Adaptive Fault Tolerant Systems: Reflective Design and Validation

Marc-Olivier Killijian

Dependable Computing and Fault Tolerance Research Group – Toulouse - France
Motivations

- Provide a framework for FT developers
  - Open
  - Flexible
  - Dependability of both embedded and large scale distributed systems
  - Adaptation of fault tolerance strategies to environmental conditions and evolutions

- Validate this framework
  - Test
  - Fault-injection
History

- Reflection for Dependability
  1. Friends v1 - off-the-shelf MOP
    - Limits: static MO, inheritance, etc.
  2. Friends v2 - ad-hoc MOP / CT reflection
  3. Multi-Level Reflection

- Validation of the platform
  - Test of MOP based architectures
  - Fault-injection and failure modes analysis
Outline

• Reflection for Dependability
  1. Friends v1 - off-the-shelf MOP
     – Limits: static MO, inheritance, etc.
  2. Friends v2 - ad-hoc MOP / CT reflection
  3. Multi-Level Reflection
• Validation of the platform
  • Test of MOP based architectures
  • Fault-injection and failure modes analysis
Why Reflection?

• Separation of concerns
  – Non functional requirements
  – Applications

• Adaptation
  – Selection of mechanisms w.r.t. needs
  – Changing strategies dynamically

• Portability/Reuse
  – Reflective platform (relates to adaptation)
  – Meta-level software (mechanisms)
Overall Philosophy

- **Metalevel** (metaobjects and policies)
- **Baselevel** (application objects)

One-to-one  
One-to-many  
Dynamic association

- **Wrappers**
- **API**
- **Middleware**
- **OS / kernel**
- **Comm. protocols**
- **Hardware platform**

Testing strategies  
Mechanisms Development  
Assumptions  
Failure Modes analysis  
SWIFI  
Fault model
Friends v2 : A MOP on Corba

- MOP design
  Identify information to be reified and controlled

- MOP implementation
  Compile-time reflection
  Using CORBA facilities

- Prototype for illustration
  Architecture and basic services
  Fault tolerance mechanisms
  Preliminary performance analysis
Necessary information:
integrated mechanism example

Client ➔ Server ➔ Replica

1-invocation ➔ 4-send checkpoint ➔ 5-apply checkpoint
2-process invocation ➔ 3-obtain checkpoint
7-reply ➔ 6-return
Necessary information: metaobjects example

1-invocation
2-process invocation
3-obtain checkpoint
4-send checkpoint
5-apply checkpoint
6-return
7-reply

Client → Stub → Server → Replica

Meta Stub → MO Server → MO Replica

Observation
invocations
state capture

Control
method calls
state restoration
Which protocol?

**Observation**

- Interception
  - Creation
  - Destruction
  - Invocations (In and out)
- State capture
- Links control
  - Object/metaobject
  - Clients/servers

**Control**

- Activation
  - Creation
  - Destruction
  - Invocations
- State restoration
- Links control
  - Object/metaobject
  - Clients/servers

**Reification**

**Intercession**

**Meta Object**

**Object**
Protocol definition

Protocol and interfaces specific to a mechanism
Using Open Compiler

```java
metaClass TranslateMethodCall (ReifiedInfo) {
    ..
    return NewCode;
}
```

`o.foo();`
Results

• A method for designing a MOP
  – Analysis of mechanisms’ needs ⇄ MOP features

• Metaobject protocol for fault tolerance
  – Transparent and dynamic association
  – Automatic handling of internal state (full/partial)
    • Portable serialization [OOPSLA’02]
  – Smart stubs delegate adaptation to meta-stubs
  – CORBA compliant (black-box)
  – Some programming conventions
Lessons Learnt

- Generic MOP
  - No assumption on low layers
  - Based on CORBA features

- With a platform «black-box»
  - Language dependent
  - Limitations
    - external state
    - determinism

- “Open” platform (ORB, OS and language)
  - Additions of new features to the MOP
  - Optimization of reflective mechanisms
  - Language level reflection still necessary
Limits to be addressed

• Behavioral issues
  – Concurrency models: Middleware level
  – Threading and synchronization: Middleware/OS level
  – Communication in progress: Middleware/OS level

• Structural/State issue
  – Site-independent internal state: Open Languages
  – Site-dependent internal state:
    • Problems: Identification, handling
    • Available means: Syscall interception, Journals and replay monitors
  – External state
    • Middleware level
    • OS level

Concept of multilevel reflection
Which protocol?

Observation

- Interception
  - Creation
  - Destruction
  - Invocations (In and out)
  - Threading
  - Synchronization
  - Communication

- State capture
  - Internal objects
  - Site-dependent objects
  - External objects (MW+OS)

- Links control
  - Object/metaobject
  - Clients/servers

Reification

- Activation/control
  - Creation
  - Destruction
  - Invocations
  - Threading
  - Synchronization
  - Communication

- State restoration
  - Internal objects
  - Site-dependent objects
  - External objects (MW+OS)

- Links control
  - Object/metaobject
  - Clients/servers

Control

Intercession

Meta Object

Object
Which Platform?

Universal VM for Distributed Objects

Middleware
Language Support
OS
Hardware

Fault-Tolerance
Which Platform?

This one?
But difference between OS/MW … LS/MW?

Universal VM for Distributed Objects

Middleware
Language Support
OS
Hardware

Fault-Tolerance

C
S
Which Platform?

Or this one?

Universal VM for Distributed Objects

Middleware
Language Support
OS
Hardware

Fault-Tolerance
Which Middleware?

FT needs to be aware of everything (potentially)

Universal VM for Distributed Objects

Under-ware

Hardware

Fault-Tolerance
Which Middleware?

FT needs to be aware of everything (potentially) but how?

Reflective languages …

Reflective middleware …

Reflective OS …

A lot of different concepts to manipulate
Multi-level Reflection

- mono-level meta-models
- aggregation
- multi-level meta-model

Self-contained, integrated, meta-interface

Fault-Tolerance
Multilevel Reflection

• Apply reflection to a complete platform
  – Application, Middleware, Operating System

• Consistent view of the internal entities/concepts
  – Transactions, stable storage, assumptions
  – Memory, data, code
  – Objects, methods, invocations, servers, proxies
  – Threads, pipes, files
  – Context switches, interrupts

• Define metainterfaces and navigation tools
  – Which metainterface (one per level? Generic?)
  – Consistency ➔ metamodel
Different Aspects

- **Intra-level information**
  - Necessary for FT
  - Efficiency (lowest possible? Same concepts at ≠ levels?)

- **Inter-level information**
  - ML management (inter-level coupling)
  - Adaptation
  - Concepts/levels navigation

Self-contained, integrated, meta-interface
Requirements of FT-Mechanisms?

• Non determinism of scheduling/execution time
  ⇒ Interlevel interactions mostly asynchronous

Trend: Leverage know-how on FT asynch. distributed sys.
  ⇒ Causality tracking/ monitoring of non-determinism is needed.
  ⇒ State capture/ recovery at appropriate granularity is needed.
  ⇒ … (?)
Inter-Level Coupling

- A Level = 1..n COTS = A set of interfaces =
  - Concepts
  - Primitives / base entities (keywords, syscalls, data types, …)
  - Rules on how to use them

- (concepts, base entities, rules) = programming model
  - Very broad notion (includes programming languages)
  - Self contained

- Base entities “a-tomic” within that programming model
  - Can’t be split in smaller entities within the programming model.
  - Implemented by more elementary entities within the component.
  - Implementation is internal ⇒ hidden to component user.
Inter-Level Coupling (II)

- **CORBA**: Location transparent object method invocation
- A CORBA request = aggregation
  - Communication “medium” (pipes, sockets, …)
  - Local control flow (POSIX threads, Java threads, LWP, …)
  ⇒ Global control flow abstraction
Inter-Level Coupling

- Within a COTS:
  - Coupling between emerging entities of next upper level and implementation entities of lower levels

- Structural coupling relationships (“abstraction mappings”)
  - translation / aggregation / multiplexing / hiding

- Dynamic coupling relationships (“interactions”)
  - creation / binding / destruction / observation / modification
Extracting Coupling in CORBA

Correlation level

Client

Server

Mutex

Socket

Thread

Signal

Observation level

Appli.

Mw.
Extracting Coupling in CORBA

- Behavioral model of connection oriented Berkeley sockets as seen by the middleware programmer.
Extracting Coupling in CORBA

Dynamic coupling between CORBA invocations and the socket API
FT + Inter-Level Coupling

- Top-down observation & control
  - State capture
  - Monitoring of non-determinism

System's Functional Interface

Application Layer \( L_A \)

Executive Layer \( L_{n+1} \)

Executive Layer \( L_n \)

Abstraction Level \( \text{Lev}_{n+1} \)

Abstraction Level \( \text{Lev}_n \)

Abstraction Level \( \text{Lev}_{n-1} \)
FT + Inter-Level Coupling

- Bottom-up observation & control
  - Fault propagation analysis / confinement
  - Rollback propagation / state recovery

System's Functional Interface

Application Layer $L_A$
Executive Layer $L_{n+1}$
Executive Layer $L_n$

Abstraction Level $Lev_{n+1}$
Abstraction Level $Lev_n$
Abstraction Level $Lev_{n-1}$
Meta-filters

• All the information is not always necessary
  – Specific mechanisms need specific info
  – Mechanisms can change over time
• Need a way to dynamically filter
  – Efficiency
    • Don’t reify unnecessary things
    • Have hooks ready but passified + subscriptions
• Meta-filters implementation
  – Simple boolean matrices
  – Code-injection techniques
Current & Future Work on MLR

• Still some work on ORB/OS analysis
• Implementation *a la carte* : several « flavours »
  – Radical style ➔ full metamodel from scratch or based on modified open-source components
  – Middle-Way
    based on available reflective components + wrappers
  – EZ way
    wrapped COTS ➔ limited metamodel
• Evaluate the benefits on mechanisms
  – Efficiency /ad-hoc /language level reflection
  – Evolution of non-funtionnal requirements/asumptions
  – Environmental evolution
• **Validation**
  – Rigourous testing stategies for reflective/adaptive systems
  – Characterization by various ad-hoc fault injection techniques
Adaptive Fault Tolerant Systems
Part II- Testing Reflective Systems

Reflection’00 - DSN’01- IEEE ToC 03
Ruiz, Fabre, Thevenod, Killijian

Dependable Computing and Fault Tolerance
Research Group – Toulouse - France
Motivations for testing MOPs

• In reflective architectures
  – the MOP is the corner stone
  – FT completely rely on the reflective mechanisms

• Very little work has been done
  – Few on formal verification
  – None on testing

• Validation of the FT architectures
  – Test of the underlying platform
  – Fault-injection
Testing Reflective Systems

1. Test order definition (reification, intercession, introspection)
2. Test objectives for each testing level
3. Conformance checks for each testing level
4. Test environments
Testing MOPs

\textit{TL0}. Testing preceding the MOP activation

\textit{TL1}. Reification mechanisms

\textit{TL2}. Behavioral intercession mechanisms

\textit{TL3}. Introspection mechanisms

\textit{TL4}. Structural intercession mechanisms
Incremental Test Order

TL0. implementation dependent

TL1. Reification mechanisms

TL2. Behavioral intercession mechanisms

TL3. Introspection mechanisms

TL4. Structural intercession mechanisms
TL1: Reification
(behavioral observation)

Comparison:
- Invoked method
- Parameter values
- Output values

- Observation
- Method execution is simulated by generating a random value for each output parameter

Oracle -> 6

1. test driver -> service

2. metaobject

3. 3

4. object

5. 5

6. Comparison:

- Invoked method
- Parameter values
- Output values
TL2: Behavioral intercession
(behavioral control)

Oracle

- Reified information is systematically delivered to the server object
- Output values are returned to the test driver

Server traces are checked according to the data supplied by the test driver
TL3: Introspection
(structural observation)

metaobject

Behavioral intercession
Introspection

test driver
Oracle

Is the introspected state consistent with the initialization state?
TL4: Structural intercession
(structural control)

metaobject

Behavioral intercession

Structural intercession

Introspection

Is the introspected state consistent with the initialization state?
interface shortTypeParameters{
    short ReturnValue ();
    void InValue (in short v);
    void OutValue (out short v);
    void InOutValue (inout short v);
    short All (in short v1,
               out short v2,
               inout short v3);
};

interface shortTypeAttributes{
    attribute short ReadWriteValue ;
    attribute readonly short ReadValue ;
};
Test Experiments (II)
(object-oriented properties considered)

• Inheritance:

• Encapsulation (methods and attributes):
  public / protected / private
Experimental results

- Reification / Behavioral intercession
  - Method invocations were incorrectly handled using inheritance
  - Internal object activity was incorrectly encapsulated

- Introspection / Structural intercession
  - Object composition vs Object references

```java
int fact(int i){
    if (i==0) return 1;
    return i*fact(i-1);
}
```
About testing MOPs

- Step forward for testing reflective systems
- Reusing mechanisms already tested for testing the remaining ones.
- Case Study: feasibility and effectiveness of the proposed approach
- Automatic generation of test case input values
- Guidelines for MOP design

Future work

- Generalizing the approach
  - Multi-level reflective systems
  - Aspect-oriented programming
- Testing reflection ➔ Reflection for testing
Conclusion

• MOPs for FT architectures
  – Language reflection / middleware not reflective
  – CORBA Portable Interceptors
    • Support for FT too limited
  – Unified approach for multi layered open systems
    • Multi-level reflection

• Validation of the platform
  – Test : augment the confidence
  – FI : failure mode analysis
    • feedback on FT mechanisms