Towards Continuous Evolution through Automatic Detection and Correction of Service Incompatibilities

Prof. dr. Benny Akesson
Motivation

Thales systems have life time > 30 years and require upgrades
  • Requirements significantly change during life time
  • New software with new capabilities becomes available

System upgrades can take 1-2 years and happen every 10-15 years
  • Many small updates collected into big infrequent upgrades
  • System evolves slowly and in big steps, increasing risk

Continuously evolution reduces risk and increases added value
Continuous Evolution

Facilitated by service-oriented architectures

Service-oriented architectures provide flexibility

- Components provide and require services for particular functionality
- Service dependencies are dynamically resolved
- Abstracting component implementing service through service interface
- Decouples application from a particular technology and implementation
Service-oriented Architecture

Thales INAETICS platform provides the context of this work

• Service-oriented architecture providing resilience and evolvability

Simplified terminology

• Services are implemented by components that communicate via message passing
• Service interface comprises set of valid message types (formal) and protocol (informal)
• Messages can be passed either synchronously or asynchronously
Problem Statement

Updating service interfaces comes with associated challenges

• Many components in many products may request or provide services
• Dynamic resolution of service dependencies makes it less explicit which components interact
• Determining impact of update on components is challenging
• Addressing this problem manually is expensive and time consuming

This applied research considers the problem of automatically detecting and correcting incompatibilities resulting from service updates
Compatibility

Two types of compatibility are considered:

- **Structural compatibility:** messages specified in the service interface, and their fields, match those used by the client in terms of name, type, and semantics.

- **Behavioral compatibility:** service and the clients agree on the protocol.
Contributions

Paper presents initial work towards by addressing the stated problem

The paper has three contributions:

1. **Survey of state-of-the-art** in areas of interface specification, and detection and correction of incompatible services
2. **Initial steps towards a methodology** to manage service incompatibilities
3. Work is discussed in context of **simplified case study** of a service in the radar domain
Presentation Outline

- Introduction
- State-of-the-Art
- Methodology
- Conclusions
Overview of State-of-the-Art

We survey the state-of-the-art in two areas, covering 30 publications:

1. **Interface specification**
   - **Structural specification**, e.g. programming languages and many interface definition languages
   - **Behavioral specification**, e.g. communication state machines, open nets, and process algebras
   - **Combinations of both**, e.g. Dezyne and ComMA languages

2. **Detection and correction of incompatibilities**
   - **Detection** of structural and behavioral incompatibilities
   - **Correction** of (structural and) behavioral incompatibilities through adapter generation

Please refer to paper for survey
Presentation Outline

- Introduction
- Case Study
- State-of-the-Art
- Methodology
- Demonstration
- Conclusions
Directions for Methodology

Five-step Methodology

1. Service Interface Specification for all services using ComMA (design time)
2. Generate Formal Model based on Open Nets from all specifications (design time)
3. Check Accordance between original and update using operating guidelines
4. Generate Adapter between services using controller synthesis
5. Generate Code from adapter model and deploy in INAETICS
Interface Specification

Step 1:
- Original ComMA model
- Updated ComMA model
- Partner ComMA model

Step 2:
- Original Open Net
- Updated Open Net
- Partner Open Net

Step 3:
- Check Accordance

Step 4:
- Generate Adapter

Step 5:
- Generate Code

SUCCESS, no adapter needed
SUCCESS, adapter generated
FAIL, no adapter exists
Service Interface Specification

ComMA selected as specification language for five reasons

1. specifies both structure and behavior, required to validate both aspects of compatibility
2. models both synchronous and asynchronous communication
3. successfully applied in industry before, i.e. at Philips
4. automatic inference and migration of interface specifications simplifies industrial adoption
5. the tooling is based on Eclipse, which is one of the most commonly used modeling tools in the embedded domain
Case study changes this service by replacing some uses of a state message with a new Performance message
Generate Formal Model
Generate Formal Service Model

Open Nets chosen as formal service model
  • Special type of Petri Nets with unconnected interface places

Open Nets for three main reasons:
  1. possible to **transform** a ComMA specification into an Open Net
  2. support both synchronous and asynchronous communication
  3. existing analysis methods are available, supported by academic tools
Check Accordance between Original and Updated Nets

Step 1
- Original ComMA model
- Update ComMA model
- Partner ComMA model

Step 2
- Generate Original Open Net
- Generate Updated Open Net
- Generate Partner Open Net

Step 3
- Check Accordance
  - accordance
  - no accordance

Step 4
- Generate Adapter

Step 5
- Generate Code
  - SUCCESS, adapter generated
  - FAIL, no adapter exists
Check Accordance

Method based on Operating Guidelines chosen

- Operating Guidelines are a characterization of all possible partners
- Basic idea is to check if all partners supported by one service are also supported by another
- Context-independent method covers all possible partner services simultaneously, advantage if the number of partners is large or unknown
- Method is exact and supported by academic tool Fiona
Accordance Checking Modified PeriodicTask

v2 does not simulate v1, nor are they equivalent
Adapter needed!
Generate Adapter

1. Specify structure and behavior
2. Generate
3. Check Accordance
4. Generate Adapter
5. Generate Code

SUCCESS, no adapter needed
SUCCESS, adapter generated
FAIL, no adapter exists
Generate Adapter

Adapter generation approach based on controller synthesis

- Adapter architecture comprises Engine and Controller
- Engine focuses on data flow and transformations
- Controller determines order of transformation and sending
- Engine structure follows directly from mapping rules
- Controllers are synthesized to guarantee deadlock freedom
- Approach supported by academic tools Marlene and Fiona

Mapping Rules

| PTOn -> PTOn;       |
| PTOff -> PTOff;     |
| PTStateReady -> PTStateReady; |
| PTState -> PTState; |
| PTPerformance -> PTState; |
Generate Code
Conclusions

Problem

• Systems with long life time need to continuously evolve
• Service-oriented architectures are enablers of continuous evolution
• Managing compatibility of evolving services remains a challenge

Methodology for detection/correction of incompatibilities was presented

• Technology selection based on survey of state-of-the-art
• Specifying structure and behavior of service interface using ComMA
• Generate Open Nets to check accordance of update and synthesize adapters, if necessary
I am happy to tell you more and demonstrate our work!