly, this information can be retrieved at various levels: at low level for performance and at high level for richer semantics.

Nevertheless, beyond the implementation of various and possibly adaptive fault-tolerance mechanisms, probably the most interesting aspect of ML reflective architectures concerning adaptation is the capacity of changing the multi-level metamodel accordingly to the needs of the mechanisms. On one hand this possibility can be seen as a workaround for performance degradation due to too many indirections in the platform implementation (imagine the performance of an OS reifying to the FT mechanism every single event!). But on the other hand, this is also a great opportunity for FT programmers to tailor the architecture to their very needs. The degree of reflection of the platform can be fine-grain controlled. A FT programmer can develop, for instance, a mechanism that adapts itself to the load of the system, using heavy pessimistic protocols when the necessary resources are available, and switching to lightweight protocols when the system is overloaded.

The mechanisms used to tailor the metamodel in a multi-level reflective architecture are called the *meta-filters*. They filter the meta-information available at the meta-level. It is worth noting that, these meta-filters are conceptual filters but they don't have to be implemented as filters. We are currently working on the

implementation of several meta-filters: either based on simple code parsers or, for performance optimization, based on the use of code injection

4. Validation of Reflective Systems

While the design of reflective systems mainly focuses on how to provide separation of concerns, the validation of their implementation with respect to the specification must be addressed. A global strategy for the verification of reflective architectures corresponds to the incremental verification of:

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

Phases 1, 3 and 4, which are highly dependent on the particular functional and non-functional mechanisms implemented in the target reflective architecture, and can be addresses using conventional testing and fault injection techniques. Regarding phase 2, the challenge is to propose a test strategy generic enough to be used for different architectures based on similar reflective mechanisms, and thus independently of any particular implementation. This objective raises some fundamental problems:

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

Most of these issues are still open, as, to the best of our knowledge, very few work has been done in the field of validating reflective platforms. In our work reported in [2], we defined a testing strategy for reflective mechanisms implemented with a MOP (cf. Section 5)

5. Overview of a Test Strategy for a MOP

This strategy identifies four different testing levels and for each level it characterizes the test objectives and the required test environment. The test objectives are focused on the verification of the properties expected from each reflective mechanism in order to be confident in its use. Obviously, the strategy instantiation for the testing of a target MOP will have to comply with the MOP implementation.

The activation of the reflective mechanisms is based on an interaction channel between base and meta-level entities. Hence, exercising (and thus testing) these mechanisms requires a high level of confidence in this interaction channel. This confidence may be obtained by testing the process followed to establish the interconnection. This issue relates to this very first testing level (called *Testing level 0*) that is highly dependent on the implementation and thus, can vary a lot from one implementation to another. As a result, our strategy cannot provide general guidelines for this level although we assume that it is successfully achieved in the first place. Further details regarding this implementation related concerns can be found in [4].

Once this link is correctly established, the reflective mechanisms (of the MOP) are exercised following a test order defined according to the dependencies existing among these mechanisms. These dependencies are exploited in order to define an incremental test strategy that reduces the testing effort to be spent. In accordance with this goal, we proposed the following order to conduct testing of the four reflective mechanisms:

- *Testing level 1*. Reification (behavioral observation);
- *Testing level 2*. Behavioral intercession;
- *Testing level 3*. Introspection (structural observation);
- *Testing level 4*. Structural intercession.

The relevance of this order was validated on a fault tolerant reflective platform [2]. For each level, we give the test objectives, that is, the requirements to be met by (and thus tested on) the implementation of the target reflective mechanisms. Then, the necessary test environment is defined in terms of the entities participating in the test experiments (the server object and its metaobject, the test driver and the oracle objects), the interactions among these entities, and the conformance checks to be performed in order to decide whether or not the MOP passes the tests. The role of the oracle object is to verify that the test executions meet the requirements imposed by the MOP specification. The test driver object manages the test experiments: (i) it acts as a client object to exercise the MOP by supplying it with test case input values and, (ii) it provides the oracle object with (part of the) data that the oracle procedure uses to determine correctness during test execution.

6. Conclusion

Reflection is today a well-known paradigm that has been successfully used to address non-functional concepts in system architectures. In particular, security and fault tolerance have benefited from this concept as demonstrated by several projects and prototypes worldwide (e.g. [5, 6, 7]). Our previous research in the field was a contribution to the development of fault- and intrusion- tolerant systems using reflective languages. The main problem we identified was the limited meta-information available at a given level regarding the above or underlying layers of the system. This was the main motivation for the introduction of multi-level reflection.

The basic concepts identified at this stage enable meta-level software to be based on a clear understanding of the entities-relations at all levels in a computer system and of their links through several software layers. We also advocate in this approach specialized meta-models that can be defined for targeting a given non-functional requirement. Several notions such as mapping and projections provide means to draw error confinement areas, and identify state information through all system layers for error recovery.

The use of reflective system in real applications remains questionable as the validation aspects of this kind of architecture were not addressed by many works, presently. Design and validation efforts must be combined for dependable adaptive systems based on reflective architectures and components.

We briefly reported in this paper on some experience we have for the validation of reflective platform, essentially based on simple metaobject protocols. Clearly, this is still on open research topic. Reflection concepts are today applied to various components at all system layers of a complex architecture. In addition, we believe that the multi-level reflection model is a good approach for building adaptive systems, but more work is needed, including from a validation viewpoint.

Adaptive fault tolerance *per se* poses new design, implementation and validation problems, that must be solved as large system today have long life time, and must evolve in both the functional and the non-functional domain.

7. References

[1] P. Maes, "Concepts and Experiments in Computational Reflection," presented at *Conference on Object-Oriented Programming Systems, Languages, and Applications* (OOPSLA), Orlando, Florida, 1987.

[2] J.-C. Ruiz-García, M.-O. Killijian, J.-C. Fabre, and P. Thévenod-Fosse, "Reflective Fault-Tolerant Systems: From Experience to Challenges", *IEEE Transactions on Computers*, Special Issue on Reliable Distributed Systems, Vol. 52, n°2, Féb 2003.

[3] F. Taïani, J.-C. Fabre, and M.-O. Killijian, "Principles of Multi-Level Reflection for Fault-Tolerant Architectures," presented at the *IEEE Pacific Rim International Symposium on Dependable Computing* (PRDC 2002), Tsukuba (Japan), 2002.

[4] J.C.Ruiz-Garcia , J.C.Fabre , P.Thevenod-Fosse,"Testing Metaobject Protocols Generated by Open Compilers for Safety-Critical Systems", 3rd *International ACM/AITO Conference REFLECTION 2001*, (Lecture Notes in Computer Science 2192, Eds. A.Yonezawa, S.Matsuoka, Springer, ISBN 3-540-42618-3, 2001) Kyoto (Japan), pp.134-152, Sep. 2001.

[5] Pérennou, T. and J.-C. Fabre, "A Metaobject Architecture for Fault-Tolerant Distributed Systems : the FRIENDS Approach". *IEEE Trans. on Computer*, Special Issue on Dependability of Computing Systems, 1998. Vol.47, p. 78-95.

[6] Garbinato, B., R. Guerraoui, and K.R. Mazouni, "Implementation of the GARF Replicated Objects Platform". *Distributed Systems Engineering Journal*, 1995. **2**(1): p. 14-27.

[7] Agha, G., *et al.* "A Linguistic Framework for Dynamic Composition of Dependability Protocols". in *the IFIP Conference on Dependable Computing for Critical Applications (DCCA-3)*. 1992. Palermo (Sicily), Italy: Elsevier. p. 197-207.