The Composite "Know Thyself"

Fault-Tolerance in Complex Computer Systems Requires an Holistic Approach!

François Taïani
Context

- **Increasingly complex** computer systems are being used for increasingly critical applications.

  application domains
  (home appliances, transport, medicine,...)

  ![Diagram]

  increasing complexity
  complex computer systems

  - Complexity ?
    - In space :
      - several hundreds of **persons**
      - numerous **components** / numerous **suppliers**
    - In time :
      - very long life cycles : 10, 20 years

  - Criticity ? : service failure ⇒ major disaster (😊😊 / $$ )
    - **Dependability** is a major concern.
The Problem

- **Dependability** is a global property.
  - Vulnerabilities are scattered all through the system.
  - The system has to behave safely as a whole.
  - Fault-Tolerance requires observation + action capabilities.
    - checkpoint, control of non-deterministic decisions, etc.

- Controlling complexity relies on locality.
  - Components are modular, mechanisms are transparent.
  - But implementation remains totally hidden.
  - Systems are organized in heterogeneous layers.
The Problem

- **Dependability** is a **global** property.
  - Vulnerabilities are scattered all through the system.
  - The system has to behave safely as a whole.
  - Fault-Tolerance requires observation + action capabilities.
    - checkpoint, control of non-deterministic decisions, *etc.*

- Controlling complexity relies on **locality**.
  - Components are **modular**, mechanisms are **transparent.😊**
  - But implementation remains **totally hidden. 😞**
  - Systems are organized in **heterogeneous layers. 😞**

⇒ Dependability (globality) and complexity control (locality) have conflicting needs.
The Problem

- **Dependability** is a **global** property.
  - Vulnerabilities are scattered all through the system.
  - The system has to behave safely **as a whole**.
  - **Fault-Tolerance** requires **observation + action** capabilities.
    - checkpoint, control of non-deterministic decisions, *etc.*

- Controlling **complexity** relies on **locality**.
  - Components are **modular**, mechanisms are **transparent.😊**
  - But implementation remains **totally hidden. 😞**
  - Systems are organized in **heterogeneous layers. 😞**

⇒ **Dependability (globality) and complexity control (locality) have conflicting needs.**
Today's Industrial Practice

- Most COTS have not been built with dependability in mind.

Diagram:

- Application
- Middleware
- OS
Today's Industrial Practice

- Most COTS have not been built with dependability in mind.
- Had hoc adaptation is required.

fault-tolerance "patches"
Today's Industrial Practice

- Most COTS have not been built with dependability in mind.
- Ad hoc adaptation is required.

fault-tolerance "patches"

ad hoc inter-level coordination
Today's Industrial Practice

- Most COTS have not been built with dependability in mind.
- Had hoc adaptation is required.

fault-tolerance "patches"

ad hoc inter-level coordination

ad-hoc connection
original code ↔ FT patches
Ad Hoc Patches...

- cannot be reused.
  - Ad hoc patches are component specific.
  - Ad hoc patches decrease system maintainability.
  - Ad hoc patches cannot evolve easily.

- require the collaboration of the component's provider.
  - Ad hoc patches often depend on undocumented features.
    - Example: VxWorks in NASA probes
  - Ad hoc patches require consulting and "in-sourcing".
    - Cf. NASA & WindRiver, French Railway & ILOG (signal system)
Ad Hoc Patches...

...cannot be reused.($!)

- Ad hoc patches are component specific.
- Ad hoc patches decrease system maintainability.
- Ad hoc patches cannot evolve easily.

...require the collaboration of the component's provider. ($!)

- Ad hoc patches often depend on undocumented features.
  - Example: VxWorks in NASA probes
- Ad hoc patches require consulting and "in-sourcing".
  - Cf. NASA & WindRiver, French Railway & ILOG (signal system)

⇒ A high cost for an entangled system.
Separation & Architecture

- We are looking for an architectural paradigm that would allow us to separate fault-tolerance concerns from the rest of the system.
Separation & Architecture

- We are looking for an architectural paradigm that would allow us to separate fault-tolerance concerns from the rest of the system.

- Fault-Tolerance is a transversal concern.
  Most fault-tolerance mechanisms don't depend on the application domain.
Separation & Architecture

- We are looking for an architectural paradigm that would allow us to separate fault-tolerance concerns from the rest of the system.

- Fault-Tolerance is a transversal concern.
  
  Most fault-tolerance mechanisms don't depend on the application domain.

- Reflection seems a powerful solution for this problem:
  
  ➜ It allows the separation of transversal concerns.
  
  ➜ It was used to add fault-tolerance to several small systems.
Separation & Architecture

- We are looking for an architectural paradigm that would allow us to separate fault-tolerance concerns from the rest of the system.

- Fault-Tolerance is a transversal concern.
  Most fault-tolerance mechanisms don't depend on the application domain.

- Reflection seems a powerful solution for this problem:
  ➔ It allows the separation of transversal concerns.
  ➔ It was used to add fault-tolerance to several small systems.

⇒ Can reflection be used to realize adaptable fault-tolerance in complex systems?
Outline
Outline

(A) What Is Reflection?
Outline

■ (A)  What Is Reflection ?

■ (B)  Algorithmic Perspective :
       Which Reflexive Needs for Fault Tolerance?
Outline

■ (A) What Is Reflection?

■ (B) Algorithmic Perspective:
     Which Reflexive Needs for Fault Tolerance?

■ (C) Architectural Perspective:
     Reflection in Complex Systems
Outline

■ (A)  What Is Reflection ?

■ (B)  Algorithmic Perspective :
Which Reflexive Needs for Fault Tolerance?

■ (C)  Architectural Perspective :
Reflection in Complex Systems

■ (D)  A Case Study :
Replicating a Multithreaded Linux/CORBA Platform
Outline

■ (A) What Is Reflection?

■ (B) Algorithmic Perspective: Which Reflexive Needs for Fault Tolerance?

■ (C) Architectural Perspective: Reflection in Complex Systems

■ (D) A Case Study: Replicating a Multithreaded Linux/CORBA Platform
What Is Reflection?

functional interface, externally visible

External World

some computer system
What Is Reflection?

functional interface, externally visible

interne implementation, hidden from the external world
uses ◼️; △; □; →

External World
What Is Reflection?

- Functional interface, externally visible
- Internal implementation, hidden from the external world
  - "programming model": procedures + variables, objects + methods...
What Is Reflection?

By default: strict separation between a program and the universe it has effects upon.

Interna implementation, hidden from the external world uses

"programming model": procedures + variables, objects + methods ...
What Is Reflection?

Reflection gives access to a system's internals, which can thus be observed and modified.
What Is Reflection?

Reflection gives access to a system's internals, which can thus be observed and modified.

How?:
by exporting a representation of the system's internals.

Generic building elements are exported.
What Is Reflection?

Reflection gives access to a system's internals, which can thus be observed and modified.

How?:
by exporting a representation of the system's internals.

This representation is called a meta-model.

Generic building elements are exported.
What Is Reflection?

Internal activities are reflected on the meta-model.
What Is Reflection?

Internal activities are reflected on the meta-model.
What Is Reflection?

Internal activities are reflected on the meta-model.
What Is Reflection?

Internal activities are reflected on the meta-model.

meta-model
What Is Reflection?

Internal activities are reflected on the meta-model.

Changes on the meta-model are reflected on the system.
What Is Reflection?

Internal activities are reflected on the meta-model.

Changes on the meta-model are reflected on the system.

meta-model
What Is Reflection?

Internal activities are reflected on the meta-model.

Changes on the meta-model are reflected on the system.
What Is Reflection?

Internal activities are reflected on the meta-model.

Changes on the meta-model are reflected on the system.
What Is Reflection?

Internal activities are reflected on the meta-model.

Changes on the meta-model are reflected on the system.
What Is Reflection?

With reflection, the **program** becomes a **data** than can be manipulated.

Internal activities are **reflected** on the meta-model.

Changes on the meta-model are **reflected** on the system.

*meta-model*
Reflection & Transversal Concerns

IF "□" + "▲" THEN "□▲" BECOMES "□▲□"

meta-model
Reflection & Transversal Concerns

With reflection, generic programs can be written to manipulate the system independently of its functional features.

IF "☐" + "↑" THEN "☐→" BECOMES "☐→☐"

meta-model
Reflection & Transversal Concerns

With reflection, generic programs can be written to manipulate the system independently of its functional features.
Reflection & Transversal Concerns

With reflection, generic programs can be written to manipulate the system independently of its functional features.
Reflection & Transversal Concerns

With reflection, generic programs can be written to manipulate the system independently of its functional features.
Reflection & Transversal Concerns

⇒ With reflection, fault-tolerance can be addressed independently of the rest of the system.
Reflection & Complexity

- Implementing a reflexive architecture requires to find out which information is needed, and where to find it.
Reflection & Complexity

- Implementing a reflexive architecture requires to find out which information is needed, and where to find it.

machine instructions
(interruptions, basic logical operations, ...)

Taïani
Reflection & Complexity

- Implementing a reflexive architecture requires to find out which information is needed, and where to find it.

**system calls**
(synchronization, memory management...)

**machine instructions**
(interruptions, basic logical operations, ...)

Taïani
Reflection & Complexity

- Implementing a reflexive architecture requires to find out which information is needed, and where to find it.

**middleware services**
(remote invocations, etc...)

**system calls**
(synchronization, memory management...)

**machine instructions**
(interruptions, basic logical operations, ...)

Taïani
Implementing a reflexive architecture requires to find out which information is needed, and where to find it.

**Middleware services**
(remote invocations, etc...)

**System calls**
(synchronization, memory management...)

**Machine instructions**
(interruptions, basic logical operations, ...)

Reflection & Complexity
Reflection & Complexity

- Implementing a reflexive architecture requires to find out which information is needed, and where to find it.

- Complex systems are organized in many heterogeneous abstraction levels.

- The different levels are coupled, but this coupling remains hidden.
Reflection & Fault Tolerance

- In complex systems:
  - Each layer possesses its own programming model.
  - The respective meta-models are heterogeneous/incompatible.
Reflection & Fault Tolerance

- Reflection has been used to add FT to complex systems but:
  - Only one level of abstraction considered at a time so far.
  - Available fault-tolerance constrained by this limitation.
Reflection & Fault Tolerance

- Reflection has been used to add FT to complex systems but:
  - Only one level of abstraction considered at a time so far.
  - Available fault-tolerance constrained by this limitation.
Reflection & Fault Tolerance

Reflection has been used to add FT to complex systems but:

- Only one level of abstraction considered at a time so far.
- Available fault-tolerance constrained by this limitation.
Revisiting our Goals

- Our original goal:
  We want to implement fault-tolerance in a **sound**, and **disciplined** way while encompassing the **whole** system.
Revisiting our Goals

- Our original goal:
  
  We want to implement fault-tolerance in a **sound**, and **disciplined** way while encompassing the **whole** system.
Revisiting our Goals

How to integrate the views provided by each component into an **holistic** and **consistent** meta-model of the system?
Revisiting our Goals

What does fault-tolerance requires?

fault-tolerance

application

middleware

OS
Revisiting our Goals

What does fault-tolerance requires?

Which elements to export? How to combine them?

fault-tolerance

application

middleware

OS
Revisiting our Goals

What does fault-tolerance requires?

Which elements to export? How to combine them?

Algorithms

fault tolerance

application

middleware

OS
Revisiting our Goals

What does fault-tolerance requires?

Which elements to export?
How to combine them?

Algorithms

Architecture
Revisiting our Goals

What does fault-tolerance requires?

Which elements to export?
How to combine them?

Algorithmicians

Application

Architecture

OS
Revisiting our Goals

What does fault-tolerance requires?

Which elements to export? How to combine them?

Algorithmicians

Praticians
Outline

■ (A) What Is Reflection?

■ (B) Algorithmic Perspective:
    Which Reflexive Needs for Fault Tolerance?

■ (C) Architectural Perspective:
    Reflection in Complex Systems

■ (D) A Case Study:
    Replicating a Multithreaded Linux/CORBA Platform
Operating Needs & Performances
Operating Needs & Performances

A generic fault-tolerance algorithm ...

- ... is defined using an abstract computation model;
  - Ex. : state machine, process, asynchronous messages
- ... observes et acts upon this computation model;
  - Ex. : message interception, checkpointing and state recovery
- ... relies on action and observation capacities.
Operating Needs & Performances

A generic fault-tolerance algorithm ...
- ... is defined using an abstract computation model;
  - Ex. : state machine, process, asynchronous messages
- ... observes et acts upon this computation model;
  - Ex. : message interception, checkpointing and state recovery
- ... relies on action and observation capacities.

"Imperfect" capacities don't necessarily violate correctness.
- But too much approximation may result in intractable solutions.
- In a complex system : "good" capacities are difficult to realize within a unique layer.
Operating Needs & Performances

A generic fault-tolerance algorithm ...
  ➔ ... is defined using an abstract computation model;
      • Ex. : state machine, process, asynchronous messages
  ➔ ... observes et acts upon this computation model;
      • Ex. : message interception, checkpointing and state recovery
  ➔ ... relies on action and observation capacities.

"Imperfect" capacities don't necessarily violate correctness.
  ➔ But too much approximation may result in intractable solutions.
  ➔ In a complex system : "good" capacities are difficult to realize within a unique layer.

In a complex system, the precision of action and observation capacities is essential for fault-tolerance.
Capturing Fault-Tolerance Needs

- Our proposal: **Reflective Footprints**
  - They explicitly capture the reflexive capacities that are needed by a family of mechanisms.
  - They uncouple algorithmic core from concrete instrumentation.
  - They are architecture neutral.

- Example: replication
Capturing Fault-Tolerance Needs

- Our proposal: Reflective Footprints
  - They explicitly capture the reflexive capacities that are needed by a family of mechanisms.
  - They uncouple algorithmic core from concrete instrumentation.
  - They are architecture neutral.

- Example: replication
Capturing Fault-Tolerance Needs

- Our proposal: Reflective Footprints
  - They explicitly capture the reflexive capacities that are needed by a family of mechanisms.
  - They uncouple algorithmic core from concrete instrumentation.
  - They are architecture neutral.

- Example: replication

```plaintext
+----------------+          +----------------+
| client         |          | server 1       |
| application     |          | server 2       |
```

Taïani
Capturing Fault-Tolerance Needs

- Our proposal: Reflective Footprints
  - They explicitly capture the reflexive capacities that are needed by a family of mechanisms.
  - They uncouple algorithmic core from concrete instrumentation.
  - They are architecture neutral.

- Example: replication

  ➔ reflective footprint: a) message interception from client to servers
    b) state transfer
    c) controlling non-determinism … etc.
Reflective Footprints & Adaptation

- The **reflective footprint** of a set of FT mechanisms ... ➔ un couples the choice of an algorithm from its operating needs.
Reflective Footprints & Adaptation

- The \textit{reflective footprint} of a set of FT mechanisms ...
  \Rightarrow \textit{uncouples} the choice of an algorithm from its operating needs.
Reflective Footprints & Adaptation

- The **reflective footprint** of a set of FT mechanisms ...
  - **uncouples** the choice of an algorithm from its operating needs.
Reflective Footprints & Adaptation

- The **reflective footprint** of a set of FT mechanisms ...
  - uncouples the choice of an algorithm from its operating needs.
  - One instrumentation can be reused $\Rightarrow$ better quality, $\nabla$ costs.
Reflective Footprints & Adaptation

- The reflective footprint of a set of FT mechanisms ...
  - uncouples the choice of an algorithm from its operating needs.
  - One instrumentation can be reused ⇒ better quality, ↘ costs.
  - Fault-tolerance can be changed during system development.
Reflective Footprints & Adaptation

- The **reflective footprint** of a set of FT mechanisms ...
  - uncouples the choice of an algorithm from its operating needs.
  - One instrumentation can be **reused** ⇒ **better quality**, ↓ costs.
  - Fault-tolerance can be **changed** during system development.
  - Lays the path for **dynamic** adaptation.
Where Do We Stand?

What does fault-tolerance requires?

Which elements to export?
How to combine them?

Algorithmicians

Praticians
Where Do We Stand?

Reflective footprints capture the reflexive needs for adaptable fault-tolerance.

Which elements to export? How to combine them?

Algorithmicians

Praticians
Where Do We Stand?

Reflective footprints capture the reflexive needs for adaptable fault-tolerance.

Where should we find the elements of a reflective footprint?
Outline

■ (A) What Is Reflection?

■ (B) Algorithmic Perspective:
Which Reflexive Needs for Fault Tolerance?

■ (C) Architectural Perspective:
Reflection in Complex Systems

■ (D) A Case Study:
Replicating a Multithreaded Linux/CORBA Platform
Abstraction & Information

- Complex systems contain heterogeneous abstraction levels.
  ⇒ Available information is heterogeneous.

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

- Lower levels:
  - 😊 Complete Information
  - 😞 But lacking semantics
Abstraction & Information

- Complex systems contain heterogeneous abstraction levels.
  - Available information is heterogeneous.

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

- Lower levels:
  - 😊 Complete Information
  - 😞 But lacking semantics

- Mono-level approaches work poorly in complex systems:
  - ➔ Lacking information: some mechanisms can't be implemented.
  - ➔ Lacking semantics ➔ resulting solution gets intractable.
Transcending Software Boundaries

- A system's **global semantics** **transcends** its lower levels.
  - For instance: observing a POSIX (OS-level) socket creation
    - It could be the start of a CORBA request.
    - It could also be some part of a X11 invocation.

- Introducing **semantic contexts**: example on an ORB

- Intercepting lower level activities:
  **On behalf of whom are we intercepting?**
Inter-Level Mapping

Understanding how levels map onto another in a complex system is a first step to be able to combine the brute action force of lower levels with the rich semantic overview of higher levels.
Inter-Level Mapping

Understanding **how levels map onto another** in a complex system is a first step to be able to combine the brute **action force** of lower levels with the rich **semantic overview** of higher levels.
Inter-Level Mapping

Understanding **how levels map onto another** in a complex system is a first step to be able to combine the brute **action force** of lower levels with the rich **semantic overview** of higher levels.

- Top-down use
  - ➔ checkpoint
  - ➔ control of non-determinism
Inter-Level Mapping

Understanding how levels map onto another in a complex system is a first step to be able to combine the brute action force of lower levels with the rich semantic overview of higher levels.

- Top-down use
  - checkpoint
  - control of non-determinism

- Bottom-up use
  - error propagation analysis
  - forward recovery
The Proposed Approach

fault-tolerance

application

middleware

OS
The Proposed Approach

1 family of mechanisms
The Proposed Approach

1. family of mechanisms

2. reflective footprint

fault-tolerance

application

middleware

OS
The Proposed Approach

1. family of mechanisms

2. reflective footprint

3. analyzing the chosen architecture
The Proposed Approach

1. family of mechanisms
2. reflective footprint
3. analyzing the chosen architecture
4. instrumentation
Outline

■ (A) What Is Reflection?

■ (B) Algorithmic Perspective:
   Which Reflexive Needs for Fault Tolerance?

■ (C) Architectural Perspective:
   Reflection in Complex Systems

■ (D) A Case Study:
   Replicating a Multithreaded Linux/CORBA Platform
Case Study: 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
Case Study: 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

- We use the 4 steps of our approach.
Case Study: 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
Case Study: 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 scheduling decisions?
# Replication's Footprint

## Reflexive Facets

<table>
<thead>
<tr>
<th>Observation</th>
<th>Communication</th>
<th>Execution</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>RequestReception</td>
<td>ExecutionPointStart</td>
<td>NonDeterministicPlatformCall</td>
</tr>
<tr>
<td></td>
<td>RequestSending</td>
<td>ExecutionPointEnd</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ReplySending</td>
<td>ExecutionPointReach</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ReplyReception</td>
<td>NonDeterministicFlowChange</td>
<td></td>
</tr>
<tr>
<td></td>
<td>getRequestContent</td>
<td>getExecutionPoint</td>
<td></td>
</tr>
<tr>
<td></td>
<td>getReplyContent</td>
<td>getServerState</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>getPlatformState</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Action</th>
<th>doSend</th>
<th>createExecutionPoint</th>
<th>forceResultOfPlatformCall</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>doReceive</td>
<td>setExecutionPoint</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>forceResultOfFlowChange</td>
<td></td>
</tr>
<tr>
<td></td>
<td>piggyBackDataOnMsg</td>
<td>setServerState</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>setPlatformState</td>
<td></td>
</tr>
</tbody>
</table>
# Replication's Footprint

## Reflexive Facets

<table>
<thead>
<tr>
<th>Observation</th>
<th>Communication</th>
<th>Execution</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>RequestReception</td>
<td>ExecutionPointStart</td>
<td>NonDeterministicPlatformCall</td>
<td></td>
</tr>
<tr>
<td>RequestSending</td>
<td>ExecutionPointEnd</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ReplySending</td>
<td>ExecutionPointReach</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ReplyReception</td>
<td>NonDeterministicFlowChange</td>
<td></td>
<td></td>
</tr>
<tr>
<td>getRequestContent</td>
<td>getExecutionPoint</td>
<td></td>
<td></td>
</tr>
<tr>
<td>getReplyContent</td>
<td>getServerState</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>getPlatformState</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Action</th>
<th>createExecutionPoint</th>
<th>forceResultOfPlatformCall</th>
</tr>
</thead>
<tbody>
<tr>
<td>doSend</td>
<td>setExecutionPoint</td>
<td></td>
</tr>
<tr>
<td>doReceive</td>
<td>forceResultOfFlowChange</td>
<td></td>
</tr>
<tr>
<td></td>
<td>piggyBackDataOnMsg</td>
<td>setServerState</td>
</tr>
<tr>
<td></td>
<td></td>
<td>setPlatformState</td>
</tr>
</tbody>
</table>
Implementing the Footprint

- Some elements of a software architecture are more stable than others.
  - Standard interfaces (CORBA, POSIX) remain (for a while).
  - Implementations (ORBacus, GNU/Linux) change (rapidly).

- Footprint implementation in 2 steps:

  Analysis: investigating the CORBA⇔ POSIX mapping
  - By factorizing our knowledge of several ORBs
  - The resulting (meta-)model covers all implementations.
  - We use the model to identify instrumentation points.

Concrete implementation on ORBacus + GNU/Linux
Investigating the CORBA ↔ POSIX Mapping

requests in process

incoming requests

outgoing requests

"thread pool"

application

CORBA

POSIX
Investigating the CORBA ↔ POSIX Mapping

requests in process

application

outgoing requests

"thread pool"

incoming requests

CORBA

POSIX

Taïani

35
Investigating the CORBA ↔ POSIX Mapping

requests in process

incoming requests

outgoing requests

"thread pool"

Taïani
Investigating the CORBA ⇔ POSIX Mapping

requests in process

incoming requests

outgoing requests

"thread pool"

Taïani
Investigating the CORBA ↔ POSIX Mapping

requests in process

incoming requests

outgoing requests

"thread pool"

application

CORBA

POSIX

Taïani
Investigating the CORBA ↔ POSIX Mapping

requests in process

incoming requests

outgoing requests

"thread pool"

Taïani
Investigating the CORBA ↔ POSIX Mapping

requests in process

incoming requests

outgoing requests

"thread pool"

Taïani
Investigating the CORBA ↔ POSIX Mapping

requests in process

incoming requests

outgoing requests

"thread pool"

Taïani
Investigating the CORBA ↔ POSIX Mapping

requests in process

incoming requests

outgoing requests

"thread pool"

Taïani
Investigating the CORBA ⇔ POSIX Mapping

requests in process

incoming requests

outgoing requests

"thread pool"

Taïani
Controlling Non-Determinism at the Application Level only

- Arbitrary scheduling of requests by middleware.
Controlling Non-Determinism at the Application Level only

- Arbitrary scheduling of requests by middleware.

- Replicating scheduling decisions observed at the application level only leads to a deadlock ...
  - ... caused by the thread pool (here of size 2).
  - The decisions taken by the middleware regarding dispatching can't be controlled from the application.

Taïani
What Causes the Problem

decision point that must be controlled

incoming requests

R1 R2 R3

requests in process

outgoing requests

application

middleware

"thread pool"

OS

Taïani
OS Level Only

- Low level thread synchronization can be controlled:
  ➞ The same thread scheduling can be enforced on all replicas.
  ➞ Requests are dispatched and processed in the same order.
  😊 All replicas reach the same state.
  (assumption: MT = only source of non-determinism)

- But this over-constrains the replicas' execution:
  ➞ Impossible to relate OS level activities to request processing.
  ➞ All lock operations must be replicated.

not equivalent 🔄 replication of every decision
Smart Scheduling Replication

- With CORBA and application semantics:
  - Application and CORBA reflection give semantic to OS-level actions.
  - This semantic allows optimal use of OS level reflection.

- Example: with a thread pool:
  - Which thread executes which request does not matter.
  - The following 2 executions are equivalent:

**no need to replicate this scheduling decision**
The Multi-Layer Meta-Model

- Meta-model centered on the lifecycle of a CORBA request
  - aggregates OS-level synchronization and request lifecycle

RequestBeforeApplication \(\rightarrow\) RequestContentionPoint \(\rightarrow\) RequestAfterApplication

request in application

request pre-processing

RequestContentionPoint (OS level synchronization)

ReceptionEnd \(\rightarrow\) ReceptionStart

request reception

ReplyStart \(\rightarrow\) ReplyEnd

sending of reply
class Request;
class Thread;
class StackChunk;
class ReifiedEvent;
class RequestLifeCycleEvent extends ReifiedEvent {
    public Request reifiedRequest;
    public Thread reifyingThread;
}
class BeginOfRequestReception extends RequestLifeCycleEvent;
class EndOfRequestReception extends RequestLifeCycleEvent;
class RequestBeforeApplication extends RequestLifeCycleEvent;
class RequestAfterApplication extends RequestLifeCycleEvent;
class BeginOfRequestResultSend extends RequestLifeCycleEvent;
class EndOfRequestResultSend extends RequestLifeCycleEvent;
class RequestContentionPoints extends RequestLifeCycleEvent;

class IntercessionCommand;
class ContinueExecution extends IntercessionCommand;
class SkipCallToApplication extends IntercessionCommand;

interface MetaLevel {
    IntercessionCommand reifyEventToMetaSynchronous(ReifiedEvent e);
}
interface BaseLevel {
    State captureApplicationState();
    void restoreApplicationState(State s);
    StackChunk captureApplicationStack(Thread t);
    void restoreApplicationStack(Thread t, StackChunk stack);
    void InjectRequestAtCommuncationLevel(Request r);
}
Instrumentation

- CORBA-POSIX mapping is generic.

- Instrumentation on GNU/Linux + ORBacus
  - The concrete architecture must be bound to the generic mapping.
  - Complex reverse-engineering: ORBacus > 110 000 LoC
  - Important abstraction effort (dedicated tool, CosmOpen)
  - Interface centered approach: «roots» / «foliage» metaphor
Experimental Apparatus

- **CosmOpen**: semi-automatic reverse-engineering suite
  - Dedicated to the **abstracting** effort needed for our work.
  - **Graph manipulation** operators, relies on dot (AT&T tool 😊)
  - **Structural** & behavioral analysis.
  - Very useful to handle **very large graphs**
    - A trace of ORBacus: 2066 invocations ⇒ **2066 nodes**
  - Free Software: [http://www.laas.fr/~ftaiani/7-software](http://www.laas.fr/~ftaiani/7-software)

- **Model extraction**:
  - Structural extraction: **4280** lines of C++ (with Doxygen)
  - Behavioral extraction: **1660** lines of C++ (with gdb)

- **Graph manipulation**: **17010** lines of Java

- **CosmOpen**: **22950** LoC
Instrumentation

- Behavioral middleware model:
  - obtained with CosmOpen
  - relates OS level actions to application level operations
  - identifies points of instrumentation of meta-model
Instrumentation

- Behavioral middleware model:
  - obtained with **CosmOpen**
  - relates OS level actions to application level operations
  - identifies points of instrumentation of meta-model

RequestBeforeApplication
Instrumentation

- Behavioral middleware model:
  - obtained with **CosmOpen**
  - relates OS level actions to application level operations
  - identifies points of instrumentation of meta-model

RequestBeforeApplication

RequestAfterApplication
Instrumentation

- Behavioral middleware model:
  - obtained with CosmOpen
  - relates OS level actions to application level operations
  - identifies points of instrumentation of meta-model

Methods:
- RequestBeforeApplication
- RequestAfterApplication
- RequestContentionPoint
Instrumentation

- **Generic** shared library (C++) for OS interception
  - 6590 lines of C++
  - meta-classes to intercept locks and mutex individually
    - MetaMutex, MetaSocket
  - supports "transcendence" by piggybacking threads
    - MetaThreadInfo, ThreadMetaMutex, ThreadMetaSocket

- **Generic** shared library (C++) for multi-level interception
  - 1460 lines of C++
  - uses OS interception to implement its meta-model
    - RequestContentionPoint, MetaRequestLifeCycle

- **Instrumenting ORBacus' original code**
  - Very low intrusion: 35 new lines
  - 0.02% of original code
Lessons Learnt

- The resulting meta-interface is **consistent & homogeneous**
  - Supports **non-determinism** and **checkpointing**.

- Our prototype implements the part on non-determinism.

- **Efficient**: for instance, replicating **synchronization**:
  - During the processing of one request in ORBacus:
    - 203 synch operations are observed (pthread_...)
    - Our prototype only needs to intercept 3 (gain: x 67).
    - Our previous analysis guaranties that these 3 interceptions are **sufficient** to maintain the ORB consistency.

- **Very low intrusion**: 0.02 % of original code was modified

- **Reusable**: tool CosmOpen, generic interception libraries
Conclusions

■ Comprehensive and adaptable fault-tolerance conflicts with 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.

■ Practical validation on an industrial platform.
  ➞ Analysis and reverse-engineering work (TAO, ORBacus, omniORB) using a dedicated tool (CosmOpen).
  ➞ Prototype implantation for the control of non-determinism on GNU/Linux + ORBacus.
Outlook

- Strict separation between interface & implementation too constraining for today's large systems

- The present work is only a first step.

- Generic gray-box approaches to gain increasing relevance.

- Consistent and disciplined exposure of implementations by exporting meta-models in a generic, standard format

- Already there for certain applications:
  - IC synthesis: IP blocks come with their meta-data
  - Computer Security: Proof Carrying Mobile Codes
Our Vision

fournisseur 1
composant fonctionnel A

fournisseur 2
composant fonctionnel B

fournisseur 3
mécanismes non-fonctionnels

intégré
intégration

système complexe multi-niveaux

Légende :

- interfaces fonctionnelles standardisées
- méta-modèle décrivant le composant de façon standardisée
- méta-interfaces génériques multi-niveaux
- découverte en ligne des choix d’implémentation

Taïani
Fault-Tolerance Algorithms
Fault-Tolerance Algorithms

- Can work with imperfect observation capacities:
Fault-Tolerance Algorithms

- Can work with imperfect observation capacities:
  - Most algorithms insures consistency constrains
    - e.g.: checkpoint, message synchronization
Fault-Tolerance Algorithms

- Can work with imperfect observation capacities:
  - Most algorithms insures consistency constrains
    - e.g.: checkpoint, message synchronization
  - They constrain the system execution among possible runs
    - Recovery from a given state, re-ordering of messages
Fault-Tolerance Algorithms

- Can work with imperfect observation capacities:
  - Most algorithms insures consistency constrains
    - e.g.: checkpoint, message synchronization
  - They constrain the system execution among possible runs
    - Recovery from a given state, re-ordering of messages
  - The « constrain » level depends on perceived reality.
Fault-Tolerance Algorithms

- Can work with imperfect observation capacities:
  - Most algorithms insures consistency constrains
    - e.g. : checkpoint, message synchronization
  - They constrain the system execution among possible runs
    - Recovery from a given state, re-ordering of messages
  - The « constrain » level depends on perceived reality.
  - « foggy » perception ⇒ pessimist behavior, over constraining
Fault-Tolerance Algorithms

- Can work with imperfect observation capacities:
  - Most algorithms insures consistency constrains
    - e.g.: checkpoint, message synchronization
  - They constrain the system execution among possible runs
    - Recovery from a given state, re-ordering of messages
  - The « constrain » level depends on perceived reality.
  - « foggy » perception $\Rightarrow$ pessimist behavior, over constraining
  - Algorithm remains correct, but becomes inefficient.
Fault-Tolerance Algorithms

- Can work with imperfect observation capacities:
  - Most algorithms insures consistency constrains
    - e.g. : checkpoint, message synchronization
  - They constrain the system execution among possible runs
    - Recovery from a given state, re-ordering of messages
  - The « constrain » level depends on perceived reality.
  - « foggy » perception ⇒ pessimist behavior, over constraining
  - Algorithm remains correct, but becomes inefficient.

- Practical consequences
Fault-Tolerance Algorithms

- Can work with imperfect observation capacities:
  - Most algorithms insures consistency constrains
    - e.g. : checkpoint, message synchronization
  - They constrain the system execution among possible runs
    - Recovery from a given state, re-ordering of messages
  - The « constrain » level depends on perceived reality.
  - « foggy » perception ⇒ pessimist behavior, over constraining
  - Algorithm remains correct, but becomes inefficient.

- Practical consequences
  - Cheap, easy to implement, observation capacities can be used.
Fault-Tolerance Algorithms

- Can work with imperfect observation capacities:
  - Most algorithms insures consistency constrains
    - e.g. : checkpoint, message synchronization
  - They constrain the system execution among possible runs
    - Recovery from a given state, re-ordering of messages
  - The « constrain » level depends on perceived reality.
  - « foggy » perception ⇒ pessimist behavior, over constraining
  - Algorithm remains correct, but becomes inefficient.

- Practical consequences
  - Cheap, easy to implement, observation capacities can be used.
  - But resulting system is inefficient.
Fault-Tolerance Algorithms

- Can work with imperfect observation capacities:
  - Most algorithms insures consistency constrains
    - e.g.: checkpoint, message synchronization
  - They constrain the system execution among possible runs
    - Recovery from a given state, re-ordering of messages
  - The « constrain » level depends on perceived reality.
  - « foggy » perception ⇒ pessimist behavior, over constraining
  - Algorithm remains correct, but becomes inefficient.

- Practical consequences
  - Cheap, easy to implement, observation capacities can be used.
  - But resulting system is inefficient.
  - In a complex system, «fogginess» increases.
Fault-Tolerance Algorithms

- Can work with imperfect observation capacities:
  - Most algorithms insures consistency constrains
    - e.g. : checkpoint, message synchronization
  - They constrain the system execution among possible runs
    - Recovery from a given state, re-ordering of messages
  - The « constrain » level depends on perceived reality.
  - « foggy » perception ⇒ pessimist behavior, over constraining
  - Algorithm remains correct, but becomes inefficient.

- Practical consequences
  - Cheap, easy to implement, observation capacities can be used.
  - But resulting system is inefficient.
  - In a complex system, «fogginess» increases.
  - Performances becomes intractable.