Model-Based Adaptation of Behavioral Mismatching Components

IEEE Transaction on Software Engineering, July/August, 2008

Carlos Canal, Pascal Poizat, and Gwen Salaün

2008.09.24
In-Gwon Song

© KAIST SE LAB 2008
Contents

- Introduction
- Background
- Overall approach
- Definitions
  - Behavioral component interface
  - Behavioral mismatch
  - Adaptation contracts
- Adaptation algorithms
  - Adaptation without reordering
  - Adaptation with reordering
- Related work
- Conclusion
- Discussion

© KAIST SE LAB 2008
Component based software engineering (development)

- Building new system by assembling existing software components
  - Increase reuse
  - Increase stability
  - Decrease development cost

Problem
- Existing software components are not perfectly matched with each other in many cases
  - Mismatch between components

Solution
- Introducing an adaptor entity to glue the mismatching

This paper propose algorithms to automatically generate an adaptor.
Classification of mismatch

- Technical level
  - Data encoding, OS, language, …
- Signature level
  - Operation name, type name, …
- Behavioral level
  - Interaction protocol
- Quality of service level
  - Non functional requirements(performance, security, …)
- Semantic level(conceptual level)
  - Functional requirements

In this paper, authors focus on signature level and behavioral level.
Labeled transition system (LTS)

- Primitive behavioral model
  - Similar to the automata and state machine

- Input-output LTS (in this paper, simply called as LTS)
  - Distinguish input/output by mark !(output) and ?(input)
PetriNet

- Consists of places, arcs, transitions, and marks
  - Places may hold marks (circle)
  - Arcs are directed edges connecting from places to transitions or from transitions to places (arrow)
  - Marks mean current execution points (big dot)
  - Transitions is fired when all places connected with incoming arcs hold marks
  - If a transition is fired, every mark held by places connected with incoming arcs are transferred to all places connected with outgoing arcs
PetriNet vs LTS for modeling reordering adaptor

- Modeling of reordering adapter is easier Petri-Net than LTS
- Example
  - Module A send message “a” and then send message “b”
  - Module B receive message “b” and then receive message “a”
Overall approach

- Generating adaptor protocol with component behavioral interfaces and adaptation contract

LTS based algorithm
(Efficient but cannot handle reordering)

PetriNet based algorithm
(Inefficient but can handle reordering)
Behavioral component interface (1/2)

- **Interface**
  - Generally, catalogue of operations of a component
    - Industrial standard
      - IDL (Interface Definition Language)

- **Behavioral component interface**
  - Usually, presented as behavioral (semi-)mathematical model
    (i.e. automata, labeled transition system (LTS), phi-calculus, ...)

- **In this paper,**
  - LTS is used to describe behavioral component interfaces
Example

- Cyber museum
  - ROOM show text or video through a PDA of an user
Behavioral mismatch (1/2)

Definition
- If a system composed of components fall into dead lock states, the components have behavioral mismatch
  - Such that each component is dead-lock free

Detecting mismatch
- Calculate system-wide LTS from component behavioral interfaces
  - By parallel composition (Imperial college)
  - By synchronous product (This paper)

- Check existence of reachable deadlock state at the system-wide LTS
Behavioral mismatch (2/2)

- Example of synchronous product

Transitions which have their another pair are remained.
Guide or hint for generating an adaptor

**Consists of**
- Synchronous vector (simply vector)
- LTS of synchronous vectors
  - Vector LTS

**Synchronous vector**
- Mapping from input event to output event
  - i.e. \((\text{query!}, \text{query?})\)
- Similar with the connector in Component and Connector view of architectural description
- Give clues for resolving signature mismatch
  - \((\text{text!}, \text{pdf?})\)
Example of synchronous vector

- \( v_{\text{query}} = \{\text{ROOM : query?}, \text{PDA : query!}\}, \)
- \( v_{\text{list}} = \{\text{ROOM : list!}, \text{PDA : list?}\}, \) and
- \( v_{\text{choice}} = \{\text{ROOM : choice?}, \text{PDA : choice!}\} \)
- \( v_{\text{end}} = \{\text{ROOM : _, PDA : shutdown!}\} \)
- \( v_{\text{vmode}} = \{\text{ROOM : videorequest?}, \text{PDA : _}\} \)
- \( v_{\text{tmode}} = \{\text{ROOM : textrequest?}, \text{PDA : _}\} \)
- \( v_{\text{vget}} = \{\text{ROOM : video!}, \text{PDA : mpeg?}\} \)
- \( v_{\text{tget}} = \{\text{ROOM : text!}, \text{PDA : pdf?}\} \)

Synchronous vectors

Solely using synchronous vectors, only static binding can be described → In any context, message mapping is not changed
Vector LTS

- Enable context-sensitive mapping
  - If a system has a state in certain context, then a mapping corresponding to the state is provided
  - Labels of vector LTS are synchronous vectors

→ All vectors are provided at any context

→ Several execution paths are restricted
Overview

Component behavioral interfaces

Adaptation contract

Synchronous vector product

Product LTS

Removing deadlock states and splitting transitions

Adaptor protocol (LTS)
Synchronous vector product

- Similar to the synchronous product except
  - Vector LTS maps or restricts input/output messages

- Rules
  - States of resulted product LTS are every combination of following states
    - States of component behavioral interfaces
    - States of vector LTS
  - Transition of resulted product LTS is created as “(a!, b?)” when
    - Transition of one of the components behavioral interfaces has label ‘a!’
    - Transition of one of the components behavioral interfaces has label ‘b?’
    - At such context, a state of vector LTS has an outgoing transition “(a!, b?)”

Unreachable states can be ignored
Adaptation without reordering (3/4)

**Example**

\[
\begin{align*}
\text{v}_{\text{query}} &= \langle \text{ROOM : query?}, \text{PDA : query!} \rangle, \\
\text{v}_{\text{list}} &= \langle \text{ROOM : list!}, \text{PDA : list?} \rangle, \text{ and} \\
\text{v}_{\text{choice}} &= \langle \text{ROOM : choice?}, \text{PDA : choice!} \rangle. \\
\text{v}_{\text{end}} &= \langle \text{ROOM : -}, \text{PDA : shutdown!} \rangle \\
\text{v}_{\text{vmode}} &= \langle \text{ROOM : videorequest?}, \text{PDA : -} \rangle \\
\text{v}_{\text{tmode}} &= \langle \text{ROOM : textrequest?}, \text{PDA : -} \rangle \\
\text{v}_{\text{vget}} &= \langle \text{ROOM : video!}, \text{PDA : mpeg?} \rangle \\
\text{v}_{\text{tget}} &= \langle \text{ROOM : text!}, \text{PDA : pdf?} \rangle
\end{align*}
\]

Synchronous vectors

Partial view of resulted LTS

ROOM LTS

PDA LTS

Vector LTS

(0,0)  (query?, query!)  (1,1)  (list!, list?)  (2,2)

© KAIST SE LAB 2008
Removing deadlock states and splitting transitions

- Deadlock states and related transitions are simply removed from product LTS \(\rightarrow\) preventing deadlocks
- All transitions "(a?, b!)") of product LTS are split into a transition "a!" and a transition "b?" to create protocols of an adapter

Example
Adaptation with reordering (1/4)

Overview

- Component behavioral interfaces
- Transforming into PetriNet
- Adaptation contract
- Transforming into PetriNet
- PetriNet representation
- Creating marking graph
- Removing deadlock states and reduction
- Adaptor protocol (LTS)
- A.K.A. occurrence graph

Traditional Approach In PetriNet
Transforming component behavioral interfaces into PetriNet

LTS (behavioral Interfaces)

PetriNet

Before firing transition
After firing transition

Message reception in components
Message emission in components

Dashed place means emission/reception to/from adapter
Transforming an adaptation contract into PetriNet

Before firing transition

After firing transition

Dashed place means emission/reception to/from components
Adaptation with reordering (4/4)

Example (component interfaces)
Related work

- Schmidt and Reussner’s work
  - Using parameterized contracts
    - Less expression power
  - Solving protocol incompatibilities (reordering)
  - Using an algorithm similar to the synchronous product

- Dumas et al.’s work
  - Dealing with various message transfering (one-to-one, one-to-many, many-to-one, one-to-zero)
  - Reordering can not be handled
Conclusion

❖ Summary

- Proposing more expressive the adaptation contract using the synchronous vector and the vector LTS
  - Context sensitive mapping between messages of interfaces
- Providing two adaptor generating algorithm for the efficiency of computation and reordering capability
  - LTS based algorithm (without reordering)
  - PetriNet based algorithm (with reordering, but inefficient)

❖ Future work

- Support for contract design
- Self-adaptive pervasive system
  - Instant and fully-automated generation of adaptor
Pros
- Very expressive adaptation contact
- Easy formalism to understand
- Clearly let me know that what is needed to generate adaptors automatically

Cons
- Only support synchronous messages even though they treat reordering
- Adaptation contract is very expressive, but very hard to design or derive
For my research,

- Deriving the adaptation contract from sequence diagrams
  - Since my overall approach is started from UML sequence diagram based scenario specification
- Supporting the asynchronous messaging style