A Multi-Level Meta-Object Protocol for Fault-Tolerance in Complex Architectures

François Taïani(*), Jean-Charles Fabre, Marc-Olivier Killijian
LAAS-CNRS ((*): Now at Lancaster University)

DSN'2005, The International Conference on Dependable Systems and Networks, Yokohama, Japan, June 28 - July 1, 2005
Motivating Example: Replication & Multithreading

**Goal:** Transparent replication of a CORBA server
- multi-layer: POSIX (OS) + CORBA (middleware)
- multithreaded: concurrent processing of requests
- thread pool: upper limit on concurrency

**Problem 1:** state capture / restoration
- application state
- middleware + OS state
Motivating Example: Replication & Multithreading

- **Goal:** Transparent replication of a CORBA server
  - multi-layer: POSIX (OS) + CORBA (middleware)
  - multithreaded: concurrent processing of requests
  - thread pool: upper limit on concurrency

- **Problem 1:** state capture / restoration
  - application state
  - middleware + OS state

- **Problem 2:** control of non-determinism
  - assumption: multi-threading *only* source of non-determinism
  - how to replicate non-deterministic mutex decisions?
Enforcing Determinism: OS Only

- The same lock allocation can be enforced on all replicas.
  - 😊 All replicas reach the same state.
  - 😞 Only a small subset of the lock allocations impacts determinism.

- Replication of every non-deterministic decision is highly inefficient.

Up to 203 sync. operations per request in middleware (ORBacus) [TAO: 52, omniORB: 64]
Only 3 of the synch. operations made by the middleware need to be replicated (ORBacus).

Smart Multi-Level Reflection

- With middleware and application semantics:
  - OS-level actions can be given a higher level semantic.
  - This semantic allows optimal use of OS level reflection.

Combining information obtained at different levels greatly increases the efficiency of crosscutting mechanisms.

(Here: only 1.5% of MD synch. activity actually needs to be replicated.)
The Vision

meta-level ⇐reflection⇒ base-level

meta-interaces

family of mechanisms

fault-tolerance

MOP (Meta-Object Protocol)

application

middleware

OS
The Problem

How to design & implement such a meta-object protocol?
Outline

- Motivating Example: Reflection and Replication
- A New Multi-Level MOP: Concepts & Design
- Practical Application: CORBA & Linux
Implementing Multi-Level Reflection

**Goal:** To provide a multi-reflective framework for the fault-tolerance of complex, non-reflective industrial platforms

**Challenges:**
- **Requirements:** *What* kind of information is needed for fault tolerant mechanisms? *Where* should this information be found?
- **Design:** How to design a multi-level meta-object protocol that supports multi-level reflection?
- **Instrumentation:** How to instrument an industrial, non-reflective platform in a non-invasive, transparent way?
Implementing Multi-Level Reflection

■ **Goal**: To provide a multi-reflective framework for the fault-tolerance of complex, non-reflective industrial platforms

■ **Challenges**:

  ➔ **Requirements**: *What* kind of information is needed for fault tolerant mechanisms? *Where* should this information be found?

  ➔ **Design**: How to design a multi-level meta-object protocol that supports multi-level reflection?

  ➔ **Instrumentation**: How to instrument an industrial, non-reflective platform in a non-invasive, transparent way?
Requirements

- Meta-interface for non-determinism [DSN-2003]

```java
interface MetaRequestLifecycle {
    /** Communication **/
    requestHasBeenReceived (RequestID);
    replyHasBeenSent (RequestID);
    /** Control Path **/
    requestBeforeApplication (RequestID);
    requestAfterApplication (RequestID);
    /** Synchronisation **/
    requestBeforeContentionPoint (RequestID, RequestContentionPoint);
    requestAfterContentionPoint (RequestID, RequestContentionPoint);
}
```
Requirements

- Multi-level nature of the meta-interface
Implementing Multi-Level Reflection

- **Goal**: To provide a multi-reflective framework for the fault-tolerance of complex, non-reflective industrial platforms

- **Challenges**:
  - **Requirements**: What kind of information is needed for fault tolerant mechanisms? Where should this information be found?
  - **Design**: How to design a multi-level meta-object protocol that supports multi-level reflection?
  - **Instrumentation**: How to instrument an industrial, non-reflective platform in a non-invasive, transparent way?
Semantics and Architecture

Motivating Example: middleware non-determinism

- request contention points (mutex operations) must be intercepted at OS level
- but not all mutex operations (otherwise highly inefficient)
- question: How to distinguish between mutexes that are relevant and those that are not?

Proposal: use of semantic context

- We need to understand the purpose of OS level mutex operations in the more general context of the whole system activity

Approach: to trace the computation process that results in a low level OS operation being called
To **trace** semantic contexts, a mechanism is needed to **transport** information between different abstraction levels (software layers).

A mechanism encountered in **plants**: in periods of droughts, the root system communicates with the foliage using dedicated chemical substances called **phytohormones**.

- **Phytohormones** travel through the **sap**.
- Design based on this metaphor.
  - Sap = **threads**
  - Phytohormones = **metamarkers**

---

**Meta-markers**
Inter-Level Communication with Meta-Markers

dormant meta-marker is attached to thread

meta-marker gets activated and modifies low level system behaviour

thread execution path

interception

higher level

meta-marker remains transparent

lower level

meta-level

base level
Using Meta-Markers for MOP Design

- Meta-markers can be used to design a multi-level MOP
- Example: synchronisation facet for middleware determinism

```java
interface MetaRequestLifecycle {
    ...
    /** Synchronisation **/
    requestBeforeContentionPoint (RequestID, RequestContentionPoint);
    requestAfterContentionPoint (RequestID, RequestContentionPoint);
}
```

- Two issues to be solved by meta-markers:
  - **P1**: the global semantic context of mutex creation must be captured by meta-markers
  - **P2**: meta-markers must insure a correct instrumentation of the selected mutexes
Capturing Semantics

Problem P1 is solved by source code annotation of semantic joint points:

```c
init_and_run_middleware(..) {
  MutexesAreRelevant metaMarker();

  metaMarker.attachToThread();
  init_request_queue(..);
  metaMarker.detachFromThread();

  init_some_refcount_object(..);
  ...
  run_ORB();
}
```
Meta-Markers as Meta-Mutex Factories

Meta-markers create new mutexes and attach them to a meta-mutex. A new mutex creation is intercepted, and newly created mutexes are released into the OS among other non-instrumented mutexes.
Back to the Meta-Interface

```java
interface MetaRequestLifecycle {

    /** Communication **/
    requestHasBeenReceived(RequestID);
    replyHasBeenSent(RequestID);

    /** Control Path **/
    requestBeforeApplication(RequestID);
    requestAfterApplication(RequestID);

    /** Synchronisation **/
    requestBeforeContentionPoint(RequestID, RequestContentionPoint);
    requestAfterContentionPoint(RequestID, RequestContentionPoint);
}
```

meta-markers to instrument appropriate mutexes
meta-markers to instrument appropriate sockets
meta-markers to transport request IDs
Implementing Multi-Level Reflection

- **Goal**: To provide a multi-reflective framework for the fault-tolerance of complex, non-reflective industrial platforms

- **Challenges**:
  - **Requirements**: What kind of information is needed for fault tolerant mechanisms? Where should this information be found?
  - **Design**: How to design a multi-level meta-object protocol that supports multi-level reflection?
  - **Instrumentation**: How to instrument an industrial, non-reflective platform in a non-invasive, transparent way?
Implementation

- **Multilevel interception framework**
  to control non-determinism; 8000 LoC C++; based on CORBA and POSIX only; platform independent.
Case Study: Orbacus

- Behavioural analysis: a reverse engineering tool dedicated to complex multi-layer systems

- This analysis indicates where to annotate the source code

- Instrumentation of Orbacus
  35 lines added
  → 0.02% of original code!
  (> 100,000 LoC)

→ Very low intrusiveness
  0.02% of original code!

→ Highly efficient: number of interceptions ÷ 70

- Graphical representation of class, thread creation, object creation, method call, request before application, request after application.
Conclusion

- **Tension** between comprehensive and adaptable fault-tolerance, and the multi-component and multi-layered nature of modern complex software systems.

- Our proposal to solve this conflict: 
  **Multi-Level Reflection**: 
  - Combines reflective capabilities found in lower and higher levels in a global system overview.

- MLR supported by a **multi-level MOP** based on:
  - semantic contexts
  - meta-markers

- Outlook: **Aspect Orientation**
  - Deep Aspects
  - Make aspects aware of software “thickness”
Any Questions?
Motivation

- Increasingly complex Computer systems (COTS / Layers) are used for increasingly critical applications.
- Most COTS have **not** been built with dependability in mind.
- Dependability is a system-wide multi-level issue.

<table>
<thead>
<tr>
<th>Fault-tolerance &quot;patches&quot;</th>
</tr>
</thead>
<tbody>
<tr>
<td>ad-hoc inter-level coordination</td>
</tr>
<tr>
<td>ad-hoc connection original code ↔ FT patches</td>
</tr>
</tbody>
</table>

⇒ How to add **fault-tolerance** to **complex multi-layered software systems** in a **transparent** and **disciplined** way?
Rationale behind Multi-Level Reflection

- Complex systems contain heterogeneous abstraction levels.
  ⇒ Available *(meta)-information* is heterogeneous.

- Higher levels:
  - 😊 Rich semantics
  - 😞 But they lack information / control.

- Lower levels:
  - 😊 Complete Information / control.
  - 😞 But lacking semantics

- Complementary roles: lower level information & control needs to be enriched with higher level semantics
What is Reflection?

"the ability of a system to think and act about itself"

- meta-level
- observation
- meta-interfaces

Fault-tolerance

- control
- meta-model "generic connector"

Base-level

Original system

Separating fault-tolerance from functional concerns
What are Meta-Object Protocols?

A particular way of organising a reflective system

meta-objects

MOP

base objects
Reflection & Fault Tolerance

- Reflection has been used to add FT to complex systems but:
  - Only one level of abstraction at a time considered so far.

Single-level Reflection $\rightarrow$ Limited Fault-Tolerance

OS

middleware
Motivating Example: Replication & Multithreading

- **Goal:** Transparent replication of a CORBA server
  - multi-layer: POSIX (OS) + CORBA (middleware)
  - multithreaded: **concurrent** processing of requests
  - thread pool: **upper limit** on concurrency
Requirements

- Example: CORBA Middleware Determinism
“Semantic joint points”: source code location where global purpose becomes apparent

The global purpose becomes apparent when backtracking the computation process that causes the low level calls

Two low level calls with different semantics

 pthread_mutex_init() (mutex creation)

needs not to be replicated

must be replicated

new RefCountObject

new RequestQueue

OS

middleware
Implementing Multi-Level Reflection

**Goal:** To provide a multi-reflective framework for the fault-tolerance of complex, non-reflective industrial platforms

**Challenges:**

- **Requirements:** What kind of information is needed for fault tolerant mechanisms? Where should this information be found?
- **Design:** How to design a multi-level meta-object protocol that supports multi-level reflection?
- **Instrumentation:** How to instrument an industrial, non-reflective platform in a non-invasive, transparent way?