Verification and Validation of a Fault-Tolerant Architectural Abstraction

Patrick H. S. Brito - Unicamp, Brazil
Rogério de Lemos - University of Kent, UK
Eliane Martins - Unicamp, Brazil
Cecília M. F. Rubira - Unicamp, Brazil
Motivation

- Fault tolerance at the architectural level
  - idealised fault tolerant architectural element
    - exception handling
- Fault tolerance doesn't come for free
  - increase in complexity
    - e.g., exception propagation
- Improve confidence
  - verification by model checking architectural configurations
  - validation by generation of test cases
- How the abstraction is implemented is not the topic of this paper
Outline

- Motivation
- Exception handling and software fault tolerance
- Idealised fault tolerant architectural element
- Rigorous development approach
- Conclusions
- Future work
An architectural solution based on **exception handling**

- components need to collaborate for handling certain failure scenarios
- configurations that allow the propagation of exceptions
  - controlled error propagation

Exception handling is not “the” solution, there are other alternatives

- it might be perceived as undesirable, but it’s reality
- depends on the failure assumptions and costs
Idealised fault tolerant architectural element (iFTE)

system ifte_abstraction
features
  I_iFTE_PS_i: in event data port Service;
  I_iFTE_PS_o: out event data port Service;
  I_iFTE_PE_o: out event data port Exception;
  I_iFTE_RS_i: in event data port Service;
  I_iFTE_RS_o: out event data port Service;
  I_iFTE_RE_i: in event data port Exception;
flows
  Ret_Ser_a: flow path I_iFTE_PS_i -> I_iFTE_PS_o;
  Sig_Exc_a: flow path I_iFTE_PS_i -> I_iFTE_PE_o;
  Req_Ser_b: flow path I_iFTE_PS_i -> I_iFTE_RS_o;
  Ret_Ser_b: flow path I_iFTE_RS_i -> I_iFTE_PS_o;
  Sig_Exc_b: flow path I_iFTE_RS_i -> I_iFTE_PE_o;
  Ret_Ser_c: flow path I_iFTE_RE_i -> I_iFTE_PS_o;
  Sig_Exc_c: flow path I_iFTE_RE_i -> I_iFTE_PE_o;
end ifte_abstraction;
iFTE: Behavioural Scenarios

Client  |  LiFTE_PS  |  LiFTE_PE  |  LiFTE_RS  |  LiFTE_RE  |  Server

- Ret_Ser_a
- Sig_Exc_a
- Req_Ser_b
- Ret_Ser_b
- Req_Ser_b
- Ret_Ser_c
- Req_Ser_b
- Sig_Exc_b
- Req_Ser_b
- Sig_Exc_c
The main objectives of the approach

- Provide support for analysing exception propagation at the architectural level
- Analyse application-specific details about the exception propagation
- Define a scalable solution with support for automatic verification
- Define an approach for generating testing cases
A Rigorous Development Approach

Represent SA in B and CSP (using the provided templates)

Verify exception propagation properties of the SA

Execution scenarios are specified in CSP to reduce the states combination and improve the scalability of the proposed solution

Determine the best integration order for the architectural elements

[no property violation]

[property violation]

Adjust the specification of the software architecture

Generate model-based test cases (component and integration testing)
A Rigorous Development Approach

Represent the Structural Constraints of the software architecture in B-Method (elements and configuration)

Represent all the Possible Execution Scenarios (Generate the CSP file of the formal specification)

Reduces the states combination and improves the scalability of the verification
For each service of an iFTE
- Provided interfaces
- Required interfaces
- Provided exceptions
- Required exceptions
- Maskable exceptions

For the software architecture
- The architectural configuration
B-Method

- Type representation
  - different contexts for each type of exceptions
- Easiness to represent relations between types
  - architectural configuration, exception conversions, etc.

CSP

- Easiness to represent complex ordered events
  - execution scenarios, complex architectural propagation rules
The ProB model checker is used to check for both

- Violations of structural (architectural configuration) constraints
- Extended architectural descriptions are used to analyse exception flow properties

Users can specify their own properties for a specific exception handling model

Violations result in error messages and counter-examples
Some architectural properties that are verified

- Absence of deadlock
- Explicit declaration of external exceptions (component interfaces)
- All the required exceptions are handled
- Only maskable exceptions can be masked
Integration Order

- Integration order tries to minimise dependencies among architectural elements
- Reduce the integration test effort for constructing stubs
- Provides a way for reasoning about the coupling among architectural elements (dependency matrix)
Generation of Test Cases

- The only input is the formal model (B + CSP) of the software architecture
- A graph is created for representing the interaction among architectural elements
- Test cases are identified based on the paths of the interaction graph
- Stubs are specified by analysing the arrows departing from the required interfaces nodes
An Application Example: A Mining Control System

- 7 iFTE architectural elements: 4 comps. and 3 conns.
- 4 non-iFTE architectural components
Architecture configuration property

- every required service refers to a valid provided service of another component.

The following goal might never be satisfied:

- \( \exists c_1, c_2 \in \text{ArchitecturalElements}, t \in \text{EventType}, s \in \text{ArchitecturalServices}, e \in \text{ArchitecturalExceptions} \cdot (c_1, c_2, t, s, e) \in \text{sequenceHistory} \land c_1 \neq c_2 \land s \notin \text{providedArchService}(c_2) \)
The architectural elements of an iFTE follow recursively the iFTE abstraction.
Conclusions

Fault tolerance at the architectural level

- error handling
  - since iFTE is application dependent, we need to obtain assurances when it is instantiated to a particular application
    - model checking specifications for exception propagation
      - ProB (B Method and CSP)
    - generation of testing cases for integration testing
Future Work

- Adapt the proposed approach to other architectural abstractions using other fault models, e.g., crash failures

- Improve the tool support for:
  - Generating the formal models from a UML component diagram (*UML2Formal*)
  - Additional information about the exceptional behaviour can be represented in XMI through meta tags