# Robust Collaborative Services Interactions under System Crashes and Network Failures

Lei Wang

**Graduation committee:**

Chairman and Secretary:
Prof.dr.ir. W.G. van der Wiel, University of Twente, the Netherlands
PhD Supervisor:
Prof.dr. P.M.G Apers, University of Twente, the Netherlands
Second Supervisor:
Prof.dr. R.J. Wieringa, University of Twente, the Netherlands
Co-Supervisor:
Dr. Andreas Wombacher, Achmea, the Netherlands

Members:
Prof.dr. Chi-Hung Chi, CSIRO, Australia
Prof.dr. Manfred Reichert, University of Ulm, Germany
Prof.dr.ir Marco Aiello, University of Groningen, the Netherlands
Prof.dr.ir L.J.M. Nieuwenhuis, University of Twente, the Netherlands
Dr.ir. M.J. van Sinderen, University of Twente, the Netherlands
Dr. L. Ferreira Pires, University of Twente, the Netherlands

CTIT

**CTIT Ph.D. thesis Series No. 15-357**
Centre for Telematics and Information Technology
University of Twente
P.O. Box 217, NL – 7500 AE Enschede

ISSN 1381-3617
ISBN 978-90-365-3868-8
DOI 10.3990/1.9789036538688

Publisher: Ipskamp Drukkers
Cover design: Wanshu Zhang

Copyright © Lei Wang

# ROBUST COLLABORATIVE SERVICES INTERACTIONS UNDER SYSTEM CRASHES AND NETWORK FAILURES

PROEFSCHRIFT

ter verkrijging van
de graad van doctor aan de Universiteit Twente,
op gezag van de rector magnificus,
prof.dr. H. Brinksma,
volgens besluit van het College voor Promoties,
in het openbaar te verdedigen
op donderdag 23 april 2015 om 14.45 uur

door

Lei Wang

geboren op 04 may 1984
te Harbin, Heilongjiang, People's Republic of China

Dit proefschrift is goedgekeurd door:
Promotor: prof.dr. P.M.G. Apers
Co-promotor: prof.dr. R.J. Wieringa

# ROBUST COLLABORATIVE SERVICES INTERACTIONS UNDER SYSTEM CRASHES AND NETWORK FAILURES

DISSERTATION

to obtain
the degree of doctor at the University of Twente,
on the authority of the rector magnificus,
prof.dr. H. Brinksma,
on account of the decision of the graduation committee,
to be publicly defended
on Thursday the 23rd of April 2015 at 14:45 by

Lei Wang

born on May 4th, 1984
in Harbin, Heilongjiang, People's Republic of China

This dissertation has been approved by:
Promotor: prof.dr. P.M.G. Apers
Co-promotor: prof.dr. R.J. Wieringa

# Abstract

Electronic collaboration has grown significantly in the last decade, with applications in many different areas such as shopping, trading, and logistics. Often electronic collaboration is based on automated business processes managed by different companies and connected through the Internet. Such a business process is normally deployed on a process engine, which is a piece of software that is able to execute the business process with the help of infrastructure services (operating system, database, network service, etc.).

With the possibility of system crashes and network failures, the design of robust interactions for collaborative processes is a challenge. System crashes and network failures are common events, which may happen in various information systems, e.g., servers, desktops, mobile devices. Business processes use messages to synchronize their state. If a process changes its state, it sends a message to its peer processes in the collaboration to inform them about this change. System crashes and network failures may result in loss of messages. In this case, the state change is performed by some but not all processes, resulting in global state/behavior inconsistencies and possibly deadlocks.

In general, a state inconsistency is not automatically detected and recovered by the process engine. Recovery in this case often has to be performed manually after checking execution traces, which is potentially slow, error prone and expensive. Existing solutions either shift the burden to business process developers or require additional infrastructure services support. For example, fault handling approaches require that the developers are aware of possible failures and their recovery strategies. Transaction approaches require a coordinator and coordination protocols deployed in the infrastructure layer.

Our idea to solve this problem is to replace each original process by a robust counterpart, which is obtained from the original process through an automatic transformation, before deployment on the process engine. The robust process is deployed with the same infrastructure services and automatically recovers from message loss and state inconsistencies caused by system crashes and network failures. In other words, the robust processes are transparent to

developers while leaving the infrastructure unmodified.

We assume a synchronous interaction scenario for collaborative processes. With this scenario, an initiator sends a request message to a responder, and waits for a response message, while a responder receives the request message, applies some state change and sends the response messages. With our proposed transformation we obtain robust processes, where each process in the responder role caches the response message if its state has changed by the previously received request message. The possible state inconsistencies are recognized by using timers and information provided by the infrastructure, and resolved by using cached state and by retrying failed interactions. We also considered more complex interaction scenarios with multiple initiator and responder instances (*1-n*, *n-1* and *n-n* client-server configurations).

We have provided a formal proof of the correctness of our transformation solution. We have also done a performance analysis and determined the overhead of the generated (robust) processes compared to the original processes. Since this overhead is low compared to the performance differences that exist as a consequence of using different process engines, we argue that the generated robust processes have applicability in real life business environments.

By doing this work, we have learnt the possible failure situations that affect the global state/behavior of collaborative business processes. Furthermore, we have defined transformations for deriving robust processes that are capable of surviving the identified failures.

# Acknowledgments

Whee! Eventually, it comes to the section I should say with most concerned. And here is my heartfelt gratitude.

There's been through some tough times in the past years, fortunately I surpassed myself with all your support and encourage, which is somehow a milestone I touched along. Life is so beautiful with all your edification and accompany, your pansophy, creative, humorous, kindness made these years a good inspiration station filled with love, laughter. I am afraid such pages of acknowledgments cannot express all my gratitude, but I swear I have them all in my mind.

I would like to express my appreciation to the members of my PhD committee, starts from the ones furthest away: Prof.Dr. Chi-Hung Chi, Prof.Dr. Manfred Reichert, Prof.Dr.Ir Marco Aiello, Prof.Dr.Ir L.J.M. Nieuwenhuis. It is a great privilege to have each of you invited in my defense committee. I feel very much indebted to encroaching upon your valuable time, and appreciate your precious feedback in sharpening my thesis. My special thanks gives to Prof. Dr. Chi-Hung Chi, thank you for you cultivation ever since my master study, thank you for being firm with me while I went through my rebellion stage. Without your disposal I couldn't get here in my doctoral research.

I would like to express my appreciation to my promotors: Prof.Dr. P.M.G. Apers and Prof.Dr. R.J. Wieringa for the support and continuous encouragement, and for the constructively review on the manuscript.

I would like to express my appreciation to my daily supervisors Andreas Wombacher, Luís Ferreira Pires and Marten van Sinderen. -Andreas, you have been a tremendous mentor for me. I would like to thank you for your encouragement on my research, for scratching my back to grow as a critical researcher. Your advice on my research as well as on my career have been priceless. Here are also thanks to your family for the hospitality at your home. -Marten, you are always there given promptly help at a pinch. I do thank you for the countless inspiring discussions, thank you for every noodlework on my papers and the tremendous time you spent on my thesis revision. Here are also thanks

for the nice dinner organized by you and Luís. -Luís, thank you for getting down to all my works. The suggestions of revisions are always put forward with long pages of solid text in red mark. Say my technical writing skills were rather weak but for sure it have improved a lot. Moreover, I would say I was much under the influence of your punctilious working manner and brilliant sense of humor, which always made our discussion efficient and pleasant.

Again, my deepest gratitude to all my supervisors, your consideration and patience in very particular sometimes means everything of impetus that kept me going over the low ebb. Thank you for tolerance and, and... I don't think I can ever thank you enough for what you have done for me.

I also would like to thank the colleagues of the DB and SCS group: Almer, Brend, Djoerd, Dolf, Ghita, Iwe, Jan, Juan, Kien, Maarten, Maurice, Mena, Mohammad, Rezwan, Robin, Sergio, Suse, Victor, Zhemin and all the others. Thank you for preserving such a nice working environment, for the nice DB colloquium and lunch time that we have spent together. Thank you for all the nice moments that we spent together during the times of group social events.

My special thanks to Ida and Suse for making a lot of impossible missions possible. Thanks Suse, Brend, Maarten and other Dutch colleagues and friends for practicing my Dutch. Thanks Mena for providing the latex template for the writing of this thesis. Thanks Brend for a highly configurable latex compile script which saves me huge amount of compilation time during this thesis writing.

I have been living in Macandra all the time working on my PhD in the Netherlands, it is a sort of slum but still gives a feeling of warmth while away from family. There I got to meet a lot of nice friends (Ashvin, Cams, Haishan, Cuiyang, Luzhou, Gaopeng, ZhaoZhao, Vivian, Michel, Dongfang, Xiao Xiexie), and I was always basking in the afterglow of whoop-de-do. I can still recall my first birthday in Macandra, the gorgeous meal, beautiful cake and the absorbing games that you prepared without my knowledge is heartwarming. I did enjoy the dinner party we spent together on every Saturday evening, you always made nice food and had a good gossip on trivial matters which brought a lot of fan. Life is not all beer and skittles. I got sentimental when good friends are leaving, but I always believe that absence diminish little passion and increase great one.

My special thanks to Ashvin, Cams, Haishan and Cuiyang. When I first arrive at Enschede, Haishan and Cuiyang helped me a lot to figure out the ropes. Cams and Ashvin, our hearty laughter is testimony of those happiness.

Then, my thanks gives to my Dutch teachers: André, Céline, Carolina, Natasja and all the classmates, for help in improving my Dutch. My thanks

gives to Prof. Liu Lin from Tsing Hua University, who was altruistic in assisting the arrangement of my research funding.

During the last year of my PhD working, I took up with an amazing sport: football. I have to thank all members in Enschede CN Old Boys Football Club, and it was wonderful when we run down the field. My special thanks to our captain (Lu Zhou) for gathering so many football funs together. Thanks Uncle Yin (Tao Yin) for always letting us hitch a ride. Thanks brother Chao (Wang Chao), Xichen, Football King Ma (Ma Yue), Huang He, Fan Yu, Liu Yi for your coaching in improving my techniques. Thanks Wang Yi, Wang Tianpei, Old Sun (Sun Xingwu), Wangyu Lai for your cooperation in our additional training from time to time. These social activities may not have immediate impact on my thesis, but it's truly one of the most beautiful memories during the years.

A special thanks to my family. Words cannot express how grateful I am to my mother, and father for all of the sacrifices that you've made on my behalf. Your love was what sustained me thus far. At the end I would like to express my appreciation to my beloved girlfriend Olivia who should give me a sense of infinite potential, and who should always be my best supporter.

The wonderful experience of today is unprecedented, it's full of possibilities to make our life exactly what we want it to be. Thank you.

# Contents

**1 Introduction** **1**
1.1 Motivation . . . . . 1
1.2 Objectives . . . . . 3
1.3 Research design . . . . . 8
1.4 Scope and non-objectives . . . . . 9
1.4.1 Process interaction failures . . . . . 10
1.4.2 Failure Assumptions . . . . . 10
1.5 Thesis overview . . . . . 11

**2 State of the art** **13**
2.1 Application layer solutions . . . . . 13
2.1.1 Exception handling . . . . . 13
2.1.2 Application implementation language support . . . . . 14
2.2 Infrastructure layer solutions . . . . . 15
2.2.1 Process layer solutions . . . . . 15
2.2.2 Network layer solutions . . . . . 16
2.3 Integration layer: transactions . . . . . 16
2.3.1 Transaction concepts . . . . . 17
2.3.2 Distributed transaction protocols . . . . . 18
2.3.3 Recovery of interaction failures using distributed transactions . . . . . 20
2.3.4 Relation with our research . . . . . 21
2.4 Conclusions . . . . . 22

**3 General concepts and models** **23**
3.1 Collaborative services . . . . . 23
3.2 Shared state types . . . . . 24
3.3 WS-BPEL processes . . . . . 25
3.3.1 Inbound message activity . . . . . 27
3.3.2 Outbound message activity . . . . . 28

3.4 Models of business process: design choices . . . . . . . . . . . . 28
3.5 Petri net models of WS-BPEL processes . . . . . . . . . . . . . . 29
3.5.1 Basic activities . . . . . . . . . . . . . . . . . . . . . . . . . 30
3.5.2 Structured activities . . . . . . . . . . . . . . . . . . . . . 31
3.5.3 Occurrence graphs . . . . . . . . . . . . . . . . . . . . . . 35
3.6 Nested word automata model of WS-BPEL . . . . . . . . . . . . 35
3.6.1 NWA (nested word automata) . . . . . . . . . . . . . . . 36
3.6.2 NWA model of WS-BPEL structured activities . . . . . . 37
3.6.3 NWA model of WS-BPEL basic activities . . . . . . . . . 37
3.6.4 Flattened automata model of WS-BPEL process . . . . . 38
3.7 Conclusions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 39

**4 Recovery of *pending request failure*** **41**
4.1 Pending request failure . . . . . . . . . . . . . . . . . . . . . . . 41
4.2 Pending request failure recovery for shared state type *1 : 1* . . . 43
4.2.1 Recovery on determinate further interaction . . . . . . . 45
4.2.2 Recovery on indeterminate further interaction . . . . . . 49
4.2.3 The robust responder process . . . . . . . . . . . . . . . . 51
4.2.4 The robust initiator process . . . . . . . . . . . . . . . . . 56
4.2.5 Recovery on no further interaction . . . . . . . . . . . . . 56
4.3 Pending request failure recovery for shared state type *n : 1* . . . 58
4.3.1 State determination criteria . . . . . . . . . . . . . . . . . 61
4.3.2 Implementation details . . . . . . . . . . . . . . . . . . . 64
4.4 Pending request failure recovery for shared state type *1 : n* . . . 67
4.5 Pending request failure recovery for shared state type *m : n* . . . 68
4.6 Conclusions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 70

**5 Recovery of *pending response failure*** **71**
5.1 Pending response failure . . . . . . . . . . . . . . . . . . . . . . 71
5.2 Pending response failure recovery for shared state type *1 : 1* . . 73
5.2.1 Pending response failure model . . . . . . . . . . . . . . 75
5.2.2 The robust process model . . . . . . . . . . . . . . . . . . 76
5.3 Pending response failure recovery for shared state type *n : 1* . . 78
5.3.1 The robust initiator process . . . . . . . . . . . . . . . . . 80
5.3.2 The robust responder process . . . . . . . . . . . . . . . . 80
5.4 Pending response failure recovery for shared state type *1 : n* . . 85
5.5 Pending response failure recovery for shared state type *m : n* . . 86
5.6 Conclusions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 88

**6 Recovery of *service unavailable* 91**
6.1 Service unavailable failure . . . . . . . . . . . . . . . . . . . . . . 91
6.2 Service unavailable failure recovery . . . . . . . . . . . . . . . . 93
6.3 Conclusions . . . . . . . . . . . . . . . . . . . . . . . . . . . . 94

**7 Composition of recovery solutions 97**
7.1 Composed solutions: pending request failure and service unavailable . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 97
7.2 Composed solutions: pending response failure . . . . . . . . . . 100
7.3 An example scenario . . . . . . . . . . . . . . . . . . . . . . . . 102
7.3.1 Collaborative processes interaction failure analysis . . . 103
7.3.2 Accounting process transformation . . . . . . . . . . . . 105
7.4 General process design principles . . . . . . . . . . . . . . . . . . 105
7.5 Conclusions . . . . . . . . . . . . . . . . . . . . . . . . . . . . 108

**8 Evaluation 109**
8.1 Correctness validation . . . . . . . . . . . . . . . . . . . . . . . 109
8.1.1 Validation procedure . . . . . . . . . . . . . . . . . . . . 109
8.1.2 Notion of state . . . . . . . . . . . . . . . . . . . . . . . 110
8.1.3 Correctness criteria for state synchronization . . . . . . . 111
8.1.4 Correctness validation . . . . . . . . . . . . . . . . . . . 113
8.2 Performance evaluation . . . . . . . . . . . . . . . . . . . . . . 118
8.3 Business process complexity evaluation . . . . . . . . . . . . . . 121
8.4 Fulfilment of requirements . . . . . . . . . . . . . . . . . . . . . 122
8.5 Sensitivity of our design . . . . . . . . . . . . . . . . . . . . . . 123
8.6 Conclusions . . . . . . . . . . . . . . . . . . . . . . . . . . . . 124

**9 Conclusions and future work 125**
9.1 General conclusions . . . . . . . . . . . . . . . . . . . . . . . . 125
9.2 Research questions revisited . . . . . . . . . . . . . . . . . . . . 126
9.3 Research contributions . . . . . . . . . . . . . . . . . . . . . . . 129
9.4 Future work . . . . . . . . . . . . . . . . . . . . . . . . . . . . 129
9.4.1 Automatic process transformation . . . . . . . . . . . . . 129
9.4.2 General software system interaction failures . . . . . . . 130
9.4.3 Other types of failures . . . . . . . . . . . . . . . . . . . 130

**Bibliography 133**

**Acronyms 143**

**About the author** **145**

# CHAPTER 1

# Introduction

This thesis presents a method to improve the robustness of collaborative services against system crashes and network failures. We investigate possible types of interaction failures caused by system crashes and network failures. We explore how these types of failures occur and their properties: we distinguish different types of state information shared between multiple runtime services instances and possible state inconsistency caused by interaction failures. Based on the above knowledge, we transform the collaborative services into their robust counterparts, which are deployed to the infrastructure where systems crashes and network failures may happen. In order to evaluate the correctness of our method, we develop formal models of the collaborative services, which are evaluated against the proposed correctness criteria. This chapter presents the motivation of this thesis, its objectives and the outline of the research approach.

The chapter is further structured as follows: Section 1.1 motivates the work in this thesis, Section 1.2 outlines our main research objectives, Section 1.3 presents the research design adopted in this thesis, Section 1.4 describes the scope of this work, and finally Section 1.5 presents the structure of this thesis.

## 1.1 Motivation

The electronic collaboration of business organizations has grown significantly in the last decade. By the year 2011, as the world's largest online marketplace, eBay was processing more than 1 billion transactions per day [1], involving different areas such as shopping, trading, checkout, etc. Amazon, the world's largest online retailer, was selling 306 items every second at its peak in 2012 [2] and 426 items in 2013 [3], via a vast collaborations between customers, suppliers, inventory, shipment, payment partners, etc.

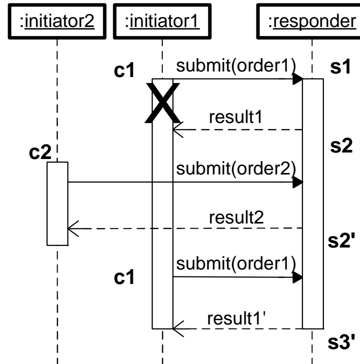



Figure 1.1: A possible failure

Often this electronic collaboration is based on processes run by different parties and exchanging messages to synchronize their states. As an example, AMC Entertainment, who owns the second-largest American movie theater chain, exchanges Electronic Data Interchange (EDI) messages to collaborate with its suppliers, theaters and business partners, who have their own private processes [4].

If a process changes its state, it sends messages to other relevant processes to inform them about this change. For example, after an accounting process has completed an order payment, it sends a shipment message to a logistics process. However, server crashes and network failures may result in loss of messages. In this case, the state change is performed by only one process and not by the other processes, resulting in state/behavior inconsistencies and possibly deadlocks.

System crashes and network failures are common events, which may happen in various information systems, e.g., servers, desktops, mobile devices, etc. In a study of 22 high-performance computing systems over 9 years, the number of failures in a system could reach an average of more than one thousand (1,159) failures per year [5]. In September and October of 2013, mainstream outlets reported iPhone 5s randomly showing a blank blue screen after which reboots occur, as well as random reboots without a blue screen [6].

A possible interaction failure situation is illustrated in Figure 1.1 using simple purchase processes. In these collaborative processes, *initiator1* submits an order, and the system of *initiator1* crashes afterwards. During the failure of *initiator1*, *responder* sends a result message and reaches state $s2$. *Responder* then

```
10:00:51,885 ERROR [ExternalService] Error sending message (
mex={PartnerRoleMex#hqejbhcnphr8fiiltdve4v [PID {http://de.f
hg.ipsi.oasys.businessScenario.sample.initiator}initiator-32
] calling org.apache.ode.bpel.epr.WSAEndpoint@2185a84b.proce
ssInteraction(...) Status ASYNC}): Connection refused: conne
ct
```

(a) Service unavailable

```
15:42:22,154 ERROR [INVOKE] Failure during invoke: No respon
se message received for invoke (mexId=hqejbhcnphr8fj908unbkr
), forcing it into a failed state.
```

(b) Pending response

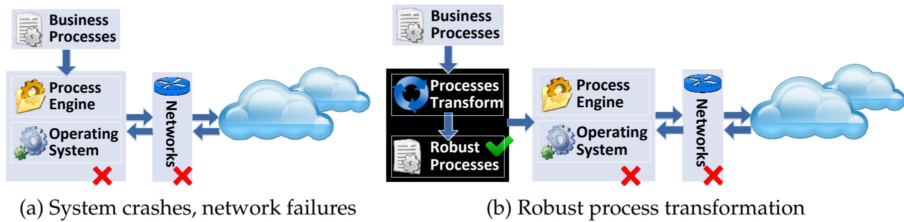
Figure 1.2: Interaction failures

goes to state $s2'$ due to a synchronization with *initiator2* who has also submitted an order. A request is said to be *idempotent* [7] if the operation can be safely repeated. However, the message $submit(order)$ is not idempotent, because the responder changes its state from $s1$ to $s2$ after receiving message $submit(order)$. If it receives the same $submit(order)$ message again, it processes the order and further transits its state from $s2'$ to $s3'$, which is an unwanted state change.

Businesses are deployed to a process engine, which is a piece of software that executes business processes.. In general, state consistency is not detected and recovered by the process engine. This can be seen from a screen dump of errors after a system crash of the Apache Orchestration Director Engine (ODE) process engine [8]. Figure 1.2a shows the case in which the initiator sends the message to an unavailable server. Figure 1.2b shows the case in which the responder receives a request message, and crashes without sending the response message. Recovery in this case often has to be performed manually after checking execution traces, which is potentially slow, error prone and expensive [9, 10].

## 1.2 Objectives

Often services collaboration is based on processes run by different parties and exchanging messages to synchronize their states, e.g., processes described using a language like WS-BPEL [11]. Normally, a business process is deployed to a process engine, which runs on the infrastructure services (operating system, database, networks, etc.), where system crashes and network failures may happen, as is shown in Figure 1.3a. Our objective is to transform business processes into their robust counterpart, as shown in Figure 1.3b. By performing



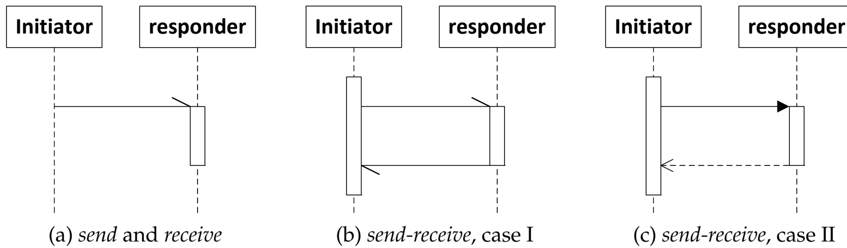

(a) System crashes, network failures (b) Robust process transformation

Figure 1.3: Our objective

process transformations, we apply our recovery principles, e.g., resending the request message, using cached results as a reply. As a result of the transformation, we obtain a robust process, which is able to recover from system crashes and network failures. The robust process is deployed on the same infrastructure services and automatically recovers from interaction failures and state inconsistencies caused by system crashes and network failures. Therefore, our goal is to build robust processes while letting the infrastructure unmodified.

Business process interaction failures are specific to interaction patterns, different types of interaction failures may happen in different interaction patterns. A collection of 13 interaction patterns is discussed in [12]. Generally speaking, interaction patterns can be described from a global point of view, i.e., defined as choreographies. They can also be described from a local point of view, e.g., as abstract interfaces of an orchestration. In this thesis, we assume that each local process involved in an interaction has knowledge of the global view of the interaction but the process designers can only deploy the transformed robust processes to their local process engine (orchestration). In this thesis, we focus on the basic patterns *send*, *receive* and *send-receive* [12]. However, more complex patterns can be composed with basic interaction patterns under a certain control flow, for example, a *one to many send* pattern can be composed by a *send* pattern nested in a loop, e.g., a *while* iteration. Figure 1.4a shows an initiator that sends a message to a responder. The initiator behavior corresponds to the *send* pattern while the responder behavior corresponds to the *receive* pattern. In pattern *send-receive* in Figure 1.4b the initiator combines one *send* and one *receive* pattern, which we call asynchronous interaction in the remaining of the thesis. In Figure 1.4c, the initiator starts a synchronous interaction, which characterize the *send-receive* pattern.

All possible failures in the interaction patterns in Figure 1.4a and Figure




(a) *send* and *receive* (b) *send-receive*, case I (c) *send-receive*, case II

Figure 1.4: Process interaction patterns

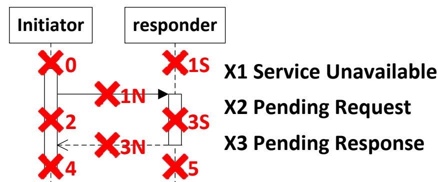



Figure 1.5: Interaction failures

1.4b are represented in Figure 1.4c, These possible failure points are marked as $X_0 ... X_5$ in Figure 1.5. $X_0$, $X_4$ and $X_5$ are system crashes, and these failure points are irrelevant as they have no impact on the interactions. We call failure points $X_1 \sim X_3$ *service unavailable*, *pending request failure* and *pending response failure*, respectively. These failure types are defined as follows.

**Pending Request Failure** The first type of interaction failure is *pending request failure*. We call $X_2$ *pending request failure* since the initiator fails after sending a request message. The failure is informed to the initiator after restart, e.g., through exceptions that can be caught and handled. However, the responder is not aware of the failure, so that it processes the request message, changes its state, sends the response message and continues execution. State inconsistency occurs because the initiator cannot receive this responder's reply and cannot change its state accordingly.

**Pending Response Failure** We call $X_3$ *pending response failure* since the response message gets lost. $X_{3S}$ is a pending response failure caused by a responder system crash. $X_{3N}$ is caused by a network failure. In both cases, the responder sends the response message after restart (in case of a system crash) or after the network connection re-establishment (in case of network failure)

Table 1.1: Interaction failures

| **Interaction Failures** | **Caused by System Crashes** | **Caused by Network Failures** |
|---|---|---|
| Service Unavailable | Failure Point $X_{1S}$ | Failure Point $X_{1N}$ |
| Pending Request | Failure Point $X_2$ | – |
| Pending Response | Failure Point $X_{3S}$ | Failure Point $X_{3N}$ |

and continues execution. However, in both cases the previous established connection gets lost and the initiator cannot receive the response message. The initiator becomes aware of the failure after a timeout. State inconsistency occurs because the responder changes its state after the interaction, but the initiator cannot change its state accordingly.

**Service Unavailable** We call $X_1$ *service unavailable*. Failure $X_{1S}$ is caused by a system crash of the responder, while $X_{1N}$ is caused by a network failure of the request message delivery. However, in both the cases, the initiator is not able to establish a connection with the responder. State inconsistency is thus caused because the responder cannot change its state accordingly. At the process level, the initiator is aware of the failure through an exception at the process implementation level, which can be caught and handled. The interaction failures we focus on in this thesis are summarized in Table 1.1.

Based on the above discussion, we define our research question as follows. **Main research question: How to recover collaborative processes interaction failures caused by system crashes and network failures?**

The question can be further refined as how to transform an original process design into robust counterpart which is recoverable from interaction failures, without putting additional burden to process designers at application level and without putting additional investment to infrastructure. This is a general question that we decompose it into several sub-questions, addressed as follows.

**Research question 1: What are the current existing solutions which can be used to recover from interaction failures?**

This is a knowledge question to make us explore the existing solutions. We need to understand the existing solutions, how are they working, what are the advantages, and what are the shortcomings of these solutions. This question is mainly discussed in Chapter 2.

**Research question 2: What are the necessary concepts/models in our solution?**

A recovery solution should be implementable using existing technologies.

Furthermore, the recovery solution should be formally presented that forms a basis for correctness validation. Then the question is raised that what are the technologies and models we use in our solution. This question is mainly presented in Chapter 3.

**Research question 3: What are the corresponding behavior and recovery approach for the interaction failures?**

The above research question are all knowledge questions from which we learn the related solutions, related models and necessary techniques. This question is the design science question that the interaction failures and their properties should be identified and for each type of interaction failure, what are their corresponding recovery approaches. This question is mainly presented in Chapters 4, 5 and 6.

**Research question 4: How to combine the recovery solutions for different approach?**

Multiple types of interaction failures may happen in one business process. This raises the question whether it is possible to combine the solutions to make the robust process recoverable from different interaction failures. This question is mainly presented in Chapter 7.

Since we present a solution at process language level, the research work addresses the following requirements:

- Requirement R0: The solution should be correct. The robust process should recover from the interaction failures.
- Requirement R1: The process transformation should be transparent for process designers. The complexity of process transformation should not distract process designers from the functional aspects of the process design.
- Requirement R2: The transformed process should not require additional investments in a robust infrastructure.
- Requirement R3: As a solution at process language level, the process interaction protocols should not be changed. For example, the message format cannot be changed, e.g., by adding message fields like message sequence numbers that are irrelevant for the application logic. The message order should not be changed either, e.g., by adding acknowledge messages to the original message sequence.
- Requirement R4: The service autonomy should be preserved. Services exposed by business processes allow flexible integration of heterogeneous

systems [11]. Thus it is required that if one party transforms the process according to our approach and the other party does not, they can still interact with each other, although without being able to recover from system crashes and network failures.

- Requirement R5: Only available standard process language specifications could be used. The existing process language specification should be used without extensions, and the robust process should be independent of any specific engine.

- Requirement R6: The solution should have acceptable performance.

## 1.3 Research design

The research design [13, 14] adopted in this thesis has three phases, namely problem investigation, solution design and solution validation, as is shown in Figure 1.6.

We started from problem investigation, which includes literature study of related research work, e.g., exception handling, transactions, WS-Reliability and HTTPR. After performing the literature study, we defined our research questions based on an analysis of possible interaction failures caused by system crashes and network failures.

The second step is the solution design. Based on the research topics identified in the previous step, we defined general concepts and models, which forms a basis of the recovery solutions and validation.e.g., models of workflow control and data dependencies. Then we worked on the solutions of the general research question using the defined concepts and models. The major research work has been done in this step, namely by developing solutions for the research problems proposed in the previous step.

Finally, we validated the research work. We proposed correctness criteria and show the correctness of the proposed transformations based on these criteria. We implemented a prototype and evaluated its runtime performance, and we analyzed the complexity of the process transformation by comparing process complexity measures before and after the transformation.

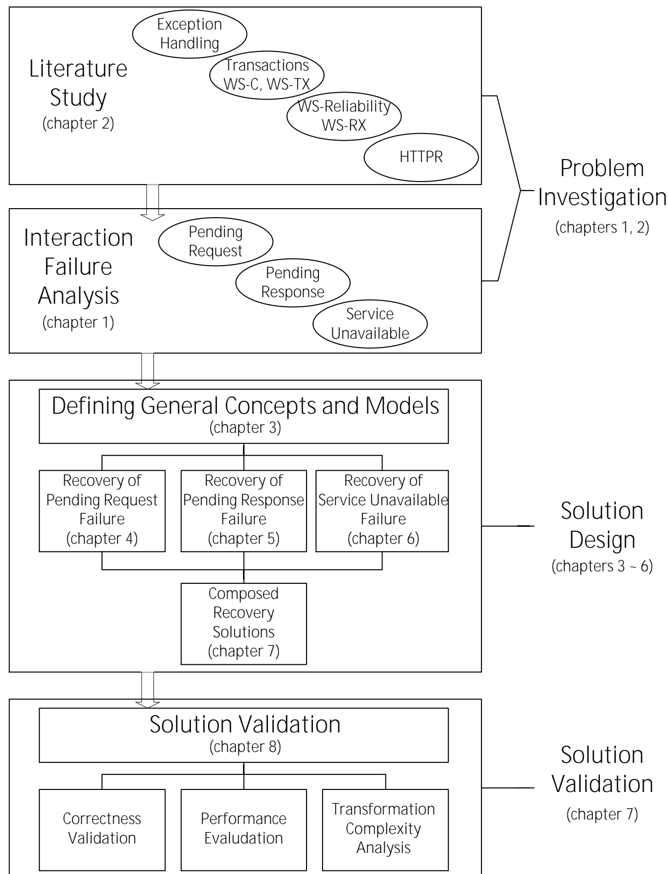



Figure 1.6: Research design

## 1.4 Scope and non-objectives

The types of interaction failures that are caused by systems crashes and network failures are discussed in this section. We define the failure properties and make some assumptions of failure behaviors in this section.

Table 1.2: Failure scheme

| | Type of failure | Description |
|---|---|---|
| Inside Scope | Crash failure | A server halts, but is working correctly until it halts. |
| | Omission failure | A server fails to respond to incoming requests. |
| | Receive omission | A server fails to receive incoming messages. |
| | Send omission | A server fails to send messages. |
| Outside Scope | Timing failure | A server response lies outside the specified time interval. |
| | Response failure | A server response is incorrect. |
| | Value failure | The value of the response is wrong. |
| | State transition failure | The server deviates from the correct flow of control. |
| | Arbitrary failure | A server may produce arbitrary responses at arbitrary times. |

### 1.4.1 Process interaction failures

Table 1.2 shows a failure classification scheme [7]. Crash failure, omission failure and timing failure are in our research scope. Crash failure is referred as *system crashes* in this thesis. Omission failure and timing failure occur when the network fails to deliver messages (within a specified time interval) and are referred as *network failures* in this thesis. However, response failures due to flaws in the process design, e.g., incompatible data formats, and arbitrary failure, also referred to as Byzantine failure, which is more of a security issue, are out of the scope of this work. The following process design errors are also out of the scope of this thesis: process control flow errors (deadlocks), message duplication or sequence errors caused by incorrect design of process interaction protocols. Since we focus on system crashes and network failures, we left those process design errors or security concerns out of the scope of this thesis.

### 1.4.2 Failure Assumptions

Due to the heterogeneous infrastructure, e.g., different process engine implementations or network environment, different levels of robustness are achieved

by different process execution environments, thus it is necessary to make consistent assumptions concerning failure behaviors of the infrastructure. These assumptions are discussed below.

**System crashes**

- Persistent execution state. The state of a business process (e.g., values of process variables) can survive system crashes.
- Atomic activity execution (e.g., invoke, receive, reply). Since a system crash causes the execution to stop in a friendly way, it is fair to assume that the previous activity is finished and the next activity has not started. A restart resumes the execution from the previous stopped activity.

These are reasonable assumptions because it is the default behavior of the most popular process engines, such as Apache ODE [8] and Oracle SOA Suite [15]. In Apache ODE's term, the *persistent processes* is in its default configuration. Otherwise this configuration can be modified to *in-memory* at deployment time [16]. For Oracle BPEL Process Manager, this is named as *durable* processes, otherwise is named as *transient* processes. By default all the WS-BPEL processes are durable processes and their instances are stored in the so called dehydration tables, which survives system crashes [17].

**Network failures**

The commonly used service messages are HTTP messages (SOAP or REST) over TCP connections. HTTP normally uses the same TCP connection for the request and response messages of the interaction pattern in Figure 1.4c. Therefore network failures interrupt the established network connections, so that all the messages that are in transit at the point of a failure get lost.

## 1.5 Thesis overview

The remainder of this thesis is structured as follows. Chapter 2 discusses the related solutions and their advantages and disadvantages. A robust process execution environment includes process engines, operating systems, database and networks, etc. We discuss solutions at different layers and their relationship with our solutions. Chapter 3 defines the general concepts and models, e.g., the model of business process using Petri nets and Nested Word Automatas

(NWAs), and the data and control flow dependencies. Chapter 4 proposes our solution for the pending request failure, which means that the initiator system crashes after sending the request message without receiving the response. The basic idea is to resend the request message and use the previous result as a response to avoid duplicate processing. Chapter 5 proposes our solution for the pending response failure, which is the case where the responder system crashes after receiving the request without sending the response or the network fails to deliver the response message. The basic idea is to split the receiving the request message and the sending of the response to avoid the impact of the failure on the response message delivery. Chapter 6 proposes our solution for the service unavailable failure, which means that responder crashes before receiving the request message or the network fails to deliver the request message. The idea is to resend the request message from the initiator side. Chapter 7 presents the composed solutions of different types of interaction failures. Chapter 8 evaluates our solutions, in terms of the correctness and the performance overhead and additional complexity are evaluated. Chapter 9 concludes this thesis and identifies some research topics for further investigation.

CHAPTER 2

# State of the art

A typical implementation of a collaborative services execution environment is shown as Figure 2.1 [18, 19]. A Web Services Business Process Execution Language (WS-BPEL) process is designed and implemented at *application layer*. Then it is deployed on the *infrastructure layer*, where the process gets executed and managed. The *integration layer* implements the interaction of business process with other services via the network. Building robust collaborative services interactions involves the efforts of the *application layer*, *infrastructure layer*, and *integration layer*.

The related solutions of robust process interactions can be found at different layers, which are discussed as follows. Section 2.1 discusses related solutions mainly on the application layer, in which robust collaborative services are designed with the support of the implementation language. Section 2.2 discusses the infrastructure layer solutions, which are placed in process engine, operating system and networks. Finally, section 2.3 discusses the transactional approach and section 2.4 concludes this chapter.

## 2.1 Application layer solutions

At application layer, business processes are implemented using specific process implementation languages. One possible way of building robust processes is to make use of the possible support of process implementation languages.

### 2.1.1 Exception handling

In the context of programming languages, an exception is raised whenever an operation should bring to the attention of its invoker source code, and by

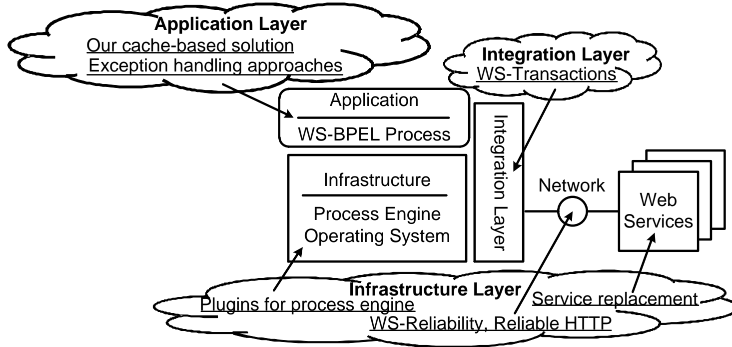



Figure 2.1: Overview of Related Solutions

handling an exception the invoker reacts to the exception [20]. The exception hanlding features of programming languages are described in [21, 22].

In the context of business process, at application layer, they are implemented by process execution languages. The process language facilities for exception handling is discussed in [23, 24, 25], amongst others. Unlike programming languages that exceptions can be defined for events such as divide by zero errors and appropriate handling routines can be defined. For business processes, this level of detail is too fine-grained and it is more effective to define exceptions at a higher level, typically in terms of the business process to which they relate. In general, exception handling require that the process designers are aware of faults and their recovery strategies [26]. Alternatively, our process transformation based solutions can be transparent to process designers in the way that we do not put the burden of building robust processes to process designers.

### 2.1.2 Application implementation language support

Another solution is to assign the ability of recovering from failures to the existing programming languages, which can be used to implement collaborative services. In [27], WS-BPEL is extended with annotations. Process designers can use these annotations to specify recovery related operations in process design. In [28, 29] an extension is added to C++, LISP and Ada to support the recovery from failures. In [30, 31], a C++ extension with the transactional properties are added in to the programming language that can be used in interaction fail-

ure recovery. In these references, the explicit client or server *abort* or *commit* is supported by extended APIs to the original language. By implementing a few basic classes with the properties of persistency or atomicity, these programming languages provide the process designers the support to design robust services at implementation language level. For example, if a class inherits from a pre-defined atomic class and contains a few recoverable operations, and a recoverable operation can be aborted by one party (client or server), the data is restored like if the operation were not executed at all. The local data recovery is implemented by combining of a few technologies, e.g., storage replication, logging, data versioning and/or timestamping [32, 33, 34, 35], Local consistency is met by changing the data from one consistent state to another, i.e., by guaranteeing the transactional property of atomicity and persistency. However, in a distributed scenario, how the mutual consistent state is automatically synchronized between client and server is not clearly specified in the languages support [28, 29, 30, 31], which is left as a burden to the process designers. Even an execution should not be aborted before completion, the process designers have to design the collaborative interaction protocol to make a crash party, after a restart, coordinate the mutual execution state in other collaborative services .

## 2.2 Infrastructure layer solutions

Infrastructure layer solutions include the solutions placed in process engine, operating system or networks.

### 2.2.1 Process layer solutions

Infrastructure layer solutions include [36, 37, 38, 39]. Recovery mechanisms implemented as plug-ins for a WS-BPEL engine is presented in [36, 37]. The approach to recovery presented in [38, 39] consists of substituting a service with another one dynamically if a synchronization error occurs. In [40, 41, 42], the QoS aspects of dynamic service substitution are considered. In all these solutions, the idea is to build the recovery capabilities in the process engine.

The advantage of these solutions is to lower the burden of process designers. With no or little extensions on the process language, the process designers are freed from the recovery details. However, the solutions strongly depend on a specific WS-BPEL engine. As the solutions mainly implemented at engine level, the solutions is engine specific, which makes the process difficult to migrate to other process engines.

### 2.2.2 Network layer solutions

Message exchange is realized at the network level using standard communication protocols like HTTP (on the TCP/IP protocol stack). However, HTTP does not provide reliable messaging. A solution to avoid the loss of state synchronization is to use reliable messaging. Reliable messaging protocols such as HTTPR [43], WS-RX [44] solve the problem by introducing a middle layer, where robust interaction protocol can be built. The basic idea behind these protocols is to re-send resend lost message.

The advantage of these solutions is that they put litter burden to the process engine implementation and process design. However, this solution increases the complexity of the required infrastructure. We assume that server crashes and network failures are rare events, and therefore extending the infrastructure introduces too much overhead. Further, adding a middle layer could turn out to be a problem for some outsourced deployments where the infrastructure layer is out of control of the process designer. For example, in some cloud computing environments, user-specific network configuration capabilities to enhance state synchronization are not available. Another possibility is to design the process to deal with unreliable messaging, which makes the process design and the created model much more complicated.

## 2.3 Integration layer: transactions

The transaction concept derives from contract law [45]. The concept of transaction in computer science originates from database management systems (the *transaction* concept is used in [46, 47, 48]). In the database context, a transaction is an execution step of a program that accesses a database [49]. Transactions were introduced in distributed systems in the form of transactional file servers, such as CFS and XDFS [50]. In a transactional file server, a transaction is the execution of a sequence of client requests for file operations. Transactions of distributed objects are implemented as a inherent of programming languages, e.g. Argus [51, 52, 53, 54]. In CORBA, a language independent transactional interface was proposed by OMG [55] to provide standardized transitional interface for distributed objects. In service collaboration context, transactional recovery approaches are based on the OASIS WS-AT [56], WS-BA [57] and WS-C [58] standards. In general, all these kinds of transactions share common properties that form a basis of building robust interactions with regards of system crashes and network failures Transactions are discussed in more detail below.

### 2.3.1 Transaction concepts

At the application layer, the transactional capabilities are exposed to clients as a few operations, such the SQL-transaction defined in the ANSI standard [59], with the following semantics:

1. *transaction start*. The operations of this kind are the explicit start of a transaction control boundary. The interaction messages (in distributed transactions) or local procedure invocations (in local transactions) that follow is in context of this transaction implicitly, or explicitly by passing the transactional identifier with the messages. Whichever way depends on the specific implementation.

2. *transaction commit*. This type of operations indicates the successful execution of a transaction.

3. *transaction abort*. This operation indicates the unsuccessful execution of a transaction. The reason of a transaction abortion includes failures, exceptions, client cancellation, etc.

The properties supported by the above APIs are Atomicity, Consistency, Isolation, Durability, represented an acronym ACID [60], described as follows.

- *Atomicity*. A transaction must either be executed in its totality or not at all. After a *transaction start*, either *transaction commit* or *transaction abort* happens. In the latter case all the intermediate effects of a transaction should rolled back to the start state of the transaction.

- *Consistency*. A transaction takes the system from one consistent state to another consistent state. The criteria for the state consistency is application-specific. However, after a *transaction commit*, a transaction should meet all the consistency criteria defined for the application.

- *Isolation*. Any intermediate results between *transaction start* and *transaction commit* should not be revealed. This is due to the consideration of concurrency control. For example, if multiple transactions execute concurrently, the intermediate result could be rolled back due to a transaction abortion. If other transactions depend on the intermediate results of this transaction and have committed, the system reaches a inconsistent state, since a committed transaction is not recoverable.

- *Durability*. The result of a transaction should be persisted in stable storage. This property is twofold. First, the result of a transaction should survive crashes or storage failures. Second, the result of a transaction cannot be modified after it is committed.

These transaction properties have two major concerns: first, when multiple transactions execute concurrently, if they update the shared state of the system, they should not interfere with each other [61]. Second, transactions are resilient of failures. The latter property is relevant to our work.

**Relaxing ACID properties**

The transactions introduced above is called flat transactions. However, some of the properties discussed above can be relaxed. For example, the atomicity of a transaction can be relaxed by introducing the concept of nested transaction [62]. Furthermore, the isolation property can be relaxed, e.g., by introducing the concept of open nested transactions (sagas), as defined in [63, 64]. A nested transaction can include a few sub-transactions, thus nested transactions are organized in a corresponding tree structure. The execution and commit rules of nested transactions are described as follows:

- Sub-transactions that have the same parent can execute in parallel to improve the performance of transaction execution.

- A parent transaction can commit even if a few of its sub-transactions have aborted.

- If a parent transaction aborts, all its sub-transactions have to abort as well.

Transactions can be classified as short-life and long-life respectively [65]. A few other transaction variations are discussed in [66].

### 2.3.2 Distributed transaction protocols

Unlike local transactions where ACID properties need to be met locally even when failures happen, distributed transactions involves several parties and a protocol is required to achieved mutual consistency. The distributed transaction protocols form a basis for the recovery from interaction failures. The two-phase commit protocol is one of the most famous distributed transaction protocols.

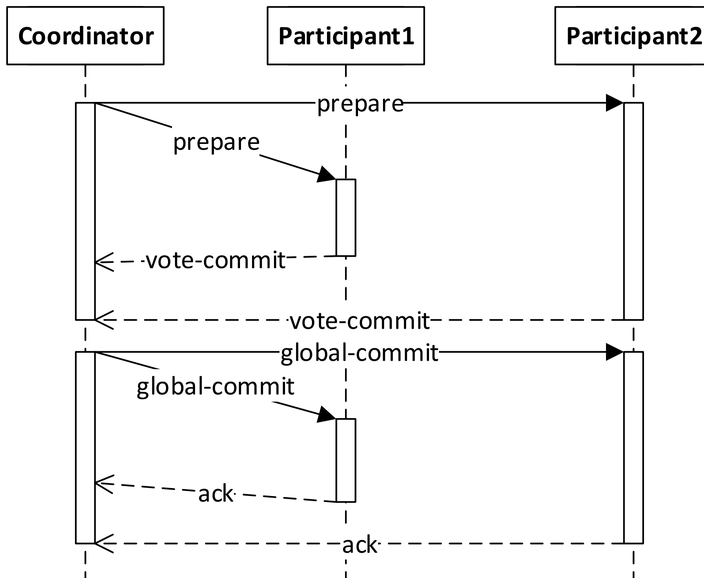



Figure 2.2: Two-phase commit protocol, commit

**Two-phase commit protocol**

The 2PC (Two-Phase-Commit) protocol [67, 68] is brief illustrated in Figure 2.2 in a UML sequence diagram for two participants and a coordinator [69].

The successful commitment to a transaction is divided into two phases:

1. The coordinator sends a *prepare* message to all participants. If all participants finish the transaction without any failure, they send back the *vote-commit* message to the coordinator.

2. The coordinator sends a *global-commit* message to all participants to indicate the success of the transaction. All participants sends back an *ack* message to end the transaction.

In the case any participant wants to abort the transaction, the sequence diagram is as shown in Figure 2.3. This is similar to Figure 2.2, but in this case an abort message is sent.

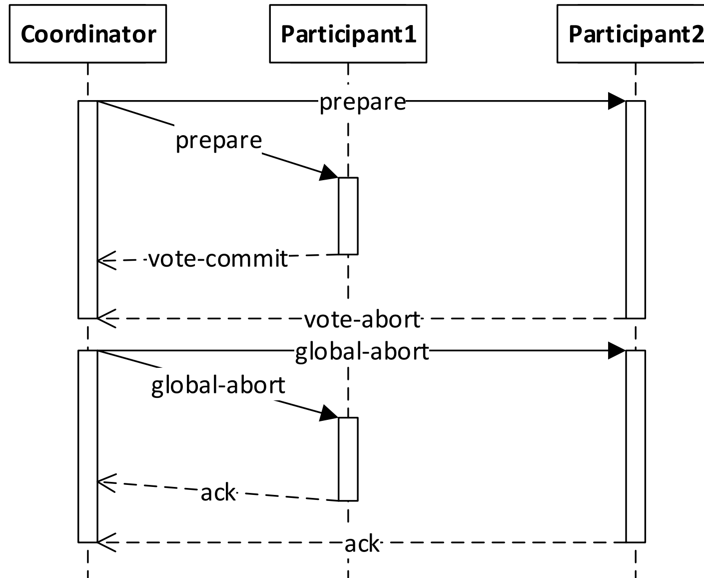



Figure 2.3: Two-phase commit protocol, abort

### 2.3.3 Recovery of interaction failures using distributed transactions

#### Transaction failure model

[70, 71] presents a failure model that a transaction is able to recover from. The failures modeled are *imperfect disk storage*, *processors failures* and *unreliable communication*. The *processors failures* are referred in this thesis as *system crashes*. The computer system works exactly as expected until it halts. The *unreliable communication* is named as *network failure* in this thesis.

#### Recovery using distributed transaction protocols

The various cases of system crashes and network failures of two phase commit protocols and their recovery methods are discussed in [72].

One example is shown as Figure 2.4, in which *participant2's* system crashes after receiving a *prepare* message. After a timeout waiting for *participant2*'s response, the coordinator sends a *global-abort* message to all other participants

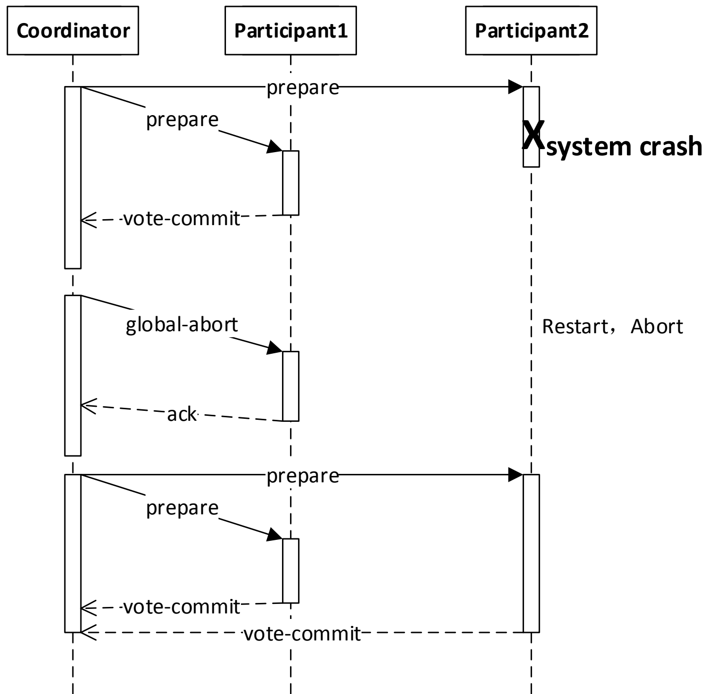



Figure 2.4: Two-phase commit protocol, system crash recovery

to abort the transaction. The *participant2* abort all uncommitted transactions after a restart.

The coordinator may restart the transaction by re-sending *prepare* message to all participants.

### 2.3.4 Relation with our research

Other types of failures, e.g., message format or content error, process design flaws (deadlocks), may result in the abort of a transaction. A transaction can also be aborted by any participant without any failure. Therefore, transaction mechanism can be used to recover more generalized types of failures. However, the 2PC transaction protocol is centralized so that not all cases of failures are recoverable. In a special case that all participants have send their vote deci-

sions to commit or abort to a coordinator and the coordinator crashes without sending any global decision message, the participants cannot know the result of the transaction. In this case, the fate of the transaction will not known and all participants will be blocked. The more complex 3PC protocol can recover from all cases of system crashes and network failure, however, this protocol is with more network latency [73]. The application of the transactional mechanisms is not transparent to application programmers, i.e., the transaction is an application level concept that the application programmers should be aware of possible interaction failures and their recovery protocols based on the application of transactions. In contrary, our research objective is to build an robust business process from the original process design transparently, without bothering the application programmers.

## 2.4 Conclusions

A business process execution environment is often built up with multiple abstraction layers, namely application layer, infrastructure layer and integration layer. Interaction failure solutions can be found at each of the layers.

Application layer solutions make use of the application programming languages support, such as exception handling features and transactional features. However, these solutions require that the programmer is aware of all possible failures and their recovery strategies.

Solutions at infrastructure layer are transparent to application programmers. However, normally these solutions require more infrastructure investment, e.g., more reliable communication channels. We assume system crashes and network failures are rare events that make additional infrastructure support expensive. Furthermore, these solutions may make the implementation specific to process engine, which make the business process difficult to migrate between different process engines.

We can conclude there is a need for a solution that is transparent to process designers and require little infrastructure investment.

CHAPTER 3

# General concepts and models

This chapter introduces the general concepts and basic terminology used throughout this thesis. Firstly, the concept of collaborative service, especially, the concept of collaborating business process with web services is explained. Secondly, service collaboration is based on shared state information, and for that purpose we present an overview of shared state types. Thirdly, we introduce the main concept of Web Services Business Process Execution Language (WS-BPEL), which is a standard executable language for specifying business processes with web services. WS-BPEL is used to illustrate our solutions, which can be applied to other similar languages. Finally, we introduce the formalisms we used in our solutions, namely, Petri nets and Nested Word Automata (NWA) to represent WS-BPEL processes for the purpose of enabling their analysis and manipulation.

This chapter is structured as follows. Section 3.1 introduces the collaborative services addressed in this thesis. Section 3.2 analyze the service state types, i.e., how state is shared among multiple services and their runtime instances. Section 3.3 introduces WS-BPEL, which is a business process execution language used to illustrate our solutions. Section 3.5 presents the Petri net model of collaborative services. Finally, section 3.6 defines the NWA model of a WS-BPEL process.

## 3.1 Collaborative services

The term *service* used in this work denotes a *web service* [74], where technical level interaction is our focus. We adopt the web services definition inspired by World Wide Web consortium[75]: *A Web service is a software system designed to support interoperable machine-to-machine interaction over a network.* This is a broad concept that many technologies match.

Table 3.1: State information types, client and server's viewpoints

| Client perspective $\underline{C}$ : S / Server perspective C : $\underline{S}$ | Each client instance interacts with 1 server instance | Each client instance interacts with variable number of server instances (n) |
|---|---|---|
| Each server instance interacts with 1 client instance | $\underline{1}$ : 1 / 1 : $\underline{1}$ | $\underline{1}$ : n / 1 : $\underline{1}$ |
| Each server instance interacts with variable number of client instances (m) | $\underline{1}$ : 1 / m : $\underline{1}$ | $\underline{1}$ : n / m : $\underline{1}$ |

Table 3.2: State information types, combined viewpoint

| Shared state types | $\underline{1}$ : 1 | 1 : $\underline{n}$ |
|---|---|---|
| 1 : $\underline{1}$ | 1 : 1<br>Figure 3.1c | 1 : n<br>Figure 3.1b |
| m : $\underline{1}$ | m : 1<br>Figure 3.1b | m : n<br>Figure 3.1d |

In this thesis, the collaborative services are characterized as the collaboration of two or more (automated) processes through the use of each other's services. In particular, this thesis is limited to collaborative processes with web services.

## 3.2 Shared state types

At runtime, a stateful process has multiple instances, so that each instance maintains its own state information, e.g., the value of process variables, or the history of interactions [76]. We use a simple vocation request process [77] to illustrate the concept of process instance. The *business process* refers to the entire vacation request process design, beginning when an employee asks for vacation, and ending with the approval and reporting of that vacation. Consequently, the term *process instance* refers to that employee's single request for a leave of absence, and *instance management* (also named as *case management*)

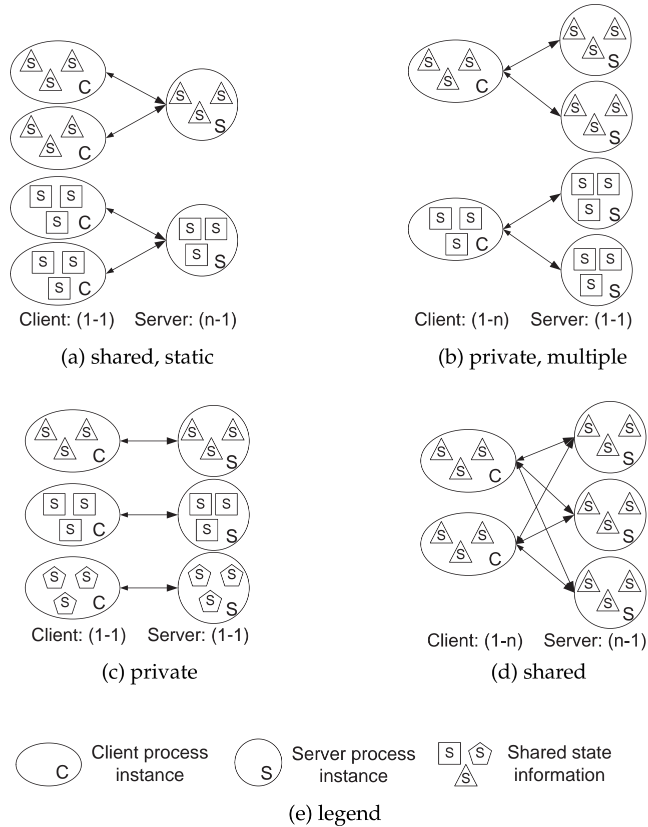
would refer to the management of each vacation request. When a employee makes a new vacation request, that request generates a new process instance (case) in the process engine, that subsequently moves through the business process according to the process design.

If an instance changes its state, it may send messages to other relevant instances to synchronize their states. Thus, state information is propagated and "shared" implicitly between multiple process instances. Although the client instance interacts with the server and is not aware of server instances. How state information is shared [78] depends on the service interaction patterns [79] of the client and server processes. As shown in Figure 3.1, from the client's point of view, one client instance can interact with one server instance (1-1) or with many server instances (1-n). From the server point of view, one server instance can interact with one client instance (1-1) or with many client instances (n-1). From a global point of view, we distinguish the combination types as shown in Table 3.2, and illustrated in Figure 3.1.

In Figure 3.1 (a), the state information is shared between clients. One client instance interacts with one server instance (1-1), while globally one server instance interacts with multiple client instances (n-1). The number of server instances is static in the sense that it could be one or more, but it is a fixed number at runtime. We call this state information type $n : 1$ *shared state*. In Figure 3.1 (b), the state information is private to each client instance, but shared between multiple server instances, since each client instance interacts with multiple server instances (1-n), and each server instance interacts with one client instance (1-1). We call this state information type $1 : n$ *shared state*. In Figure 3.1 (c), the state information is private to the requester-responder pair, since each initiator process instance is dedicated to synchronize its state with a single responder instance. We call this state information type $1 : 1$ *shared state*. In Figure 3.1 (d), the state information is shared between all instances, since each client instance interacts with multiple server instances (1-n), and each server instance interacts with multiple client instances (n-1). We call this state information type $n : n$ *shared state*.

## 3.3 WS-BPEL processes

In order to describe the collaborative behavior of web services, a standard language is required to implement complex interactions and control flow, i.e., to orchestrate the web services. In this thesis, we choose WS-BPEL as the collaborative services description language. A WS-BPEL process is a container where




(a) shared, static

(b) private, multiple

(c) private

(d) shared

(e) legend

Figure 3.1: Shared state types

relationships to external services, process data and handlers for various purposes and, most importantly, the activities to be executed are declared. As an OASIS standard [11], it is widely used by enterprises.

WS-BPEL activities perform the process logic. Activities are divided into 2 classes: basic and structured. Basic activities are those which describe elemental steps of the process behavior. Structured activities encode control-flow logic, and therefore can contain other basic and/or structured activities recursively. The complete WS-BPEL specification is available at [11].

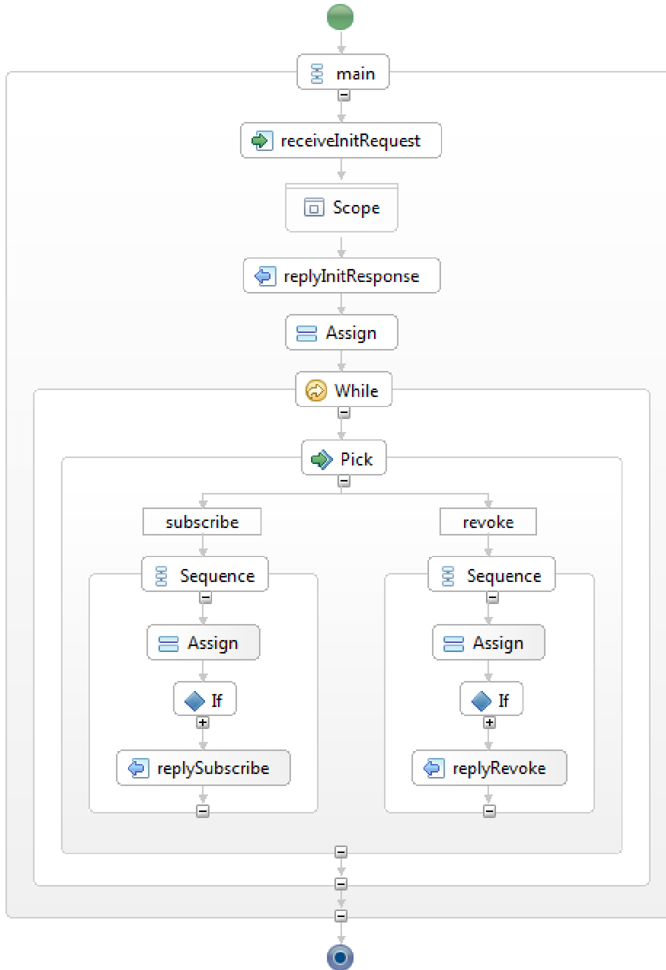



Figure 3.2: An example WS-BPEL process with Eclipse WS-BPEL editor

### 3.3.1 Inbound message activity

An Inbound Message Activity (IMA) of a WS-BPEL process is an activity in which messages are received from partner services. In this work we consider

the inbound message activities *receive* and *pick*, while other types of IMAs, like *event handlers*, are out scope of this thesis.

### 3.3.2 Outbound message activity

An Outbound Message Activity (OMA) of a WS-BPEL process replies the response message. In this work we consider the outbound message activities *invoke* and *reply*.

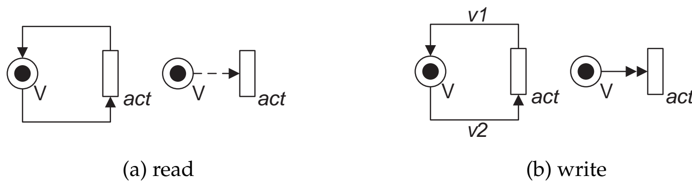
IMAs and OMAs correspond to the begin and end of the control boundary of a synchronous operation, respectively. As an example, in Figure 3.2, which is graphical representation produced with Eclipse WS-BPEL editor [80], the IMA "receiveInitRequest", which is a *receive* activity, is the begin of a synchronous operation, while the OMA "replyInitResponse", which is a *reply* activity, is the end of this operation. The IMA "Pick" , which is a *pick* activity, is the begin of multiple process operations, namely "subscribe" and "revoke", while the OMA "replySubscribe" and "replyRevoke", which are *reply* activities, marks the end of these operations respectively.

## 3.4 Models of business process: design choices

Formal models of business process eliminate ambiguity in process specification and enable a rigorous for analysis [81]. Furthermore, a formal model make our solution independent of any specific process design language or vendor implementation of process engines.

We choose Petri nets and Nested Word Automata (NWA) as our process formalisms. The models of Petri nets are used for correctness validation. The other purpose is to infer data dependencies of business process, which is used to detect if there is possible state change caused by interactions. We choose Petri nets because in contrast with some other process modeling techniques, the state of a process instance is modeled explicitly in a Petri net [82], by the distribution of tokens over places. By simulating of a Petri net, an occurrence graph can be generated, which can be mapped to an equivalent automata model and be used to represent all possible states and transitions of the Petri net.

By using NWA is used to infer all possible further incoming messages, where the recovery of pending request can be based. We choose NWA because the structural information concerning process hierarchies can be maintained. For example, in the syntax of WS-BPEL process contains the structure information




(a) read (b) write

Figure 3.3: Convention for reading and writing of WS-BPEL process variables

that one activity is nested in another structured activity. These structure information is necessary if we want to map these formalisms to a specific process language with a hierarchical structure.

## 3.5 Petri net models of WS-BPEL processes

This section presents our Petri net model of WS-BPEL processes in which the dataflow is also annotated. WS-BPEL models using Petri nets have been reported in the literature, however, each approach has its particular focus. For example, [83] focuses on control flow modeling, thus state information is implicit. [84, 85, 86] address activity stops and correlation errors, which are not relevant in this work and cause the formalism is unnecessarily complex for our purposes. Thus, we propose a simplified Petri nets representation, in which the Petri net structure of each WS-BPEL activity has one start place and one sink place. The net structure of each activity can be nested or concatenated with the structure of other activities, which is the semantics of WS-BPEL structured activities.

This Petri nets model is not a functional model for WS-BPEL processes which is used to support process design or implementation. Its purpose is to allow the inference of data dependencies and control flow dependencies based on an existing business process. In order to improve readability, we use the two conventional notations to denote Petri net models of the reading and writing behavior, respectively, of process variables by activities. Figure 3.3 (a) shows the Petri net representation of an activity reading a process variable $V$ in which a transition takes a token from the place that represents the variable and then puts a token back. We use a dashed arrow as a graphical notation for this. Figure 3.3 (b) shows the Coloured Petri Net (CPN) representation of an activity writing a process variable $V$ in which a transition takes a token $v1$ out from the

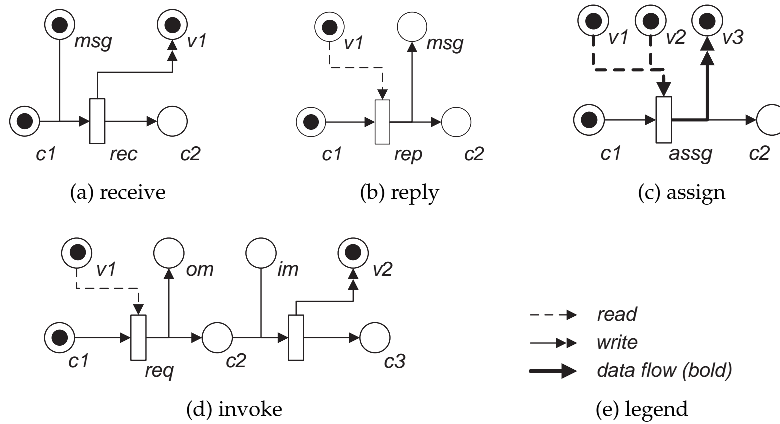



Figure 3.4: The Petri net model for basic activities

place that represents the variable and then puts another token $v2$ into it. We use a double arrow as a graphical representation for this. The values $v1$ and $v2$ is not relevant in our work and are omitted in the Petri net representation of writing a process variable.

WS-BPEL activities are divided into two categories: basic and structured activities. Each category is discussed in the sequel.

### 3.5.1 Basic activities

The basic activities supported in this thesis are: *receive*, *reply*, *assign* and *invoke*. Figure 3.4 (a) shows the Petri net representation of a *receive* activity, where places *c1* and *c2* are the input and output control places, respectively. In order to express the *receive* semantics of WS-BPEL, the transition takes a token out from the *msg* place and "writes" to the place *v1*. Similarly, we have modeled basic activities *reply*, *assign*, and *invoke* as shown in Figure 3.4 (b), Figure 3.4 (c) and Figure 3.4 (d), respectively.

We denote data flow as a set of the arcs annotated in bold. The data flow of the *assignment* activity (bold arcs in Figure 3.4c) is from place *v1* (and *v2*) to the transition *assg*, then to the place *v3*.

```
<if>
  <condition>boolean_expression($v1, $v2)</condition>
  <!-- body_true -->
  <else>
    <!-- body_false -->
  </else>
</if>
```

Figure 3.5: The WS-BPEL code of an *if* activity

### 3.5.2 Structured activities

The structured activities supported in this thesis are: *if*, *while*, *pick*.

The Petri net representation of an *if* activity is presented in Figure 3.6, where the corresponding WS-BPEL code is shown as Figure 3.5. The places $c1$ to $c6$ model the control flow. In WS-BPEL, the condition of an *if* activity is a boolean expression, such as $\$v1 < \$v2$. The process variables that appear in the condition expression are modeled as places $p_v1$, $p_v2$ in our Petri nets. The positive (negative) evaluation of the condition results in the execution of the *true* (*false*) branch of the WS-BPEL process, which is modeled as a hierarchical transition *body_true* (*body_false*), and is initialized by firing transition *cond_true* (*cond_false*). In the Petri net model, the transitions *cond_true* and *cond_false* “read” the places $p_v1$ and $p_v2$. A token in the place *in_true* (*in_false*) represents that the modeled WS-BPEL executes the *true* (*false*) branch. We name this place *dependency indication place*. The Petri net of conditional expression does not model the actual evaluation of conditional expression.

The data flow (denoted as bold arcs) starts from the “reading” of places $p_v1$ (and $p_v2$) by the transition *cond_true* (*cond_false*), to the dependency indication place *in_true* (*in_false*). The evaluation of values of variables in a condition determines the variables that are changed, because it determines the branch to be chosen. Thus the process variables changed inside of the *if* branches should depend on the conditional variables. We model this as a “read” of the dependency indication place by the assignment transition that is hierarchically nested in the *if* construct. This is illustrated in Figure 3.7, which shows a *true* branch of an *if* activity. The corresponding WS-BPEL code is shown as Figure 3.8. The transition *assg* is the Petri net representation of an assignment activity. The data flow generated by the *if* activity model is the path from conditional variable $p_v1$ (and $p_v2$) to the transition *cond_true*, and then from the transi-

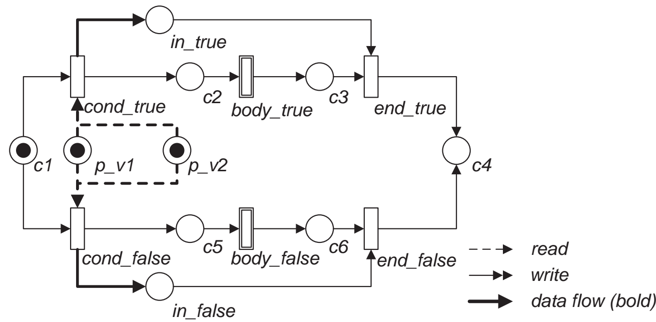



Figure 3.6: The Petri net model for *if* activity

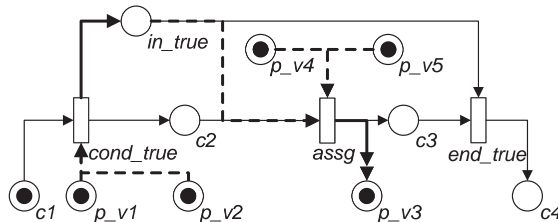



Figure 3.7: The data flow path of *if* activity

tion *cond_true* to the dependency indication place *in_true*. The data flow model generated for the assignment activity is from the places $p_v4$ and $p_v5$ (representing the process variables $v4$ and $v5$, which appear in the right hand side of the assignment) to the transition *assg*, and then from *assg* to the place $p_v3$ (representing the process variable $v3$, which appears in the left hand side of the assignment). By the application of the rule, we add a "read" of the indicator place *in_true* by the transition *assg*, so that the data dependency path represent-

```
<if>
  <condition>boolean_expression($v1, $v2)</condition>
  <assign>$v3 := $v4 + $v5</assign>
</if>
```

Figure 3.8: WS-BPEL code of an illustrative *if* activity

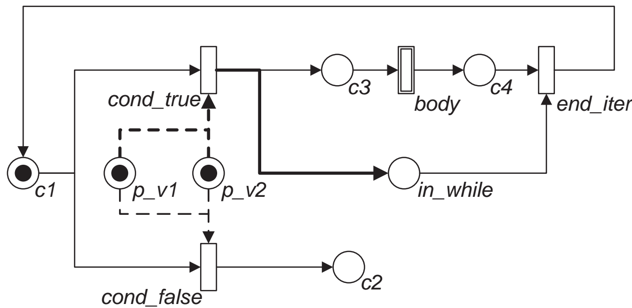



Figure 3.9: The Petri net model for *while* activity

```
<while>
  <condition>boolean_expression($v1, $v2)</condition>
  <!-- while_body -->
</while>
```

Figure 3.10: The WS-BPEL code of a *while* activity

ing that *v3* depends on *v1* and *v2* can be generated.

The Petri net model of a *while* activity is shown in Figure 3.9. The corresponding WS-BPEL code is shown in Figure 3.10. Places $c1$ to $c4$ model the control flow. The variables $v1$ and $v2$ (and could be more), which appear in the *while* conditional expression, are modelled as places $p_v1$ and $p_v2$. At runtime, the evaluation result of the conditional expression determines whether the $body$ of the $while$ iteration is executed or not. This is modeled as the transitions $cond_true$ and $cond_false$. These transitions "read" the places $p_v1$ and $p_v2$. The "read" behavior of process variables is modeled, but we do not model the actual evaluation of condition expression.

The process variables could be changed inside the $while$ iteration, so a data dependency should be generated to indicate that these variables depend on the variables that occur in the $while$ condition ($v1$ and $v2$). This mechanism is similar with the $if$ model, but in this case we use the place $in_while$ to indicate that the WS-BPEL execution is inside the while iteration. This place works together with the Petri net model of the $assignment$ activity to generate this dependency.

The Petri net model for a $pick$ activity with two alternatives is shown in

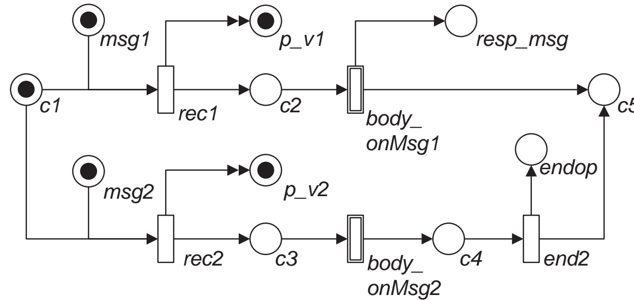



Figure 3.11: The Petri net model for *pick* activity

```
<pick>
  <onMessage variable="v1">
    <!-- body_onMsg1 -->
  </onMessage>
  <onMessage variable="v2">
    <!-- body_onMsg2 -->
  </onMessage>
</pick>
```

Figure 3.12: The WS-BPEL code of a *pick* activity

Figure 3.11, where the corresponding WS-BPEL code is shown in Figure 3.12. Places $c1$ to $c5$ model the control flow of the *pick* activity. Transitions $rec1$ and $rec2$ model the receiving behavior of each <*onMessage*> branch although more receives are possible. Hierarchical transitions $body_onMsg1$ and $body_onMsg2$ model each of the $onMessage$ branches of the pick activity. If there is a $reply$ activity corresponding to the $onMessage$ branch, the output message corresponds to the $resp_msg$. If there is no $reply$ for the $onMessage$ (the message $msg2$ is a one-way message without corresponding reply), we use the transition $end2$ to model the end of this branch to facilitate simulation. The timer-based event is not supported in our current version of the $pick$ model.

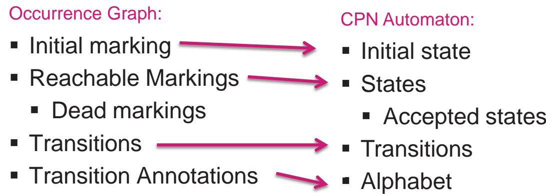



Figure 3.13: A mapping from occurrence graph to automata

### 3.5.3 Occurrence graphs

An occurrence graph can be used to represent all possible states and transitions of a Petri net model, thus to represent all possible control flows of a process. In a Petri net model, the state of a process is modeled as the distribution of tokens over the places. A distribution of tokens over the places is called a marking. The occurrence graph of a Petri net model is a directed graph, where the set of nodes corresponds to the set of all reachable markings such that from the initial marking. The set of arcs corresponds to the firing of Petri net transition. There is a arc from $M$ to $M'$ if and only if there is a transition that the fire of transition re-distribute the marking from $M$ to $M'$.

We can simulate the Petri net model to get an occurrence graph. By our design of bounded Petri net model, the occurrence graph has only finite markings. We represent the occurrence graph model with automata. Figure 3.13 shows how Petri net concepts are mapped to automaton concepts. The Petri net transitions are annotated with the names of the business activities, so the Petri net transition set is represented the automaton alphabet, which contains the names of the activities.

The automata model is used for correctness validation where the detail of the correctness validation based on these automata model is presented in section 8.

## 3.6 Nested word automata model of WS-BPEL

We model WS-BPEL processes using NWA in this section. We use NWA to describe the underlying semantics of WS-BPEL and use them as a basis for our formal solution of pending request failure. An NWA model is an extended automata that can be used to describe the process control flow and the messages

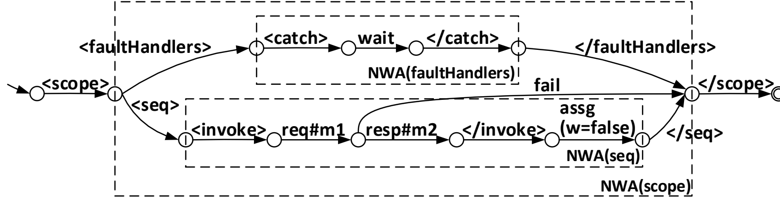



Figure 3.14: An example of NWA graphical representation

sending and receiving status. We choose NWA because we need to model the nested structure of WS-BPEL syntax. While traditional finite state automata can be used for describing all possible states of messages, and their sending and receiving sequences, they lack the capability of describing nested structures of activities. An NWA keeps the nested structure of a WS-BPEL process so that the NWA model can be transformed back to a robust WS-BPEL process.

### 3.6.1 NWA (nested word automata)

Informally, a nested word automaton is an automaton that has hierarchical nesting structures. For instance, the NWA in Figure 3.14 has an nested NWA (scope), which has a nested NWA (seq). The transition pointing to the initial state of the nested automaton is called a *call transition*. As an example, the transitions *<scope>* and *<seq>* in Figure 3.14 are call transitions. The transition "leaving" the nested automaton is called a *return transition*. For example, the transitions *</scope>* and *</seq>* are return transitions. The other transitions, which are similar to transitions of classical automata [87], are named *internal transition*. Formally, a NWA (*nested word automata*) $A$ over an alphabet $\Sigma$ is a structure
$(Q, q_0, Q_f, P, p_0, P_f, \delta_c, \delta_i, \delta_r)$ consisting of

- a finite set of (linear) states $Q$,
- an initial (linear) state $q_0 \in Q$,
- a set of (linear) final states $Q_f \subseteq Q$,
- a finite set of hierarchical states $P$,
- an initial hierarchical state $p_0 \in P$,

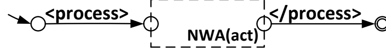



Figure 3.15: NWA model of a process

- a set of hierarchical final states $P_f \subseteq P$,
- a call-transition function $\delta_c : Q \times \Sigma \mapsto Q \times P$,
- an internal-transition function $\delta_i : Q \times \Sigma \mapsto Q$, and
- a return-transition function $\delta_r$: $Q \times P \times \Sigma \mapsto Q$.

The hierarchical states $P, p_0, P_f$ are used to describe the nesting structure of a NWA. The detailed presentation can be found in [88].

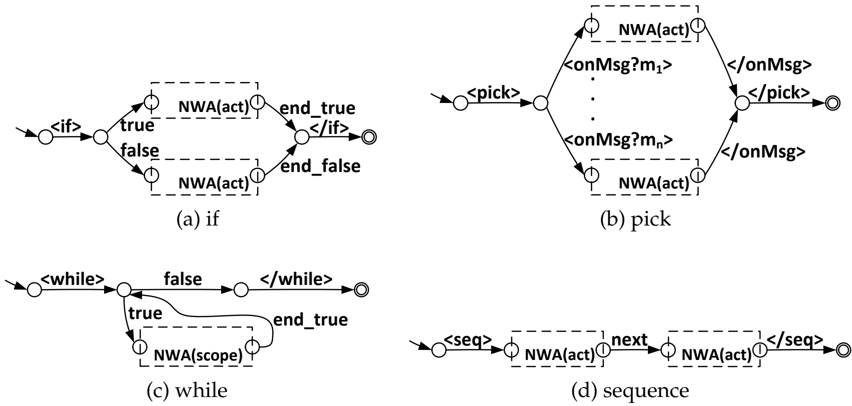
In order to improve readability, we use some graphical conventions presenting NWA model of WS-BPEL. As shown in 4.15b, the nested structures are graphically presented as dashed boxes. A call-transition is named in our paper as <...> (in angle brackets), for example, *<invoke>*, *<while>*. A return-transition is named in as </...> (in angle brackets with slash), for example, *</invoke>*, *</while>*. WS-BPEL activities are divided into two categories, basic and structured activities. The NWA model of a WS-BPEL process is shown as Figure 3.15. A call transition *<process>* starts from the initial state and a return transition *</process>* leads to the accepted state, with the NWA model of an activity, NWA(act), "nested" within it.

### 3.6.2 NWA model of WS-BPEL structured activities

The currently supported activities are *if*, *pick*, *while* and *sequence*, as are shown in Figure 3.16. By elaborate design of NWA model of WS-BPEL activities, from a initial state, there is only a call transition out and there is only a return transition that leads to the accepted state. The NWA model of other activities are "nested" within it. The conditional branch (*if*), repeat (*while*), and sequential (*sequence*) activities as modeled as Figure 3.16a, 3.16c, 3.16d. *pick* (Figure 3.16b) is another case of conditional control flow, which depends on the type of incoming message.

### 3.6.3 NWA model of WS-BPEL basic activities

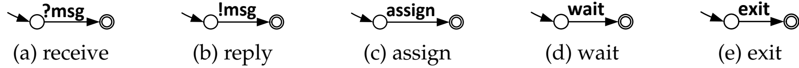
The currently supported basic activities are modeled as Figure 3.17 and 4.15a. The models are pretty straightforward, thus we omit the description. However,




(a) if (b) pick

(c) while (d) sequence

Figure 3.16: NWA model of WS-BPEL structured activities




(a) receive (b) reply (c) assign (d) wait (e) exit

Figure 3.17: NWA model of WS-BPEL basic Activities

the model of *exit* is different. Once the transition *exit* fires, the NWA goes to a terminated state. From this state, the NWA is *dead*, which breaks the well nested structure.

### 3.6.4 Flattened automata model of WS-BPEL process

We use boolean operations of automata, e.g., intersection, difference, to test some control flow properties of a WS-BPEL process, e.g., to determine further interactions of at a specific state. These operations are based on traditional automata, thus, we maintain a flattened automaton by ignoring hierarchical information. Given an NWA $(Q, q_0, Q_f, P, p_0, P_f, \delta_c, \delta_i, \delta_r)$ over the alphabet $\Sigma$, the

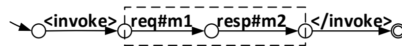



Figure 3.18: NWA model of *invoke* activity

corresponding flattened automaton is $A(Q, q_0, Q_f, \Sigma, \delta)$, where $Q, q_0, Q_f$ and $\Sigma$ are the same as in the NWA, and the transition function $\delta$ is defined as:

1. $\delta(q_{i1}, a) = q_{i2}$, if the NWA has an internal transition $\delta_i(q_{i1}, a) = q_{i2}$.
2. $\delta(q_{c1}, a) = q_{c2}$, if the NWA has a call transition $\delta_c(q_{c1}, a) = (p, q_{c2})$.
3. $\delta(q_{r1}, a) = q_{r2}$, if the NWA has a return transition $\delta_r(q_{r1}, p, a) = q_{r2}$.

Both call and return transitions are treated as flat transitions so that the hierarchical relationship between state is not considered in the flattened model.

## 3.7 Conclusions

This chapter presents the general concepts and models which are used throughout this thesis. Our process transformation based solution is specific to the state types which is defined at the beginning of this chapter. The transformation method works at model level. We have proposed a Petri net model and an NWA model to provide a formal basis upon which the transformation is based.

CHAPTER 4

# Recovery of *pending request failure*

This chapter presents our solution for the *pending request failure*. As introduced in Chapter 1, a pending request failure is caused by the initiator, whose system crashes after sending a synchronous request message and before receipt of the corresponding response message. For example, if a user submits an order for a flight reservation, the browser may crash without receiving any result. The solutions for this failure depend on the shared state types, i.e., how state is shared between business processes and their runtime instances, as described in chapter 3.

In this section, several languages/notations are used for different purposes. The UML sequence diagram is used to illustrate the interaction failures and our high level idea of recovery solution. The graphical notation of business process is used to present our recovery solution. The Web Services Business Process Execution Language (WS-BPEL) code is used to illustrate some implementation details. The Petri net models form a basis for correctness validation. The purposes of these languages are shown in Figure 4.1.

For each shared state type, we use one section to present our solution. This chapter is structured as follows. The pending request failure is introduced in Section 4.1. Our solution for state type *1 : 1* and *n : 1* are presented in Sections 4.2 and Section 4.3 respectively. The solution for the other two state types is discussed in Section 4.4 and 4.5. This chapter is concluded in Section 4.6.

## 4.1 Pending request failure

The pending request failure is depicted in Figure 4.2 after sending a synchronous request message. Due to an initiator system crash, the network connection cannot deliver the response message to the initiator, which leaves the request message ($reqMsg$) pending.

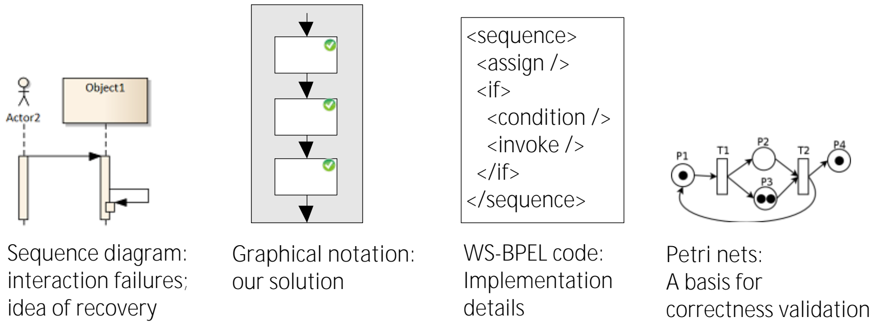



Figure 4.1: Purposes of languages/notations

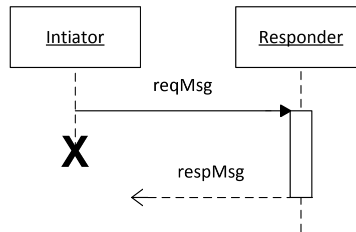



Figure 4.2: Pending request failure

According to our experimental experience, the responder ignores this failure in the default mode of the well-known process engines such as Apache ODE, and Oracle SOA Suite. The responder sends the response message and continues execution. However, after an initiator system restart, the initiator waits for a response message until a timeout. At process language level, a catchable exception will be thrown by the process engine (a software that executes business process). The pending request failure can be produced as follows.

1. Initiator sends a synchronous request message.
2. Responder receives the request message, starts processing while initiator waits for the response.
3. Initiator system crashes.
4. Responder finished processing, sends the response message and contin-

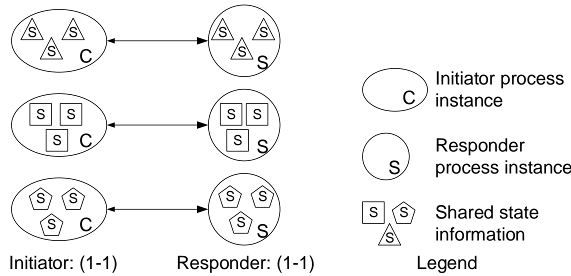



Figure 4.3: *1 : 1* state type

ues execution.

5. The network cannot deliver the response message to initiator and the response message is lost.

6. Initiator system restarts, it continues to wait for the response message, until timeout, then an exception is thrown, which is catchable at process language level.

## 4.2 Pending request failure recovery for shared state type *1 : 1*

The concept of the *1 : 1* state type is shown as Figure 4.3. At runtime, the initiator process may have three running instances, represented as ellipses. The responder process may have three running instances, represented as circles. Each initiator process instance synchronizes its state with a single responder instance (1-1) and vice versa. The state information is private to each initiator-responder instance pair.

If a pending request failure happens, the responder can continue execution until the point of waiting for the next incoming message (if any), which is from the crashed initiator. Due to the *1 : 1* shared state type, no message from other initiator instances will be sent to this responder instance.

Our idea of recovery is that the initiator resends the request after a system restarts, and that the responder, after receiving the resent request message, uses a cached response message as a reply [89, 90]. The transformation of the responder is based on the *further interaction*, to make the further interaction

(a) original processes, no failure

(b) original process, pending request failure

Figure 4.4: An overview of our recovery method, original processes

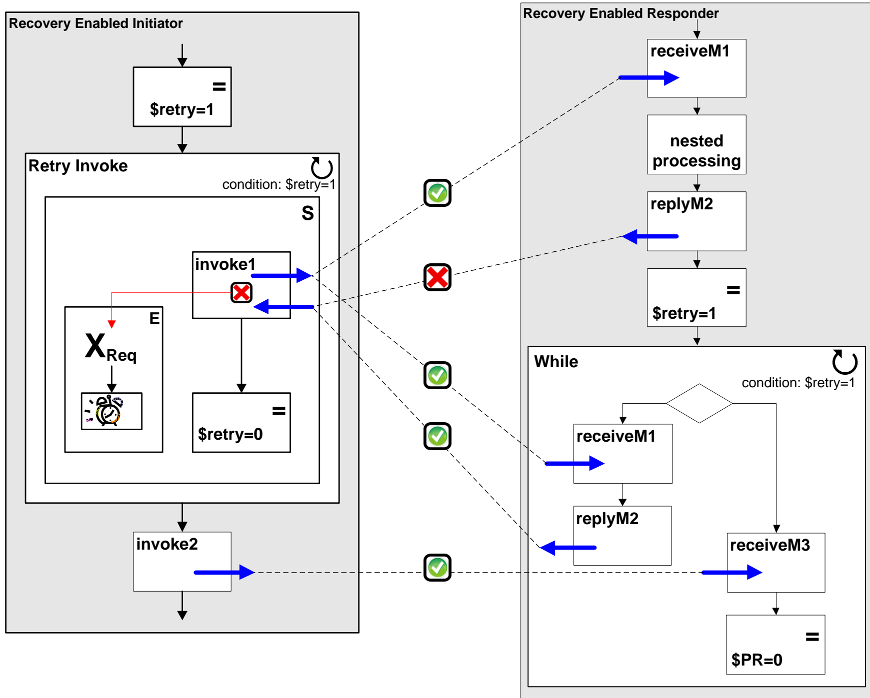
able to accept the failure resent message and to use the cached response in the reply.

Generally speaking, for a business process interaction where failure may happen, there are three kinds of possible *further interaction*. The first kind is a determinate further interaction. The solution for this case is presented in subsection 4.2.1. The second kind is an indeterminate further interaction. In this case the process control flow is complex and has multiple possible next incoming messages, possibly due to the arbitrary combination of the conditional control flow and *while* iteration. This will be further explored in subsections 4.2.2-4.2.4. Finally, the failed interaction could be the final interaction. In this case, there is no further interaction. The solution of this case is discussed in subsection 4.2.5.

### 4.2.1 Recovery on determinate further interaction

In this subsection we assume that a determinate further interaction is the next interaction in a sequence control flow. We present the original behavior when a pending request failure occurs and the corresponding robust business process. Figure 4.4a shows the process behavior of a synchronous interaction followed by an arbitrary further interaction, represented by a one way message. On the initiator side, a synchonous interaction is denoted by the *invoke* activity "invoke1", while on the responder side, the activities "receiveM1" and "replyM2" are used to accept the request message and send the response message. A pending request failure is marked as $X_{Req}$ on initiator side (see Figure 4.4b), in the middle of the *invoke* activity "invoke1". However, the responder is ignorant of this failure and continues execution, until blocked waiting for a further incoming message.

The transformed processes are shown as Figure 4.5. On the initiator side, inside a *while* iteration called "Retry Invoke", we put a scope activity with a fault handler. When the pending request failure happens, the fault handler will be triggered, where an *wait* activity will delay the execution of the process. The outside *while* iteration will make the request sent again. If no failure happens, the variable *retry* will be set to 0 to end the while iteration. On the responder side, the *receive* activity "receiveM2" is replaced by a *while* iteration with a *pick* branch in it. The *while* iteration is used to process the possible request resent by the initiator due to failure. On the left hand side of the *pick* branch, the activities "receiveM1" and "replyM2" are used to respond to the request resent from the initiator. Note that the process will not process the resent request message (as this has already been done), but only replies with the previous response message. On the right hand side of the *pick* branch, the activity "receiveM3" represents possible processes further interaction. In case that the message for a



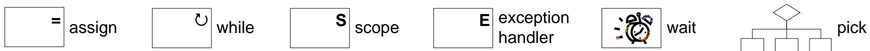




Figure 4.5: An overview of our recovery method, robust processes

further interaction is sent and the *receive* activity "receiveM3" is executed, this implies that the initiator has received the previous response. We then assign the process variable $\$PR = 0$ to end the *while* iteration.

The following Petri nets models of interactions and our corresponding failures are used for correctness validation. How these formal models are used is presented in section 8.

Figure 4.6: Petri net model of the original behavior with a pending request failure

### Petri net representation of the failure model

The Petri net model of the original behavior with occurrence of a pending request failure (see Figure 4.4b) is shown in Figure 4.6. The transitions $t_ch1$ and $t_ch2$ are the model of the communication channel between initiator and responder. The left hand side is the model of *invoke* activity of an initiator. The places $p_v1, \ldots, p_v4$ represent the four process variables, where the process variables represented by places $p_v1$ and $p_v2$ are read and written by the *invoke* activity of the initiator. The place $p_v3$ represents the process variable written by the *receive* activity of the responder. The place $p_v4$ represents the process variable read by the *reply* activity of the responder. The places $p_m1, \ldots, p_m4$ are the model of the messages and the places $p_c1, \ldots, p_c7$ represent the process control flow. The transitions $t_inv1$ and $t_inv2$ model that the initiator sends and receives the messages. The transitions $t_rec$ and $t_rep$ is the model that the responder receives and replies messages. The transition $t_any$ can be replaced by any sub Petri net to represent the processing of the request message. The initiator system crash is represented as a transition $t_req$. If this transition fires, a token is taken from the place $p_c2$ to represent that the initiator has deviated from the normal control flow. Another token is taken from $p_m2$ to represent the response message loss. A token is put into the place $p_req$ to indicate that pending request failure happens.

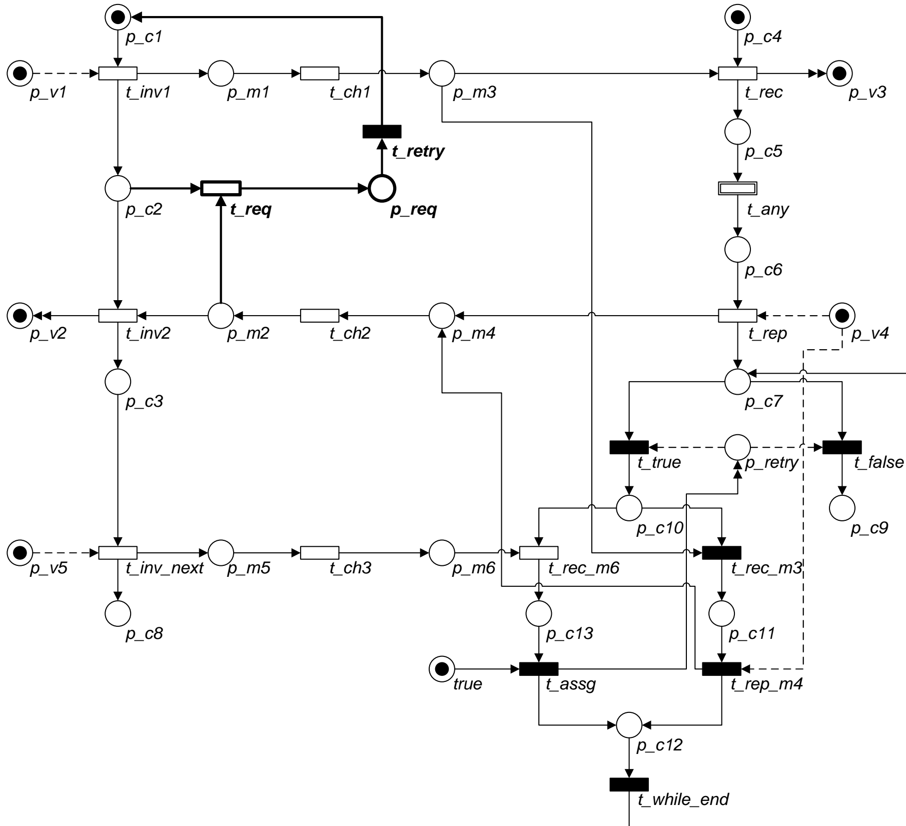



Figure 4.7: Petri net model of pending request failure recovery

## Robust process model

The Petri net model of the transformed process for pending request synchronization failure is presented in Figure 4.7. The pending request synchronization failure happens when the initiator crashes during a synchronous exchange. The basic idea of our synchronization mechanism for pending request failure is based on message resending. The initiator process should resend the request once it recovers. If the responder process receives the same synchronous request message multiple times, it should send the response for each request. One problem behind this idea is that if the initiator process fails after send-

ing the request (transition $t_req$ in Figure 4.6), the responder process may still send the response and continue its execution (transition $t_rep$ occurs in Figure 4.6). If this synchronous exchange is the final interaction between the two processes, the responder process may terminate its instance after the synchronous exchange. Therefore, our synchronization mechanism works based on the assumption that there will be an additional interaction between the initiator and the responder, represented as $t_inv_next$ and $t_rec_m3$ in Figure 4.7.

In order to make recovery possible, the transitions added to the original behavior are represented as black boxes. Divided by the channel transitions $t_ch1$, $t_ch2$ and $t_ch3$, the left side corresponds to the Petri net model of the initiator process and the right side corresponds to the responder process. Places $p_c1$, $p_c2$, $p_c3$ and $p_c8$ represent the control dependencies of the initiator process. Process variables are represented by places $p_v1$, $p_v2$ and $p_v5$. Sent and received messages are represented by places $p_m1$, $p_m2$ and $p_m5$. After the initiator process has sent the request message (transition $t_inv1$ fires) but has not received yet the response message (a token in place $p_c2$), firing the transition $t_req$ introduces a token in place $p_req$, representing a pending request synchronization failure observable by the initiator process. The control dependency of the responder process is represented by the places $p_c4, \ldots, p_c7, p_c9, \ldots, p_c12$. Places $p_v3$, $p_v4$ and $p_retry$ are process variables. The places $p_m3$, $p_m4$ and $p_m6$ represent the messages. The transition $t_rec$, $t_any$ and $t_rep$ represent the receiving, processing and replying of the synchronous interaction.

The transformed responder process model is specified as follows. The process variable $p_retry$ is initialized to $true$ by default. The conditions of the following while iteration are represented as $t_true$ and $t_false$. The pick branches end with a token put into place $p_c12$. If the synchronous request message is sent by the initiator multiple times (transition $t_rec_m3$ fires), the responder process sends a reply message without processing the message (firing of transition $t_rep_m4$). If the responder process receives a message for further interaction (transition $t_rec_m6$ fires), the assignment activity modeled by transition $t_assg$ changes condition variable $p_retry$ in order to end the *while* iteration.

### 4.2.2 Recovery on indeterminate further interaction

The solution of recovery from a pending request failure presented above has an implicit assumption that the type of the next incoming message is determinate, i.e., there is exactly one possible further interaction in the control flow. However, in some processes this is not the case. Let's consider an example of orig-

```
<while>
  <condition>…</condition>
  <receive variable="m1" />
  <reply variable="m2" />
  <receive variable="m3" />
  <reply variable="m4" />
</while>
<receive variable="m5" />
```

Figure 4.8: Indeterminate further interaction in the original responder behavior, an illustration

inal responder behavior [91], illustrated in Figure 4.8: if the process receives message $m3$ and then replies with message $m4$, the possible next incoming messages are:

1. The message $m1$, because of another iteration of the while loop;
2. The message $m5$, because the condition variable does not allow another iteration of the while loop.
3. The message $m3$, because of a resent request message due to a pending request failure on the client side;

For this process, a robust process acquired based on Figure 4.5 does not apply because of the indeterminate further interaction, which could be incoming message $m1$, $m3$ or $m5$. However, for this simple process, we can easily infer all the possible process interactions, and then acquire robust process, as shown in Figure 4.9. One possible further interaction, the receipt of message $m1$, is replaced by a *pick* activity nested in a *while* iteration to process the possible resent message $m3$ as well as the message $m1$. In the *pick* activity, we add one *onMessage* branch to accept the resent message $m3$ and use the message $m4$ as the response. The variable $\$succ$ is used as a *while* condition flag, which is assigned to $1$ only when the initiator sends the request for the next interaction (the second *onMessage* branch that receives message $m1$). However, a resent message could be sent multiple times before the response is ultimately received. We nest the *pick* activity in a *while* to cope with the duplicate resent message. Similarly, another possible further interaction, the *receive* of message $m5$, is replaced by another *while* iteration. However, a general mechanism is required that whenever a message is replied ($m4$ in the above case), all possible further interactions can be identified ($m1$ and $m5$ in the above case) and

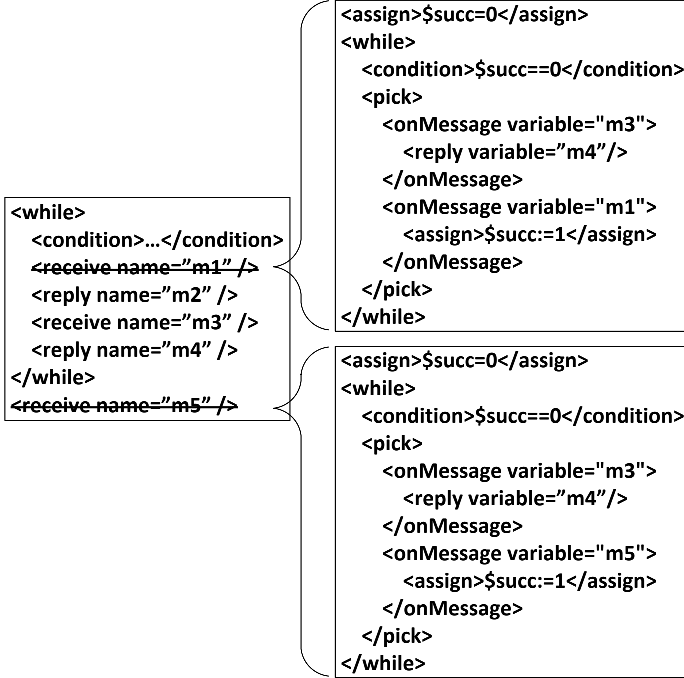



Figure 4.9: Indeterminate further interaction, an illustration

equipped with the capability of accepting the resent message ($m3$) and reply with the previous cached response ($m4$ again).

### 4.2.3 The robust responder process

For a WS-BPEL process, given its *NWA* model $(Q, q_0, Q_f, P, p_0.P_f, \delta_c, \delta_i, \delta_r)$ over the alphabet $\Sigma$, we assume that the alphabet that represents the response messages is $\Sigma_{resp}$ and the alphabet that represents the request messages is $\Sigma_{req}$, thus $\Sigma_{req} \subseteq \Sigma$ and $\Sigma_{resp} \subseteq \Sigma$. The transformation algorithm is as Figure 4.10.

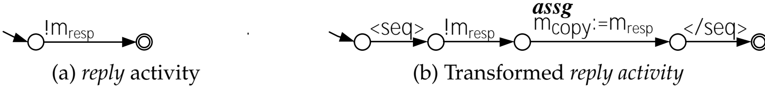
The algorithm iterates through all combinations of a state $q$, a request message $?m_{req}$ and a response message $!m_{resp}$. In line 2, we check if the message pair $(?m_{req}, !m_{resp})$ corresponds to the request and response for a synchronous operation and at state $q$, the response message $!m_{resp}$ is sent, represented by a transition $\delta_i(q, !m_{resp})$. This is the failure point that the response message may

1: **for all** $q \in Q$, $?m_{req} \in \Sigma_{req}$ **and** $!m_{resp} \in \Sigma_{resp}$ **do**
2: **if** $(m_{req}, m_{resp})$ is a synchronous message pair **and** $\delta_i(q, !m_{resp})$ is defined in *NWA* **then**
3: $save_reply(q, !m_{resp})$
4: N $\leftarrow next_receive(q, !m_{resp})$
5: **for all** $?m_{next} \in N$ **and** $q_{next} \in Q$ **do**
6: **if** $\delta_i(q_{next}, ?m_{next})$ is defined in *NWA* **then**
7: $transform_receive(q_{next}, ?m_{next})$
8: **else if** $\delta_c(q_{next}, ?m_{next})$ is defined in *NWA* **then**
9: $transform_pick(q_{next}, ?m_{next})$
10: **end if**
11: **end for**
12: **end if**
13: **end for**

Figure 4.10: Responder process transformation algorithm

be lost due to interaction failures and where our transformation method applies.

As defined in line 3, we first make a copy of the response message, as shown in Figure 4.11. The NWA model of *reply* activity in Figure 4.11a is replaced by an NWA model of a *sequence* activity in Figure 4.11b, in which a *reply* activity model and an *assign* activity model are nested. The *assign* activity model represents the copy of the reply message into the variable $m_{copy}$. In order to process the possible resent request message $?m_{req}$ due to the lost of the message $!m_{resp}$ sent at state $q$, we calculate the set of all possible next incoming messages, which is defined as $next_receive(q, !m_{resp})$ in line 4. We construct an automaton $A(!m_{resp}, ?m_{next})$ as in Figure 4.12 to describe that a process replies with a message $!m_{resp}$ and waits for some possible next incoming message $?m_{next}$. $\delta(q_0, !m_{resp}) = q_1$ models the reply of the response message $!m_{resp}$. $\delta(q_1, \Sigma/\Sigma_{req}) = q_1$ represents some process execution in which no messages are received. $\delta(q_1, ?m_{next}) = q_2$ represents that the process receives an incoming message $?m_{next}$. $\delta(q_2, \Sigma) = q_2$ models any process execution. For the process *NWA* model, at some state $q$, a reply of a message $!m_{resp}$ is represented by an internal transition $\delta_i(q, m_{resp})$. We change the initial state of the process *NWA* model to from $q_0$ to $q$, and call this automaton *NWA*$(q)$. Starting at $q$, after replying the message $!m_{resp}$, if one possible next incoming message is $?m_{next}$, then *NWA*$(q) \cap A(!m_{resp}, ?m_{next}) \neq \emptyset$, i.e., the process modeled by *NWA* has




(a) *reply* activity (b) Transformed *reply activity*

Figure 4.11: Responder process transformation, *reply* activity

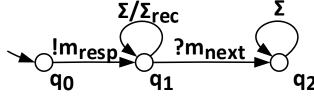



Figure 4.12: The automaton $A(!m_i, ?m_{next})$

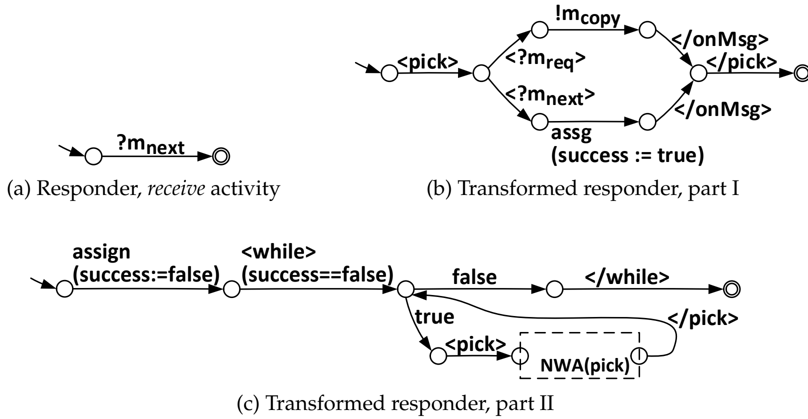
the behavior described by $A(!m_{resp}, ?m_{next})$.

The intersection operation $\cap$ between an *NWA* and an finite state automaton is defined to check whether the business process modeled by the *NWA* has the message sending and receiving behavior modeled by the automaton. The intersection operation is based on finite state automata. We "flatten" an *NWA* to a finite state automaton by skipping hierarchical information, described as follows. Given a NWA $(Q, q_0, Q_f, P, p_0, P_f, \delta_c, \delta_i, \delta_r)$ over the alphabet $\Sigma$, the "flattened" automaton is $A(Q, q_0, Q_f, \Sigma, \delta)$, where $Q, q_0, Q_f$ and $\Sigma$ are the same as the NWA, the transition function $\delta$ is defined as

1. $\delta(q_{i1}, a) = q_{i2}$, if the NWA has an internal transition $\delta_i(q_{i1}, a) = q_{i2}$.
2. $\delta(q_{c1}, a) = q_{c2}$, if the NWA has a call transition $\delta_c(q_{c1}, a) = (p, q_{c2})$.
3. $\delta(q_{r1}, a) = q_{r2}$, if the NWA has a return transition $\delta_r(q_{r1}, p, a) = q_{r2}$.

Both call transitions and return transition are treated as flat transitions that the hierarchical state p is not considered. The intersection operation can be done between two finite state automata, as defined in [87]. We define the set of all possible next incoming messages as
$next_receive(q, !m_{resp})$
$= \{?m_{next} | ?m_{next} \in \Sigma_{req} \wedge NWA(q) \cap A(!m_{resp}, ?m_{next}) \neq \emptyset\}$.

For all $?m_{next} \in next_receive(q, !m_{resp})$ and $q_{next} \in Q$, if at the state $q_{next}$ the next incoming message $?m_{next}$ is received, two cases of transition may be defined in *NWA*: in a model of a *receive* activity as an internal transition $\delta_i(q_{next}, ?m_{next})$ or in the model of a *pick* activity as a call transition $\delta_c(q_{next}, ?m_{next})$. For the first case (line 6), as shown in Figure 4.13a, the procedure $transform_receive(q_{next}, ?m_{next})$ is introduced as follows. We replace




(a) Responder, *receive* activity

(b) Transformed responder, part I

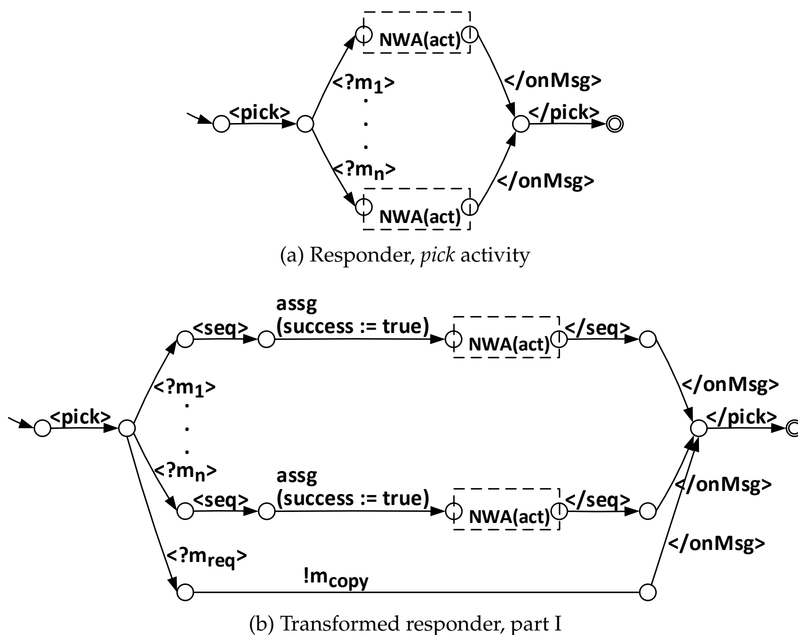
(c) Transformed responder, part II

Figure 4.13: Responder process transformation, *receive* activity

the transition with a *pick* activity with two branches, as shown in Figure 4.13b. One *onMessage* branch models the receive of the resent message $?m_{req}$ and the reply of the result message $m_{copy}$. The other *onMessage* branch models the receive of the message $?m_{next}$, and after that we set the flag $success$ to $true$ to indicate that the previous interaction is finished successfully. However, a possible loss of the response message $m_{copy}$ triggers multiple resending of the request $m_{req}$. Therefore, the *pick* activity is defined in a *while* iteration so that multiple requests $?m_{req}$ can be accepted. Figure 4.13c shows that the *while* iteration ends when the flag $success$ is set to *true*.

For the second case (line 8), as shown in Figure 4.14a, the message $?m_{next}$ is one of the messages in $m_1, ..., m_n$. Figure 4.14b shows that we then add a call transition $<?m_{req}>$ to model that the process accepts the resent message, and an internal transition $!m_{copy}$ to represent the reply using a copy of the previously cached result $m_{copy}$. In the other branches, the nested NWA(act) is replaced by the model of a *sequence* activity, in which we model the assignment of the flag variable *success* to *true*, followed by the original NWA(act). Similarly, in order to cope with a possible loss of the response message $m_{copy}$, the *pick* activity model is nested in a *while* iteration to handle multiple resent messages, as shown in Figure 4.13c.

After the transformation, at some states the responder can receive more messages than the original process, because the resent message can be accepted




(a) Responder, *pick* activity



(b) Transformed responder, part I

Figure 4.14: Responder process transformation, *pick* activity

and be replied. However, the request is not processed again. In this sense, we do not give malicious initiators any chance of jeopardizing the process by changing the sequence of requests or sending the same request multiple times.

**Recoverable assumption**

Assume that $(?m_{req}, !m_{resp})$ is a pair of synchronous request and response messages, the process receives request message $?m_{req}$, then at state $q$, the process sends the response message $!m_{resp}$. However, if $?m_{req} \in next_receive(q, m_{resp})$, then one of the next possible messages is still $?m_{req}$, in this case, the responder cannot distinguish a resent message due to a failure from a normal request message. Thus we have to require that in the process design the condition $?m_{req} \notin next_receive(q, !m_{resp})$ can be met. However, by following a few process design principles during the design of the original process, this condition can be met. An example is, a split of message $?m_{req}$ into two different messages, $?m_{req1}$ and $?m_{req2}$ (for example, one message is used to send request,

the other asks for results). The initiator sends the two messages back to back. If a responder receives $?m_{req1}$, then it waits for $?m_{req2}$, rather than waiting for $?m_{req1}$ again.

### 4.2.4 The robust initiator process

The initiator starts the interaction by executing the *invoke* activity. An *invoke* activity, which is shown as Figure 4.15a, is replaced by the model of a scope activity, which consists of an NWA of a fault handler and a *sequence* activity model. Nested in the *sequence* activity model there is the model of the original *invoke* activity, followed by an assignment of the $false$ value to a process variable $w$. The NWA of the *scope* activity is nested in a *while* activity model.

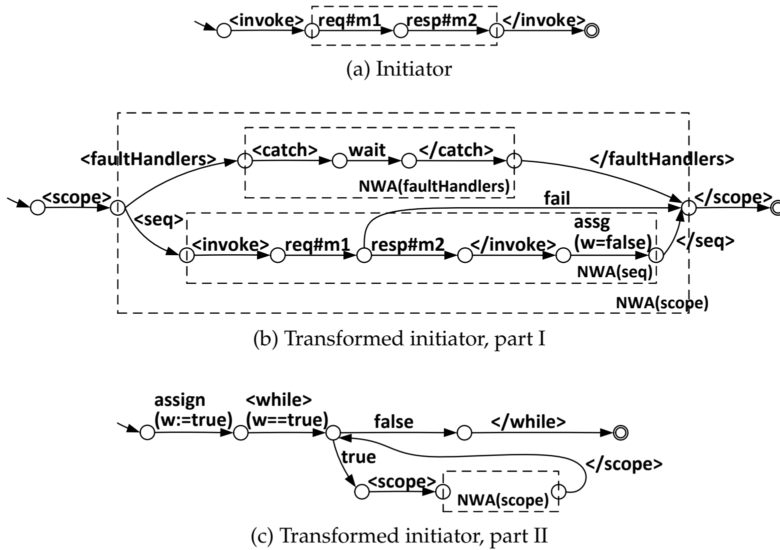
The whole model represents the process behavior of invocation. If failure happens and gets caught by the fault handler, the process waits for a specific time period and finishes the *scope* activity, then the outside *while* makes the *invoke* activity be executed again, until it finishes successfully and the variable $w$ is assigned to $false$. The possible interaction failure is modeled as the transition *fail*. This is a reasonable failure model since that if a failure happens the control flow is deviated from the normal flow to the end of the scope to which a fault handler is attached, rather than leading the process to an exceptional end.

### 4.2.5 Recovery on no further interaction

The solution of recovering from a pending request failure so far has been based on the occurrence of a further interaction after a request message has been accepted and replied. However, if the reply that belongs to the failed synchronous call is the last interaction of the responder process, then the responder will terminate immediately after it has sent the response message. The initiator and re-sends the request after is has restarted. The request cannot be processed by a terminated responder instance.

Hence, the problem here is how to achieve sending a final response message, using the cached response, for a terminated process instance. Several ideas are discussed below:

1. Achieve caching response at engine level: caching the terminated instance responses by process engine. If the request mapped to a terminated instance, the cache response is sent by engine. However, an engine




(a) Initiator

(b) Transformed initiator, part I

(c) Transformed initiator, part II

Figure 4.15: Initiator process transformation

specific implementation will not migrate between different runtime environments.

2. Achieve caching response at process level: we implement a standalone process used as a caching support process. The process provides services like cache operations. The implementation may involve a backend database to store the cached messages.

Generally speaking, by following a few process design guidelines, when there is no further interaction, the occurrence of the pending request failure can be avoided, i.e., when the final interaction is finished, all process instances are safe to be terminated . Possible design guidelines are discussed as follows:

1. Only one party can make the decision to terminate all partner instances: e.g., only the client can decide when the server instance can be terminated. In this way, if the client crashes, it will recover and re-send the request, and it will allow the server to terminate the instance only after it has received the final response.

2. A time based termination. A time interval can be coordinated, the server

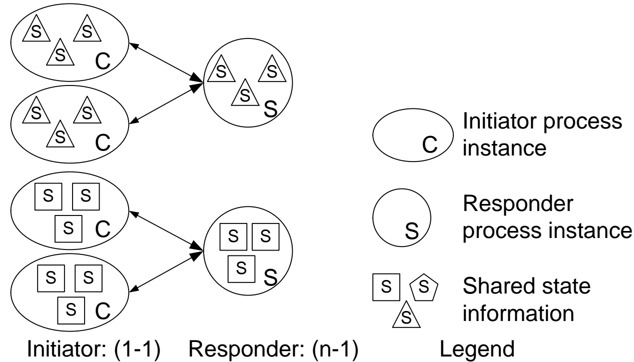



Figure 4.16: *n : 1* state type

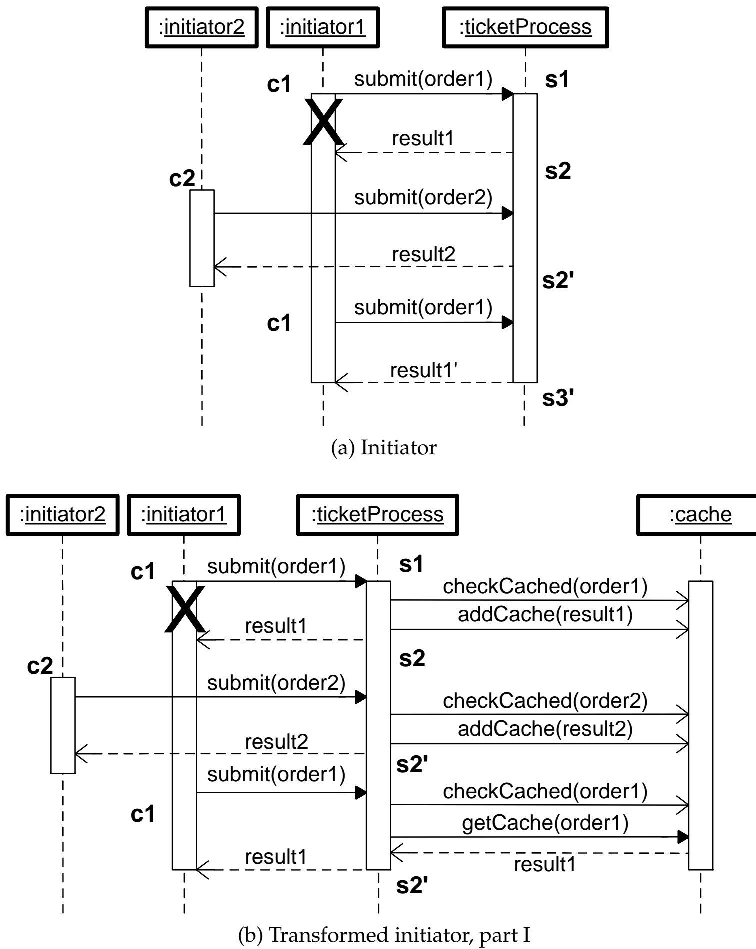
instance should wait until a time out. Then the server can terminate its instance. This is implementable at a technical level, however, this is not preferred because a further coordination is required, making our solution similar to the transactional approach [58, 56, 57].

However, the detailed implementation of these guidelines are out of the scope of this thesis.

## 4.3 Pending request failure recovery for shared state type *n : 1*

In previous section, we have considered the coordination scenarios where the effects of the state changes in one collaboration do not affect other collaborations. In this section, we focus on a responder process instance collaborating with multiple initiator process instances, where one collaboration may affect another collaboration.

The concept of the *n : 1* state type is shown as Figure 4.16. At runtime, the initiator process may have four running instances, represented as ellipses. The responder process may have two running instances, represented as circles. Each initiator process instance synchronizes its state with a single responder instance (1-1), while globally one responder instance interact with multiple initiator instances (n-1). The state information is shared between initiator instances, while the number of the responder instance is *static* (could be one or



(a) Initiator


(b) Transformed initiator, part I

Figure 4.17: Caching response message

more, but it is a fixed number at runtime).

Figure 4.17a illustrates the *pending request failure* with a ticket selling process and multiple initiator processes. At runtime, each initiator process may have multiple instances (*initiator1*, *initiator2*), which submit order messages (*order1*, *order2*). The *initiator1* process may crash after submitting the *order1* message

without receiving the *result1* response message. At a certain state *s1*, the *ticket process* receives the *order1* message, changes its state to s2 and sends the *result1* response. However, the response is not received by the *client1* due to the crash. The *initiator2* process submits message *order2* to the *ticket process* afterwards. The *ticket process* changes its state to *s2′*. Now, the *initiator1* process re-submits the order after recovery. By re-processing the same order at state *s2′*, the *ticket process* will reply with a different *result′*, which may incur state inconsistency. Some operations can be safely repeated. A request that has this property is called "idempotent" [7]. For example, a request asking for weather information can be repeated as many times as possible. However, the ticket subscription operation described above that receives the order submission does not have this property. First, the ticket process state changes to *s2*, but *initiator1* does not change its state accordingly. Second, the *ticket process* further changes its state to *s2′* after interaction with *initiator2*. The solution in the previous section cannot be directly applied because the collaboration between *initiator1* and *ticket process* is affected by the collaboration between *initiator2* and *ticket process*. After the crash of *initiator1*, the state of *ticket process* is further changed. After a restart of *initiator1* and a resend of *order1*, the *ticket process* cannot use a copy of the previous result because it is further changed by the collaboration between *initiator2* and *ticket process*.

Our basic idea to solve the problem is that whenever the *state* of a business process changes, the response message is cached [92, 93]. As shown in Figure 4.17b, after a state change from *s1* to *s2*, the *ticket process* caches *result1*. When *initiator1* re-submits *order1* after recovery, the *ticket process* uses cached *result1* as response to restore state consistency.

The *state* of a business process is described by the values of the process variables. In order to identify process *state* as a subset of the process variables, we model processes using Petri nets [94] to extract the data dependencies. In subsection 4.3.1, we propose state identification criteria and we represent them based on the Petri nets model of the business processes. The original processes can be (automatically) transformed into their synchronization-enabled counterparts via process transformations, which is described in subsection 4.3.2. The transformation is done in such a way that in the resulting processes possible state inconsistencies are recognized and compensated by state-caching, and these processes retry failed interactions based on the contents of the cache.

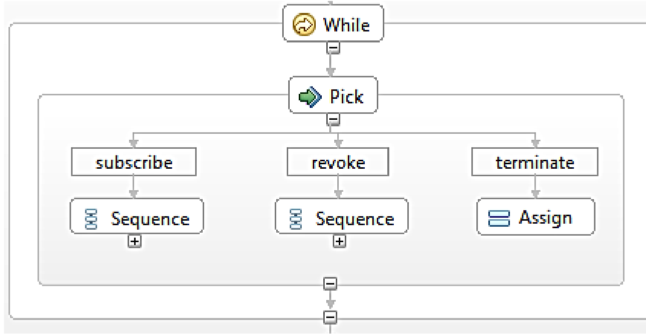



Figure 4.18: Snippet of Ticket Process

## 4.3.1 State determination criteria

### Inbound message activity

In order to identify the synchronous operation boundaries, which is the begin and end of the control flow of a synchronous operation, we introduce the concept of Inbound Message Activity (IMA) in WS-BPEL. IMAs are activities in which messages are received from partners, and consists of the activities *receive* and *pick*. Other types of IMAs, like $eventhandlers$, are out scope of this paper. The control boundary of a synchronous process operation starts with an IMA and ends with a *reply* activity.

Outbound Message Activity (OMA) replies the response message, and consists of the activities *invoke* and *reply*.

IMAs and OMAs correspond to the begin and end of the control boundary of a synchronous process operation, respectively. If a state variable is identified for a synchronous process operation, we cache the response message. We will use a ticket subscribing process to illustrate our criteria to identify process state variables. As shown in Figure 4.18, the core of the process is a *pick* activity. Three *onMessage* handlers are nested inside the *pick* activity for the corresponding message type: “subscribe” for the subscription operation; “revoke” for the ticket revoke operation and “termination” to end the business process. The *pick* activity is nested in a *while* activity, allowing the process operations “subscribe” and “revoke” to be executed multiple times.

Below we discuss the criteria used inside the control boundary of a process operation.

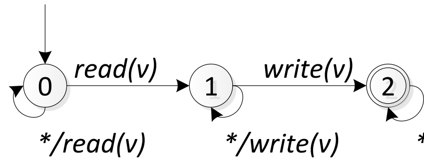



Figure 4.19: Criterion Automaton of Read Before Write

**Inside process operation criterion: read before write**

The process variables that describe the state of a business process should be read first and written afterwards. Formally, in Figure 4.19, this criterion is presented as an automaton with the alphabet $\{read(v), write(v), *\}$, where $read(v)$ and $write(v)$ denote the reading and writing of the process variables $v$, respectively. State $0$ denotes the initial state, State $1$ denotes the state in which the process variable $v$ is read but not being written, and State $2$ is the accepted state, which represents that variable $v$ is read first and written afterwards.

We discuss the use of the criteria automaton to check the Petri net model in subsection 4.3.2.

**Inside process operation criterion: circular dependency**

The data flow denoted by the bold arcs in the Petri Net representation of the places should form a cycle, and the places representing the *state* variables should be included in this cycle. The Petri Net model of the operation "subscribe" of the ticket process is shown in Figure 4.20. The data flow path *true*, *inT*, *assg2*, *sub*, *assg1*, *ticket*, *true* forms a cycle, where two places representing variables can be found: *sub* and *ticket*, which are considered as state variables.

**Cross-process operation criteria**

If a variable $v$ has its value written inside an operation and read outside the operation afterwards, $v$ should considered as a state variable. Without loss of generality, for a specific synchronous process operation, say, the subscribe ticket process operation, we can construct a criteria automaton $\{q_0, Q, F, \sum, \delta\}$, with the alphabet $\sum = \{IMA_subscribe, OMA_subscribe, r_history, w_history\}$ for a process variable $\$history$. $IMA_subscrib$e represents the *receive* activity, while $OMA_subscribe$ represents the *reply* activity. $r_history$ is an assignment activity that reads the value of $\$history$ and $w_history$ is an assignment activity

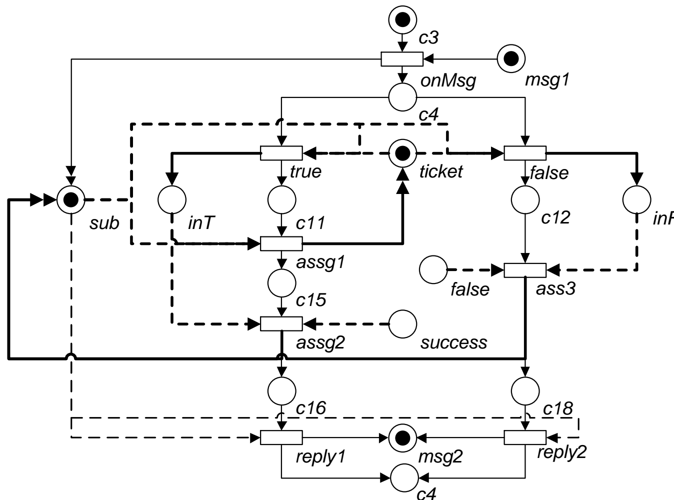



Figure 4.20: Petri Net of Subscribe Operation of the Ticket Process

that writes the value of $\$history$. We define state set $Q$ to contain five states, indexed from $0$ to $4$. The initial state $q_0$ is state $0$. The final state set is $\{4\}$. Figure 4.21 shows the automaton constructured in this way. The transition function $\delta$ is specified as follows:

1. From state $0$: $IMA_subscribe$ leads to state $1$; Stay in state $0$ otherwise.

2. From state $1$: $OMA_subscribe$ leads to state $0$; $w_history$ leads to state $2$; Stay in state $1$ otherwise.

3. From state $2$: $OMA_subscribe$ leads to state $3$; Stay in state $2$ otherwise.

4. From state $3$: $w_history$ leads to state $0$; $r_history$ leads to state $4$. Stay in state $3$ otherwise.

5. From state $4$: Stay in state $4$ for any element of $\sum$.

We discuss the use of the above automaton to check the Petri net model in subsection 4.3.2.

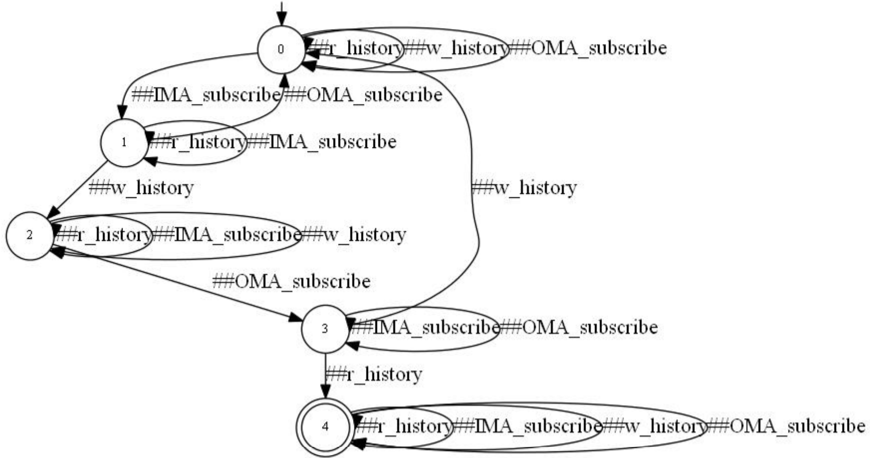



Figure 4.21: Automaton Model of Cross Process Operation Criteria

## 4.3.2 Implementation details

The architecture of our prototype implementation is shown in Figure 4.22. We implemented the state determination criteria proposed in Section 4.3.1 in the State Dependency Analysis module to determine the state information. The result is used to decide whether to trigger the process transformation. The Process Transformation module performs the actual process transformation to cache the response message to achieve robust client/server interaction.

### State Dependency Analysis Module

At the bottom layer is the CPN Simulation Module and the Automaton Class Library of our architecture in Figure 4.22. The CPN Simulation Module generates the Occurrence Graph model from the Petri Net model. Inside this module, the Access/CPN Class Library provides the Petri Net simulation support and the Graph Search Library provides graph representation support. The Occurrence Graph generation algorithm implemented in the State Space Generation Module is presented below.

1 $Init : Queue : \mathbf{Q} \Leftarrow Empty,$
2 $add\ init\ marking\ \mathbf{m0}\ to\ Graph : \mathbf{G}$
3 $Enqueue(\mathbf{Q}, \mathbf{m0})$

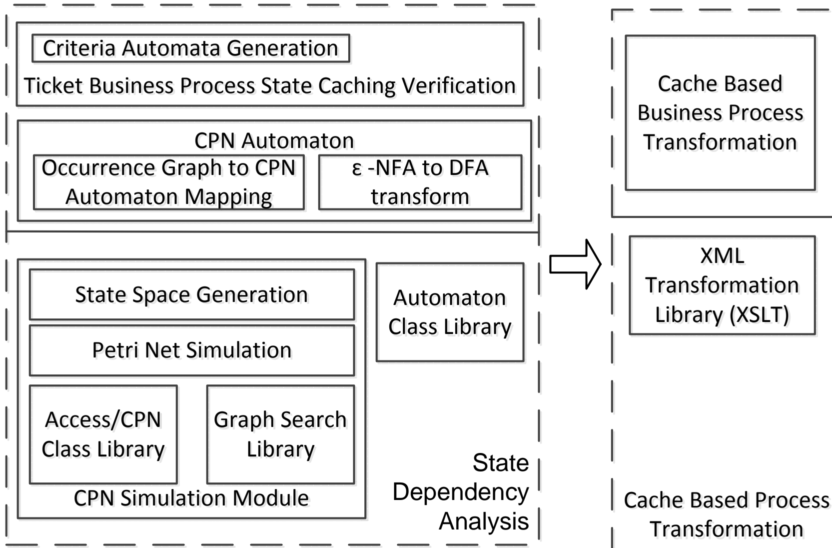



Figure 4.22: Architecture of Prototype Implementation

4 **while**($\mathbf{Q}$ *is not empty*) **do**
5 $marking\ \mathbf{u} \Leftarrow Dequeue(\mathbf{Q})$
6 **for**($each\ \mathbf{v}\ in\ directly\ reachable\ markings$
$from\ \mathbf{u}$) **do**
7 **if**($\mathbf{v}\ is\ not\ in\ \mathbf{G}$) $then$
8 $Enqueue(\mathbf{v}, \mathbf{G})$
9 $add\ \mathbf{v}\ to\ \mathbf{G}$
10 $add < \mathbf{u}, \mathbf{v} >\ to\ \mathbf{G}$

In the middle layer, the occurrence graph is mapped to the automaton. Figure 4.23 shows how Petri Nets concepts are mapped to automaton concepts. The Petri Net transitions are annotated with the names of the business activities, so when the Petri Net transition set is mapped to the automaton alphabet, an additional alphabet $\Sigma'$ is required as input. If the transition name is in $\Sigma'$, the Petri Net transition is mapped to the corresponding automaton transition. If not, the Petri Net transition is mapped to an $\epsilon$ automata transition. We then transform the Non-deterministic Finite Automaton (NFA) containing the $\epsilon$ to a Deterministic Finite Automaton. Finally, we calculate the intersection of the

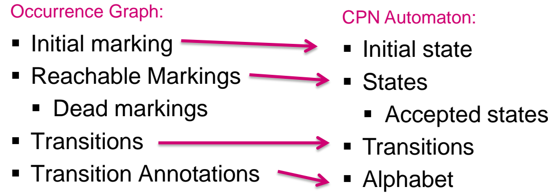



Figure 4.23: A Mapping from Occurrence Graph to Automaton

DFA with the criteria automata in order to determine the necessary state information.

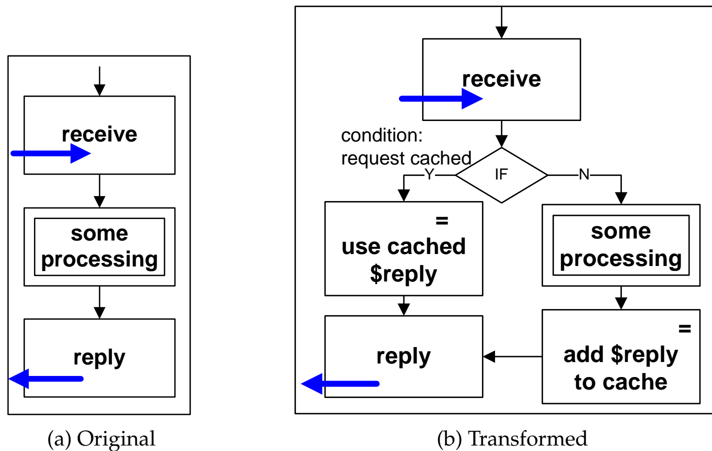
### Process Transformation Module

As shown in Figure 4.24a, a synchronous operation receives a message, performs some processing and then replies. Our transformation replaces the processing and *reply* by an *if* activity, where the condition of the *if* activity checks whether the request message has been cached before. If the message is cached, the process uses the cached response as reply. If the message is not cached, which implies that the message was sent for the first time, the message is processed. The response message is cached and replied.

The data structure of the cache is declared as an array of cached items. Each item is a <request, response> value pair. The cache structure is declared as an XSD definition in WSDL. In the WS-BPEL process, the cache is declared as a variable. Three cache operations are required: 1) Given a request message, check whether the corresponding response message is cached. 2) Given a request, get the corresponding response. 3) Given a value pair of request and corresponding response messages, add it to the cache.

The cache data operation is implemented as a XSLT transformation. An *assign* activity to check whether the request is cached is shown in the following WS-BPEL code:

```
<bpel:assign>
  <bpel:copy>
    <bpel:from>bpel:doXslTransform(
      testCached.xsl,
      $cache, cacheItem,
      $request.payload)
    </bpel:from>
```




(a) Original

(b) Transformed

Figure 4.24: Cache Based Process Transformation Details

```
    <bpel:to
       variable=foundCachedReques />
  </bpel:copy>
</bpel:assign>
```

The *from* part of the assignment activity is the BPEL function doXslTransform() with the request message and $\$cache$ as its parameters. Variable $\$foundCachedReques$ contains the result.

## 4.4 Pending request failure recovery for shared state type *1 : n*

The concept of the *1 : n* state type is shown as Figure 4.25. At runtime, the initiator process may have two running instances, represented as ellipses. The responder process may have four running instances, represented as circles. Each initiator process instance synchronizes its state with multiple responder instances (1-n), while globally one responder instance interact with one initiator instance (1-1). The state information is private to each initiator instance, but shared between multiple server instances.

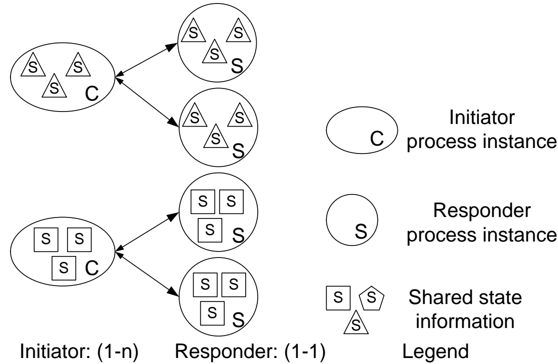



Figure 4.25: *1 : n* state type

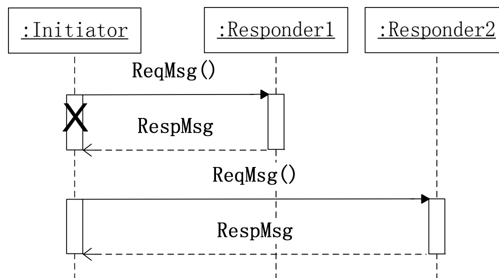



Figure 4.26: Pending request failure of state type *1 : n*

The pending request failure of shared state type *1 : n* is marked as $X$ in Figure 4.26. The solution in section 4.2 can be applied because that after the *initiator* system crash, there is no further interaction between *initiator* and *responder1*, thus after the *initiator* system restart, a request resent from the *initiator* will be replied with the previous result.

## 4.5 Pending request failure recovery for shared state type *m : n*

The concept of the $m : n$ state type is shown as Figure 4.27. At runtime, the initiator process may have two running instances, represented as ellipses. The

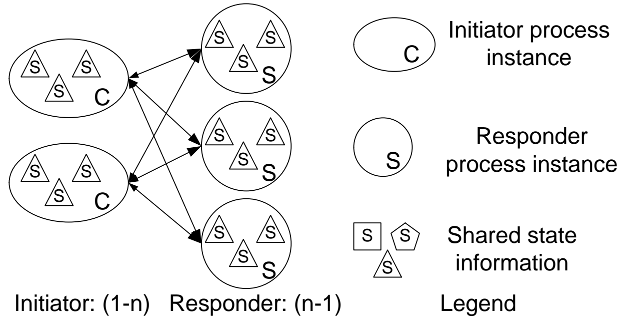



Figure 4.27: $m : n$ state type

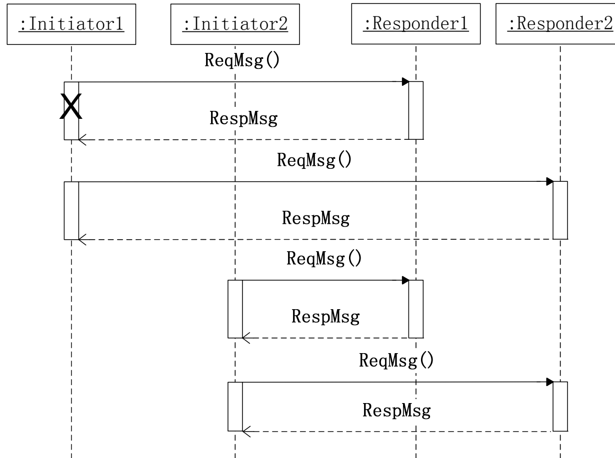



Figure 4.28: Pending request failure of state type $m : n$

responder process may have three running instances, represented as circles. Each initiator process instance synchronizes its state with multiple responder instances (1-n), while each responder instance interact with multiple initiator instance (n-1). The state information is shared between all running instances.

The pending request failure of shared state type $m : n$ marked as $X$ in Figure 4.28. The solution in section 4.3 can be applied because that after the *initiator1* system crash, although there is further interaction between *initiator2* and *responder1*, after the *initiator1* system restart, a request resent from the *initiator*

will be replied with the previous cached result.

## 4.6 Conclusions

*Pending request failure* is a very common interaction failure caused by an initiator system crash. In this section, we have presented a solution to the pending request failure, i.e., we have proposed a robust behavior for the involved initiator and responder instances such that state inconsistencies will be resolved in case of occurrence of such failures.

The recovery from a pending request failure depends on the four shared state types, i.e., how state information is shared between the process instances at runtime. The four shared state types are:

*1 : 1 shared state type*. The state information is shared between one initiator instance and one responder instance. The recovery method is that the initiator resends the request message while the responder uses a copy of the previous result as a response without reprocessing the duplicate request message.

*n : 1 shared state type*. The state information is shared between multiple initiator instances and one responder instance. The difficulty is that if one initiator system crashes, the interaction between other running initiator instances and the responder instance may further change the responder state and overwrite the previous interaction result, which makes using the previous result as a reply impossible. The recovery method is to cache the response message when the responder system state changes, and to use the cached response message as a response for a resent request message due to failure. This is a general solution that can be applied to *1 : 1* shared state type. However, the cache related operations make the solution performance lower (see Table 8.1 in chapter 8) than the solution proposed for *1 : 1 shared state type*.

*1 : n shared state type* The state information is shared between one initiator instance and multiple responder instances. The solution of state type *1 : 1* can be applied because if initiator system crashes, all responder instances cannot interact with the corresponding initiator instances. Thus one interaction does not affect another interaction, just the same as the case of *1 : 1* shared state type.

*m : n shared state type* The state information is shared between multiple initiator instances and multiple responder instances. The solution of state type *n : 1* can be applied because if one of the initiator system crashes, the other initiator instances may interact with the responder instance. Thus one interaction may affect another interaction, just the same as the case of *n : 1* state type.

CHAPTER 5

# Recovery of *pending response failure*

This chapter presents our solution for the *pending response failure*. As introduced in Chapter 1, a pending response failure is caused by a crash of the responder system or by a failure of the network before the response message could be delivered. The solution for this failure depends on shared state types, i.e., how state is shared between business processes and their runtime instances. For each shared state type, we use one section to present our solution.

In this section, several languages/notations are used for different purposes. The UML sequence diagram is used to illustrate the interaction failures and our high level idea of recovery solution. The graphical notation of business process is used to present our recovery solution. The Web Services Business Process Execution Language (WS-BPEL) code is used to illustrate some implementation details. The Petri net models form a basis for correctness validation. The purposes of these languages are shown in Figure 5.1.

This chapter is further structured as follows. The pending response failure is introduced in Section 5.1. Our solution for state type *1 : 1* and *n : 1* are presented in Sections 5.2 and 5.3. The solutions for the shared state types *1 : n* and *m : n* are discussed in Section 5.4 and 5.5. This chapter is concluded in Section 5.6.

## 5.1 Pending response failure

The pending response failures are marked as $X_S$ and $X_N$ in Figure 5.2. $X_S$ is a pending response failure caused by a responder system crash. $X_N$ is caused by a network failure. In both cases, the response message is not sent or lost.

According to our experimental experience, as is shown in Table 5.1, the initiator is aware of the failure by waiting until timeout while the responder continues execution after a restart. The experiment entials two process engin-

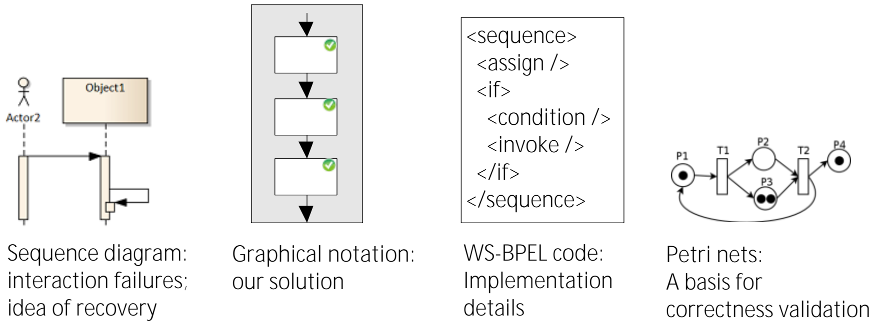
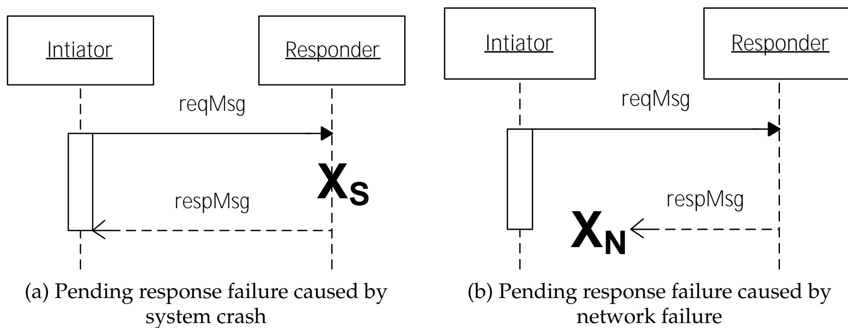



Figure 5.1: Purposes of languages/notations




(a) Pending response failure caused by system crash

(b) Pending response failure caused by network failure

Figure 5.2: Pending response failure

ers, namely Apache ODE and Oracle SOA Suite. The two process engines are set up in two physical systems. In the first experiment, the Oracle SOA suite is the client while the Apache ODE is the server. In the second experiment, the Apache ODE is used as client while the Oracle SOA suite is the server. On the client side, a catchable exception is thrown at process language level. However, on the responder side, the two famous process engines will send the response message and continue execution (after a restart in the case of responder system crash), although the connection is lost and the initiator cannot receive the response message.

The pending request failure can be produced as follows.

1. Initiator sends a synchronous request message.

Table 5.1: Process Engine Behavior under Pending Response Failure

| Process engines | Server crash | Server restart |
|---|---|---|
| Client: Oracle SOA suite | Client instance exception | Client instance exception not handled |
| Server: Apache ODE | Server crashes | Server instance complete without error |
| Client: Apache ODE | Client instance waiting until timeout | Client instance exception not handled |
| Server: Oracle SOA suite | | |

2. Responder receives the request message, starts processing while initiator waits for the response.

3. Responder system crashes or a network failure happens. In both cases, the established network connections are aborted and the expected response message will not be delivered to the initiator.

4. Initiator waits until timeout, then an exception is thrown, which is catchable at process language level.

5. In case of a responder system crash, responder sends the response message and continues execution after a restart of the responder system.

## 5.2 Pending response failure recovery for shared state type *1 : 1*

The concept of the *1 : 1* shared state type is shown as Figure 5.3. At runtime, the initiator process may have three running instances, represented as ellipses. The responder process may have three running instances, represented as circles. Each initiator process instance synchronizes its state with a single responder instance (1-1) and vice versa. The state information is *1 : 1* to the initiator-responder instance pair.

As is shown in Figure 5.4a, a responder design of nested activities between the request and the response increases the possibility of the pending response

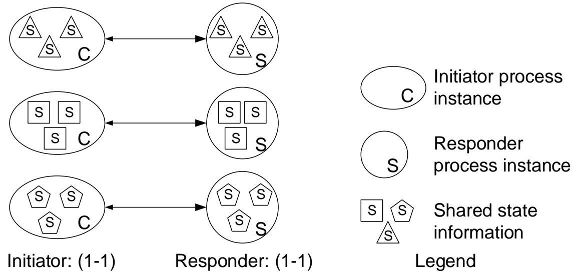
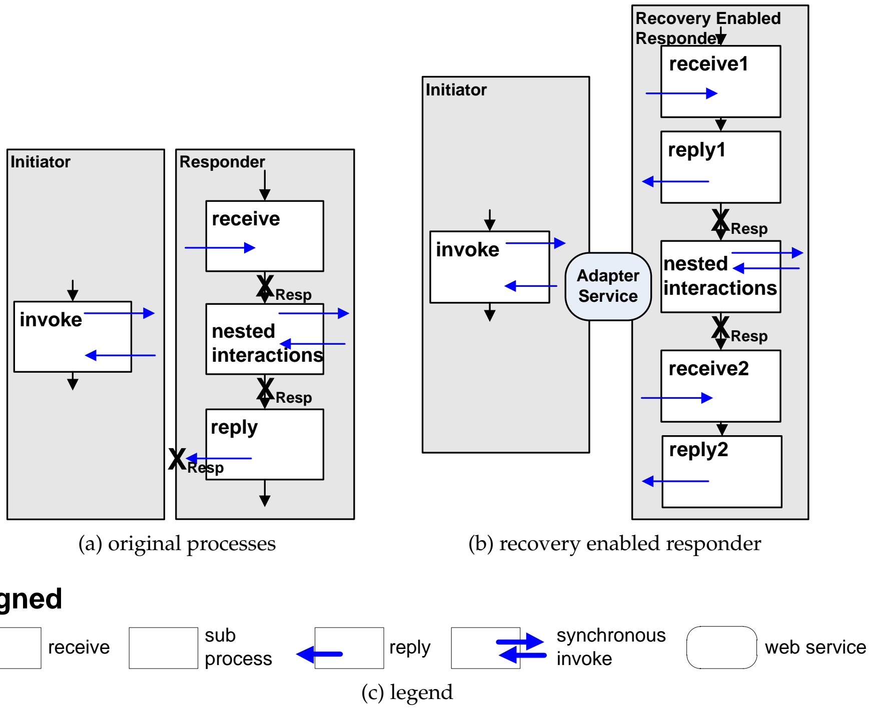



Figure 5.3: *1 : 1* state type




(a) original processes

(b) recovery enabled responder

(c) legend

Figure 5.4: An overview of our recovery method

failure. It receives a request, processes it using some activities nested between the request and the response. System crashes or network failures may happen before sending the response. The responder system crash will halt the execution of the responder process. The established connection is aborted and the response message is lost.

Our idea is, in order to avoid the process design of nesting activities between the request and the response, we split one synchronous interaction into two, as is shown in Figure 5.4b. On the responder side, one synchronous interaction ("receive1" and "reply1") is to receive the request parameters from the initiator. The other ("receive2" and "reply2") is to return the response to the initiator. Then a failure during the execution of any nested activities will not interfere the execution of the initiator process because there are no open connections between the process instances.

The above process transformation changes the responder process interface, i.e., it changes the message formats and sequences. In order to hide the changes that are necessary to the responder from the initiator, we use an adapter service to make the initiator still use the same interaction protocol while the responder has an adapted interaction protocol. The adapter service receives the request from the initiator, interacts with responder and sends a response back to initiator. We will present the detail of the design and deployment of adapter services in Section 5.3.

### 5.2.1 Pending response failure model

The Petri net model is used for the correctness validation in section 8.

The petri model of the collaborative processes with a pending response failure is shown as Figure 5.5. The transitions $t_ch1$ and $t_ch2$ represent the communication channel between initiator and responder. The left hand side models *invoke* activity of an initiator. The transitions $t_inv1$ and $t_inv2$ model an *invoke* activity that the initiator sends and receives the request and response message respectively. The place $p_v1$ represents the input variable of the *invoke* activity. The place $p_v2$ represents the output variable of the *invoke* activity. The transitions $t_rec$ and $t_rep$ is the model that the responder receives and replies the request and response message respectively. The place $p_v3$ represents the output variable of the *receive* activity. The place $p_v3$ represents the input variable of the *reply* activity. The places $p_m1, \ldots, p_m4$ represents the messages and the places $p_c1, \ldots, p_c7$ represent the process control flow. The transition $t_any$ can be replaced by any sub Petri net to represent the processing of the request message.

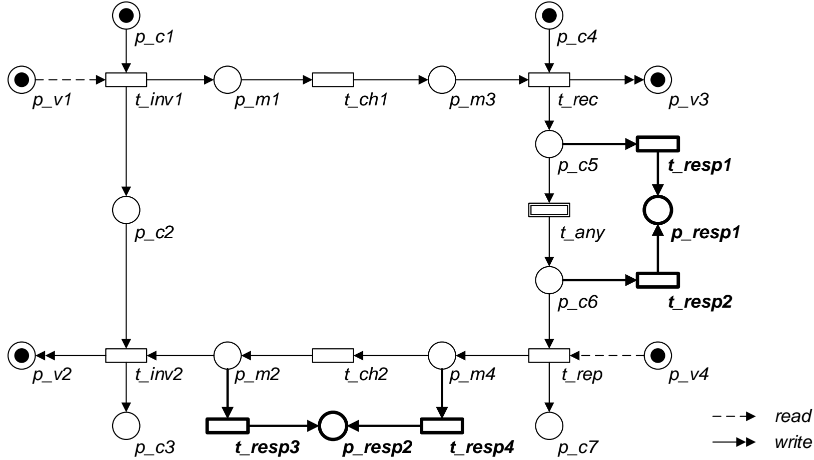



Figure 5.5: Petri net model of collaborative processes with a *pending response failure*

The responder system crash is represented as transitions $t_resp1$ and $t_resp2$. If these transitions fire, a token is taken from the place $p_c5$ or $p_c6$ to represent that the responder has deviated from the normal control flow. A token is put into the place $p_resp1$ to indicate that pending response failure happens.

Another case of pending response failure is caused by a network failure, which is represented as transitions $t_resp3$ and $t_resp4$. A token is taken from $p_m2$ or $p_m4$ to represent the response message loss. A token is put into the place $p_resp2$ to indicate that the pending response failure happens.

### 5.2.2 The robust process model

The Petri net model of the transformed process is presented in Figure 5.6. The model is divided into three parts: the part depicted to the left of transitions *t_ch1* and *t_ch2* corresponds to the initiator process. The part between *t_ch1* and *t_ch2* and *t_ch3* to *t_ch6* corresponds to the adapter service, and the part to the right of *t_ch3* to *t_ch6* corresponds to the transformed responder. In order to avoid the process design of nesting activities between the synchronous request and response, we split one synchronous interaction into two, so that no nested task remains between the synchronous request and response. The first synchronous interaction (represented by transitions *t_inv3, t_rec2, t_rep2, t_inv4*) is to send the request parameters from initiator to responder. The second syn-

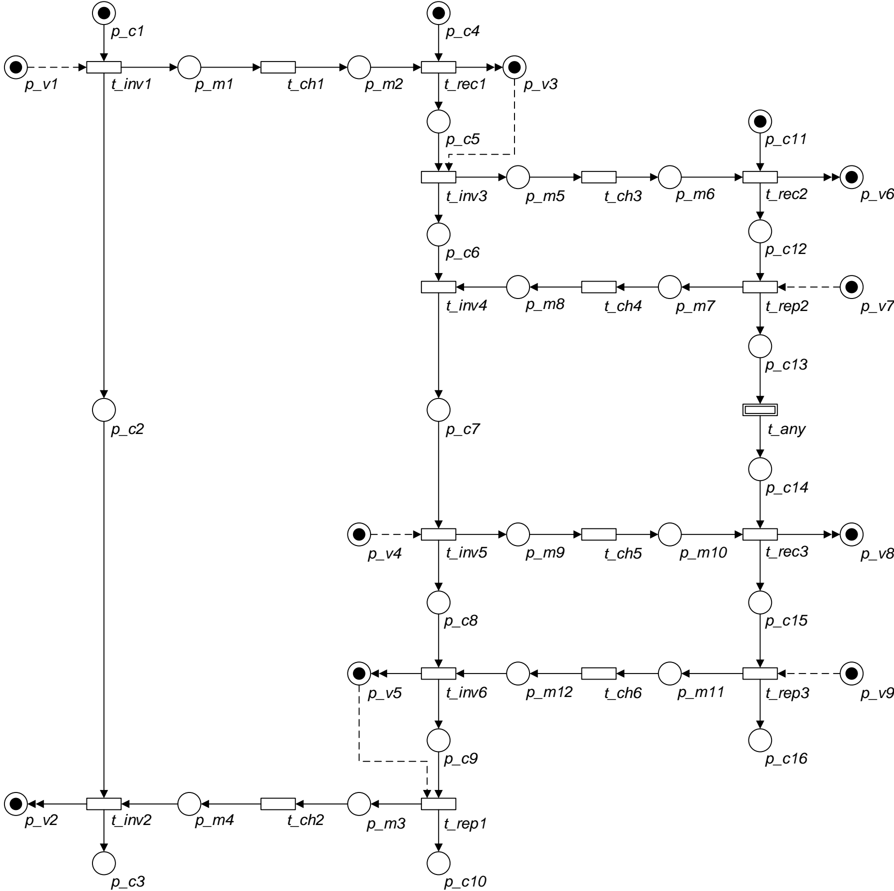



Figure 5.6: Petri net model of collaborative processes with *pending response failure* recovery

chronous interaction (represented by transitions *t_inv5*, *t_rec3*, *t_rep3*, *t_inv6*) is to reply the result from the responder to the initiator process. The adapter process is used to keep the original process interface for the initiator process. The adapter service receives the request from the initiator, interacts with responder and sends a response back to the initiator. Since the responder process server may crash, the adapter service should be deployed together with the initiator.

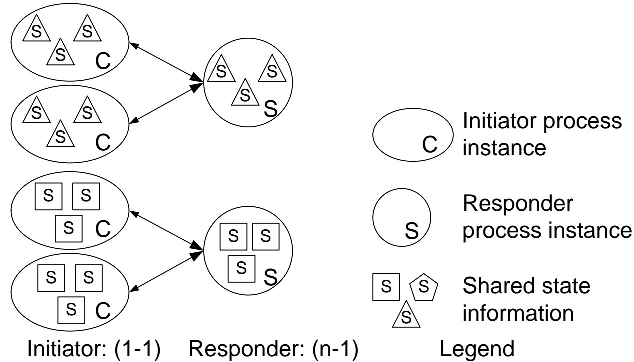



Figure 5.7: *n : 1* state type

(A robust adapter design will be presented in the next section).

Each synchronization request gets an immediate reply. At the responder side, each synchronous interaction consists of a receive immediately followed by a reply so that we expect no pending response failure happens during such an interaction. If, however, a pending response failure happens during any of the synchronous interactions, it is not recoverable. We can assume that the possibility of pending response failure during one of the synchronous interactions in the collaborative process is low.

## 5.3 Pending response failure recovery for shared state type *n : 1*

In the previous section, we have considered the scenarios where the effects of the state changes in one collaboration between an initiator process instance and a responder process instance do not affect other collaborations. In this section, we focus on a responder process instance collaborating with multiple initiator process instances, where one collaboration may affect other collaborations.

The concept of the *n : 1* shared state type is shown as 5.7. The initiator process may have four running instances, represented as ellipses. The responder process may have two running instances, represented as circles. Each initiator process instance synchronizes its state with a single responder instance (1-1), while one responder instance interacts with multiple initiator instances (n-1).

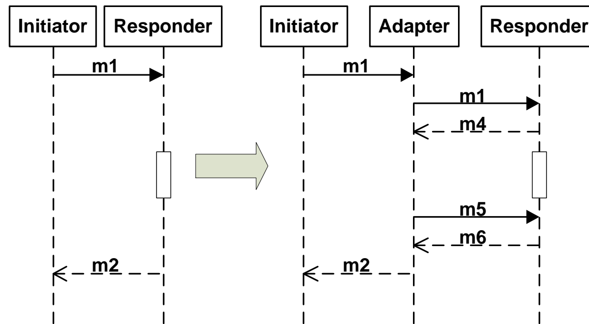



Figure 5.8: Recovery solution for shared state type *1 : 1*, as proposed in the previous section

The state information is shared between initiator instances.

The solution in the previous section cannot be applied because of a potential message queue overload on the responder side. As is shown in Figure 5.8, the lefthand diagram is the original collaborative process, and the righthand diagram is the transformed collaborative process (i.e., with pending response failure recovery, as proposed in the previous section). After sending the response message $m4$, the responder starts processing the request message. However, the adapter sends the message $m5$ immediately after it receives $m4$. The responder may not be ready to receive this message until the nested processing is finished. Under heavy workload, the queued messages may lead to server buffer overload. Another case of message queue overload may occur during the processing of the request message, if multiple initiators send request messages concurrently. At a technical level, this problem appears as an engine warning shown in Figure 5.9. If the message queue is made persisted in a stable storage, the possibility of the message queue to exhaust the storage space is low. However, under the assumption of a possible responder failure, it is not advisable to keep a large message buffer. If the responder system restarts after a system crash, the responder message queue can be recovered from the persistent storage. However the network connection is lost and the response messages cannot be delivered to the initiator. The initiator either fails (throws an exception) or waits forever for the server response.

Our idea of recovery is to transform the original process into a recovery-enabled process [95]. On the initiator side, the main idea of the transformation is to resend the request message whenever a pending response failure is de-

```
11:19:38,888 INFO  [BpelServerImpl] Registered pro
cess {http://de.fhg.ipsi.oasys.businessScenario.sa
mple.transformed}logistic.transformed-97.
11:19:38,889 INFO  [DeploymentPoller] Deployment o
f artifact logistics.transformed successful: [{htt
p://de.fhg.ipsi.oasys.businessScenario.sample.tran
sformed}logistic.transformed-97]
11:20:10,957 WARN  [BpelRuntimeContextImpl] A mess
age arrived before a receive is ready for a reques
t/response pattern. This may be processed to succe
ss. However, you should consider revising your pro
cess since a TCP port and a container thread will
be held for a longer time and the process will not
 scale under heavy load.
```

Figure 5.9: Potential buffer overflow warning from a WS-BPEL engine

tected. On the responder side, we parallelize the processing of the request message and the initiator query for the processing result. The transformation adds a caching capability, i.e., the response message for a newly incoming message representing a non-idempotent operation is cached. If the responder receives a resent message from the initiator due to a failure, the responder replies the cached response message and does not execute the operation again. In the following we discuss the transformed initiator and responder processes.

### 5.3.1 The robust initiator process

The initiator starts a state synchronization by executing an *invoke* activity. The transformed *invoke* activity is shown as Figure 5.10. The *invoke* activity is nested within a *scope* activity with an exception handler. If a synchronization failure happens (marked as $X_{SU}$, $X_{Req}$, $X_{Resp}$), the exception handler in the scope can be executed, i.e., adding a delay before retrying using a *wait* activity. A interaction failure can be delivered to the process layer as an exception which is catchable by the exception handlers. This is supported by the process engines such like Apache ODE. The *scope* activity is inside a *while* activity to implement a retry behavior. If the *invoke* activity finishes without failure, an *assign* activity is used to alter the value of the variable $\$retry$ to $0$ to end the while iteration.

### 5.3.2 The robust responder process

The original responder process is shown in Figure 5.11a. The process receives an order message, processes it, and sends a response. The robust responder

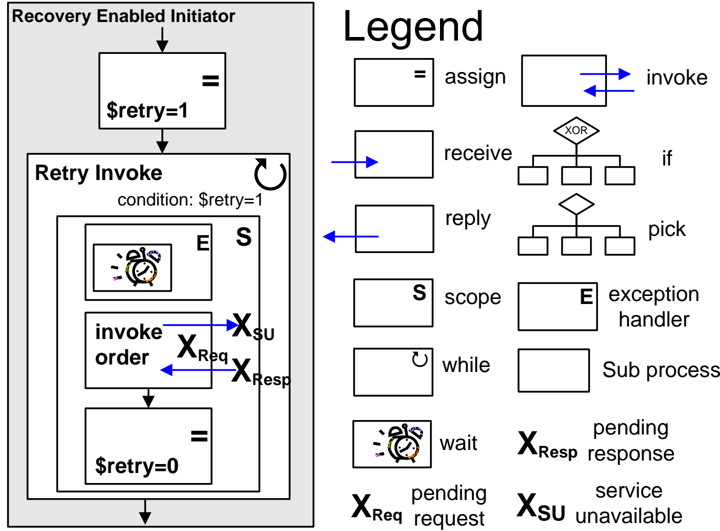
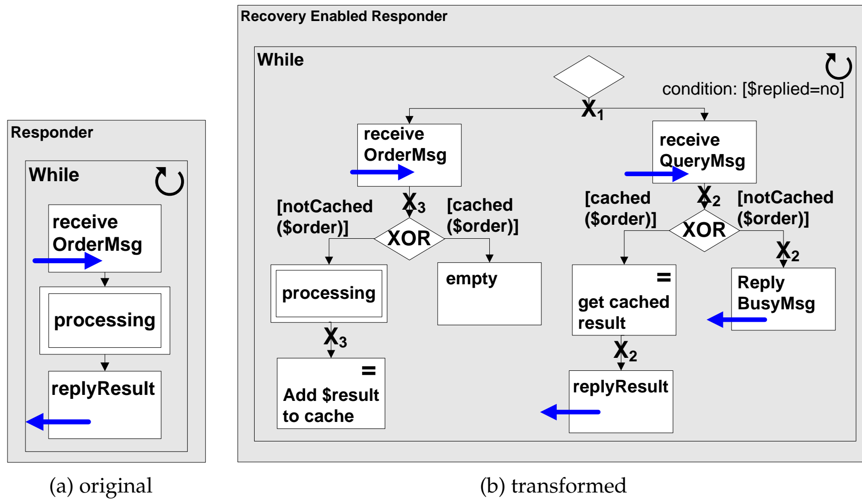



Figure 5.10: Transformed Initiator Process

process is shown in Figure 5.11b. The original synchronous interaction is split into an asynchronous request to send the original order message, and a request response pattern to query the response, thus the result of the original request.

There are two steps corresponding to handling the asynchronous request and handling the request-response to query the response to the original request. In the first step, the responder receives a message $OrderMsg$ (the left branch of the *pick* activity in the *while* iteration in Figure 5.11b). This is a one-way message, so that the responder does not send a response, thus pending request or response failure are avoided. After receiving the order message, an *if* activity is used to check whether the order is cached. If the order is cached, this implies that the order has been processed before and this is a resent message due to failure. In this case the responder does nothing (*empty* activity). If the order is not cached, the responder processes the order and adds the result to the cache.

In the second step, the responder receives the query message from the initiator. The WS-BPEL correlation mechanism can be used to correlate the query message with the corresponding order message. If the order is cached, the responder will use the cached response message as a reply. If the request is not




(a) original (b) transformed

Figure 5.11: Responder Process

cached, the responder sends a message $BusyMsg$ to indicate to the initiator that the processing is not finished.

The two steps are placed via a pick activity in a while iteration to support the interaction with multiple client instances and retries per instance.

After the transformation, the possible types of failures are marked as $X_1$, $X_2$ and $X_3$. Failure $X_1$ is a service unavailable failure and can be compensated by the transformed initiator's resend of the request message. Failure $X_2$ is a pending response failure. The initiator can detect the failure by not receiving the response message and recover by resending the query message. On the responder side, the resent message is replied with a cached response. Failure $X_3$ happens in a control flow outside a synchronization block, thus, this error does not affect the state synchronization with the initiator.

### Implementation of Cache Related Operations

The cache is declared as a process variable in WS-BPEL with an XML structure of entries. Each entry is a mapping from request message to response message. A sample cache entry is shown below.

```
<cache>
```

```
 <cacheEntry>
  <requestEntry>
   <requestMsg />
  </requestEntry>
  <responseEntry>
   <responseMsg />
  </responseEntry>
 </cacheEntry>
 <cacheEntry .../>
</cache>
```

The cache operations are pre-defined using XSLT to operate on the cached XML data. The invocation of cache operation is an “assign” activity. We use the standard WS-BPEL function $doXslTransform()$ in the assign activity with a pre-defined XSLT script to manage the cache. A sample read cache activity is shown in the following listing.

```
<bpel:assign>
 <bpel:copy>
  <bpel:from>
   bpel:doXslTransform(
   "testCached.xsl", $cache,
   "requestMsg", $requestMsg)
  </bpel:from>
  <bpel:to>$foundCachedRequest</bpel:to>
 </bpel:copy>
</bpel:assign>
```

### Adapter Design

In the solution defined so far, there is a mismatch between the interaction patterns expected by the transformed responder and the interaction pattern of the transformed initiator. To solve this problem, the initiator is further transformed. In case it is not possible to modify the initiator, we have to place an adapter between the initiator and the responder to mediate this mismatch.

The design of the adapter process is shown in Figure 5.12. The adapter receives an *order* message from the initiator. In the following *while* iteration, it forwards the message to the responder (activity *invoke1*). Then it sends the query message to ask the responder for the result (activity *invoke2*). If the *result* message has been sent with the reply activity at the responder, and received by the initiator ([$\$reply = result$]), the *while* iteration is terminated by changing

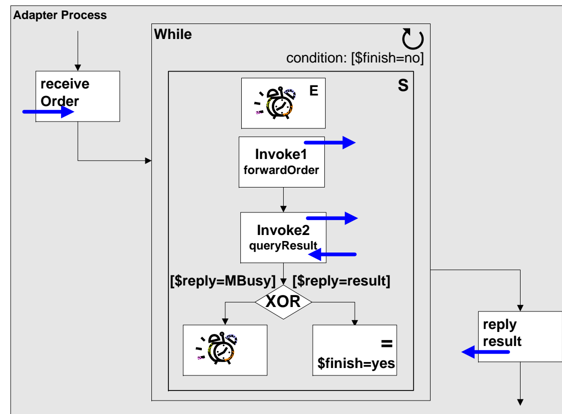



Figure 5.12: Adapter Process Design

the value of the conditional variable $\$finish$ to $yes$. If the result message is $MBusy$, which indicates that the responder is still processing, a $wait$ activity is executed to introduce a delay and then the outer *while* iteration sends the *query* message again. Finally, the *result* message is accepted by the initiator, possibly after multiple queries. Both *invoke* activities are defined in the while iteration, because the first *invoke* activity is a one-way message exchange, which is non-blocking. If the first *invoke* activity would have been defined outside the while iteration, it is possible that the network delivers the second message (query) before the first message (order).

From the initiator point of view, the adapter is designed as a stateless process: each client request triggers a new adapter instance creation. In the case of an adapter failure, this is a pending response failure from the client point of view. This triggers the client to resend the request message, which creates a new adapter instance to fulfill the synchronization.

### Design Considerations of Generic Adapter

If we deploy a separate adapter process for each specific initiator and responder, lots of adapter instances are created, which will probably increase the processing overhead. Another option is to re-use a generic adapter for all initiators and responders. Three related design considerations are discussed in the following. First, the messages delivered to and from adapters should be

independent from the initiator and responder processes. The parts of the message should refer to a generic typed element or declared as a generic type, such as "xsd:anyType". The drawback is due to the correlation mechanism of WS-BPEL. In particular, for different messages different correlation set configurations are required. This makes it necessary to distinguish messages, which is not the case with anyType. Second, the responder process should describe its operations in a process-independent way. The generic adapter should not refer to any responder specific operations, e.g., process specific port type definitions in their WSDL. The drawback of using generic message type such as "xsd:anyType" is that the responder cannot use the control flow branching activities (like "pick" in WS-BPEL), because all messages are generic types and dedicated to generic process operations. Finally, a generic addressing mechanism is required to forward an incoming message to a proper responder, for example, by mapping specific information of an incoming message to the address of the responder. This can be achieved by using a mapping from message to responder addresses in XSLT. In WS-BPEL, the function doXslTransform(inMsg, XSLT) can be used to query the responder address.

From the above discussion, we can conclude that the possibility of using a generic adapter is quite limited. On the other hand, with additional process management effort, a process-specific adapter can be automatically generated from the initiator and responder services descriptions, e.g., their WSDL descriptions.

## 5.4 Pending response failure recovery for shared state type *1 : n*

The concept of the *1 : n* shared state type is shown as Figure 5.13. At runtime, the initiator process may have two running instances, represented as ellipses. The responder process may have four running instances, represented as circles. Each initiator process instance synchronizes its state with multiple responder instances (1-n), while globally one responder instance interact with one initiator instance (1-1). The state information is private to each initiator instance, but shared between multiple server instances.

The pending response failure of *1 : n* shared state type is marked as $X$ in Figure 5.14, which is the case that the "Responder1" system crash happens. The solution in section 5.2 can be applied because that after the *Responder1* system crash or the response message *RespMsg* is lost, the *initiator* has to wait after a

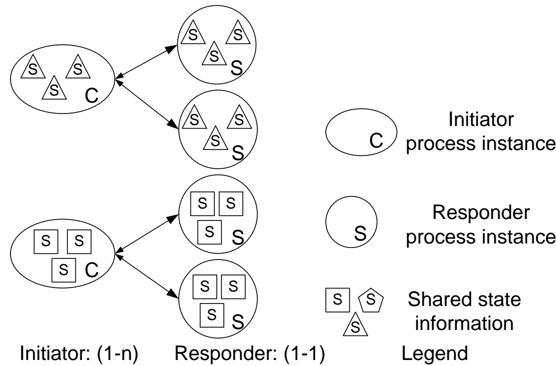



Figure 5.13: *1 : n* shared state type

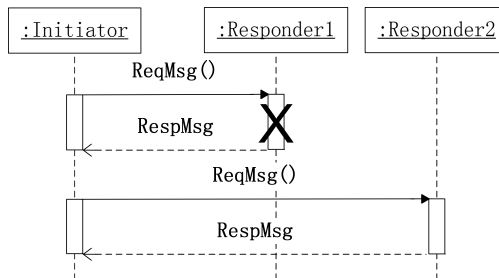



Figure 5.14: Pending response failure of *1 : n* shared state type

timeout and then resends the request message. During this wait-and-resend period, the initiator cannot interact with other responders instances. Thus in this case is the interaction is equivalent to the *1 : 1* shared state type.

## 5.5 Pending response failure recovery for shared state type $m : n$

The concept of the $m : n$ shared state type is shown as Figure 5.15. At runtime, the initiator process may have two running instances, represented as ellipses. The responder process may have three running instances, represented as circles. Each initiator process instance synchronizes its state with multiple

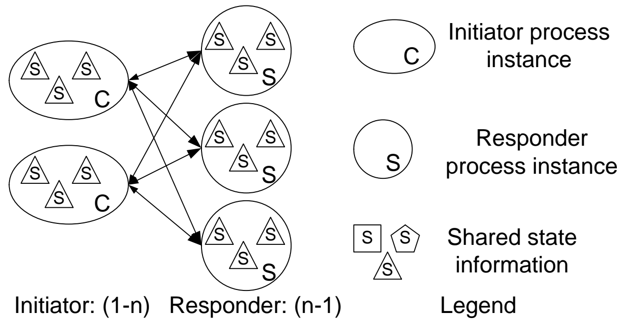



Figure 5.15: *m : n* shared state type

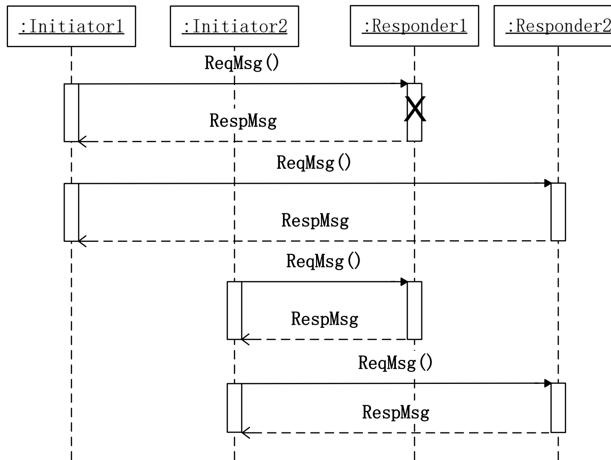



Figure 5.16: Pending response failure of *m : n* shared state type

responder instances (1-n), while each responder instance interact with multiple initiator instance (n-1). The state information is shared between all running instances.

The pending response failure of $m : n$ shared state type is marked as $X$ in Figure 5.16. The solution in section 5.3 can be applied because after the *Responder1* system crash, the *Initiator1* will wait until timeout and then resend the request, thus the interaction between *Initiator1* and other responders is relevant. The other initiators may send requests concurrently, however, this is the

same case that we have solved in section 5.3.

## 5.6 Conclusions

*Pending response failure* is a very common interaction failure caused by a responder system crash or a network failure preventing the delivery of the expected response message to the initiator. In this chapter, the solutions are presented to recover from the pending response failure. The solutions are presented in the form of transformed initiator and responder processes, together forming a robust collaborative process.

The recovery of pending response failure depends on the four state types, i.e., how state information is shared between process instances at runtime. The four shared state types are:

*1 : 1* The state information is shared between one initiator instance and one responder instance. To avoid the crash in the middle of a processing activity nested between receiving a request and replying a response, our approach to recovery is to split the synchronous interaction between the initiator and responder into two interactions. One interaction is used to send the request message and the other interaction is used to return the response message.

*n : 1* The state information is shared between multiple initiator instances and one responder instance. The approach to split one synchronous interaction into two synchronous interactions does work here. The request messages for the second interaction from multiple initiators will accumulate at the responder side, thus leads to the possibility of a message queue overflow and a potential performance problem. Our solution is to parallelize the processing of the request message and the initiator query for the processing result. The transformation adds a caching capability, i.e., the response message for a newly incoming message representing a non-idempotent operation is cached. If the responder receives a resent message from the initiator due to a failure, the responder replies the cached response message and without processing the duplicate request message. This is a general solution that can be applied to *1 : 1* shared state type. However, the cache related operations make the solution performance lower (see Table 8.1 in chapter 8) than the solution proposed for *1 : 1 shared state type*.

*1 : n shared state type* The state information is shared between one initiator instance and multiple responder instances. The solution of state type *1 : 1* can be applied because if pending responder failure happens, the initiator will block and resend the request for the lost reponse message. It will not interact

with other responder instances during the recovery. Thus this failed interaction cannot be affected by other interactions, just the same as the case of *1 : 1* shared state type.

*m : n shared state type* The state information is shared between multiple initiator instances and multiple responder instances. The solution of state type *n : 1* can be applied because if pending response failure happens, the initiator instances will block and resend the request. It will not interact with the other responder instances during the recovery. Thus this failed interaction will not be affected by the initiator's interaction with other responder instances, just the same as the case of *n : 1* state type.

CHAPTER 6

# Recovery of *service unavailable*

This chapter presents our solution for the *service unavailable* failure. As introduced in Chapter 1, a service unavailable failure is caused by a crash of the responder system before receiving the request message or by a failure of the network before the request message could be delivered. As an example, if a user fills an order form and submit for a flight reservation, the network may fail before the request message could be delivered.

In this section, several languages/notations are used for different purposes. The UML sequence diagram is used to illustrate the interaction failures and our high level idea of recovery solution. The graphical notation of business process is used to present our recovery solution. The Web Services Business Process Execution Language (WS-BPEL) code is used to illustrate some implementation details. The Petri net models form a basis for correctness validation. The purposes of these languages are shown in Figure 6.1.

This chapter is structured as follows. The *service unavailable failure* is introduced in section 6.1. Section 6.2 presents our recovery mechanism and the whole chapter is concluded in section 6.3.

## 6.1 Service unavailable failure

The service unavailable failure is depicted in Figure 6.2. $X_S$ is a service unavailable failure caused by a responder system crash before receiving the request message. $X_N$ is caused by a network failure before the request message could be delivered. In both cases, the request message is lost.

The Petri net model of the service unavailable failure is shown in Figure 6.3. The transitions $t_ch1$ and $t_ch2$ are the model of the communication channel between an initiator and a responder. The left hand side of the model represents the *invoke* activity of the initiator. The places $p_v1, \ldots, p_v4$ represent the

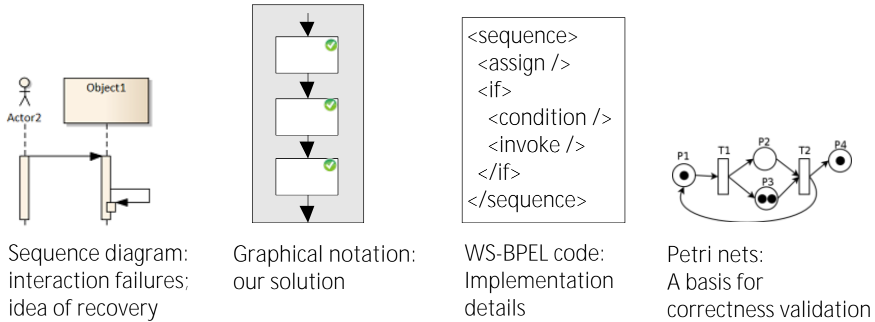



Figure 6.1: Purposes of languages/notations

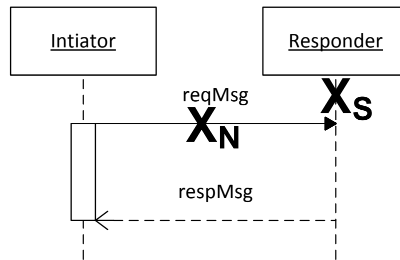



Figure 6.2: Service unavailable failure

four process variables. The places $p_m1, \ldots, p_m4$ represent the messages exchanged between the initiator and the responder, and the places $p_c1, \ldots, p_c7$ represent the process control flow. The transitions $t_inv1$ and $t_inv2$ model that the initiator sends and receives the request and the response message respectively. The transitions $t_rec$ and $t_rep$ model that the responder receives and sends the request and response message respectively. The transition $t_any$ can be replaced by any sub Petri net to represent the processing of the request message by the responder process.

The responder system crash is represented as transition $t_su3$. If this transition fires, a token is taken from the place $p_c4$ to represent that the responder has deviated from the normal control flow. A token is put into the place $p_su2$ to indicate that the service unavailable failure happens.

Another case of service unavailable failure is caused by a network failure, which is represented as transitions $t_su1$ and $t_su2$. A token is taken from

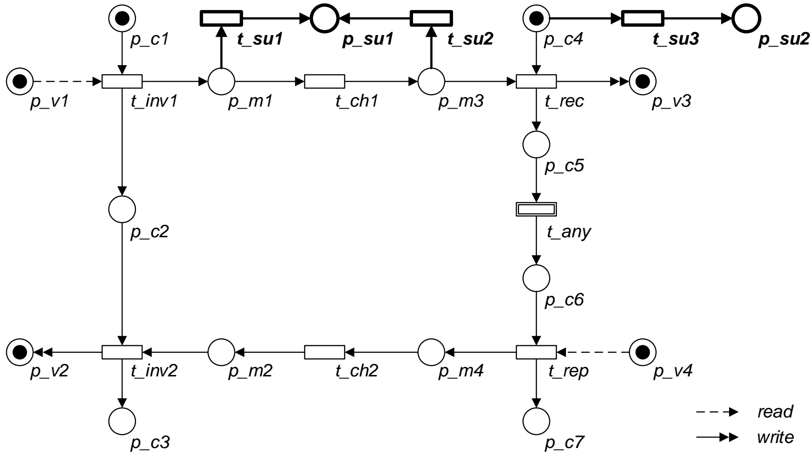



Figure 6.3: Petri net model of service unavailable failure

$p_m1$ or $p_m3$ to represent the request message loss. A token is put into the place $p_su1$ to indicate that the service unavailable failure happens.

## 6.2 Service unavailable failure recovery

The recovery solution for service unavailable is the same for all shared state types. The responder does not have to be changed. On the initiator side, our idea of recovery is to make it resend the request whenever this failure happens. In the WS-BPEL context, an *invoke* activity of the initiator can be transformed as shown in Figure 6.4 to achieve a request resend behavior as long as service is unavailable (i.e., the response message is not received). Inside a *while* iteration called "Retry Invoke", we put a *scope* activity (a box annotated with *S*) with a exception handler (a box annotated with *E*). The exception handler is implemented with a *wait* activity. When the target service is not available, the *invoke* activity of the initiator will throw an exception, which is caught by the exception handler. The exception handler of the initiator will delay the execution of the process. The outside *while* iteration will repeat the sending of the request until the response message has been received. After the successful receipt of the request message, an assignment of variable $*retry* to 0 will end the *while* iteration. One implementation issue is that by default some process

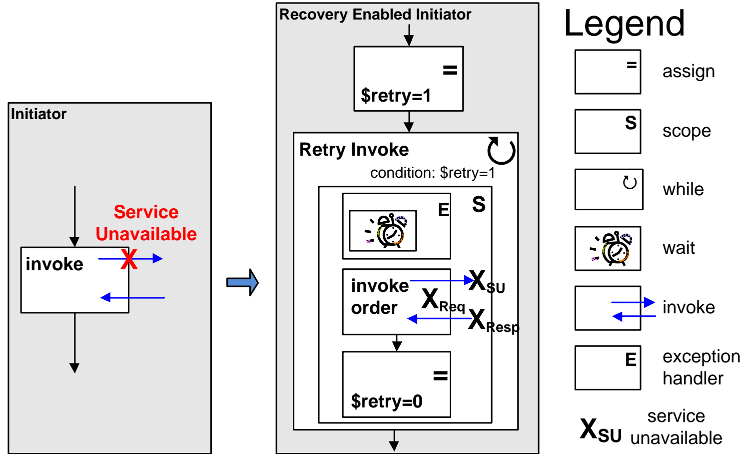



Figure 6.4: An overview of our recovery method for the service unavailable failure

engines may not propagate the message delivery fault as an exception to the initiator process layer. The process may not be aware of this fault. In this case, some engineering effort is necessary to propagate the fault as an exception to the process behavior.

The Petri net model of the transformed process which is able to recover from service unavailable failure is shown as Figure 6.5. In the initial marking, we put a token in both places $p_c1$ and $p_c4$ to represent the beginning of the control flow of the processes. We put a token in each of the places $p_v1$ to $p_v4$ to represent that the process variables are initialized. If the target service is available, transitions $t_inv1$, $t_ch1$ and $t_rec$ occur. If the target service is not available, transitions $t_su1$, $t_su2$ or $t_su3$ fire. A token is put into place $p_su1$ or $p_su2$. Transitions $t_resend$ and $t_restart$ fire to reset the marking of the Petri net model to the initial marking, and the recovery work is finished.

## 6.3 Conclusions

*Service unavailable failure* is a very common interaction failure caused by a responder system crash or a network failure during the sending of a request message. A lot of research and industrial effort have been put to improve the availability of services [96, 97, 98]. However, even a cloud infrastructure crashes

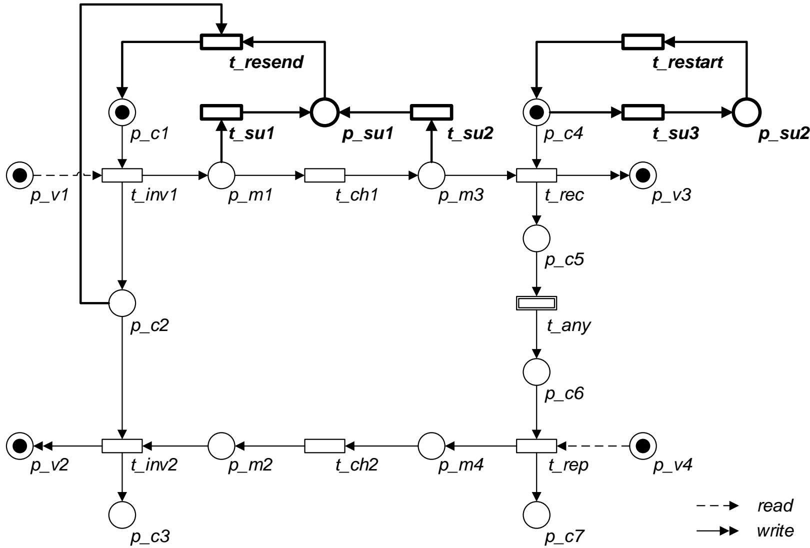



Figure 6.5: Petri net model of the collaborative process with recovery from a service unavailable failure

sometimes, causing serious availability issues [99].

In this section, the service unavailable failure of during the interaction between an initiator and a responder process is studied and a solution is presented. The idea is based on request message resending. Whenever an initiator sends a request message, the message sending activity will be repeated until the target service is available. The idea is pretty straightforward, however, we study the service unavailable in the context of service collaboration and present the failure and the corresponding solution in a formal way.

CHAPTER 7

# Composition of recovery solutions

The solutions for each of the different interaction failure types addressed in this thesis are presented separately in Chapters 4, 5 and 6. However, more than one type of interaction failure may happen in a single service collaboration. In this section, we discuss how the solutions for different interaction failures are applied in combination to a single service collaboration, i.e., how the solutions in chapters 4, 5 and 6 can be composed to derive robust processes. The solutions differ depending on the shared state types, which are described in subsection 3.2. The solution of pending request failure for state type *n : 1* is described in section 4.3. The solution of pending response failure for state type *n : 1* is described in section 5.3 and the solution of service unavailable is described in chapter 6. We show the combined solutions for shared state type *n : 1* in this chapter. This solution can be applied to shared state type *m : n* as well, as discussed in Sections 4.5 and 5.5. For state type *1 : 1* and *1 : n*, they are simpler case that this solution can be applied, as discussed in sections 4.6 and 5.6.

In this chapter, we apply the solutions one by one to show the resulting robust processes and how they recover from different types of interaction failures. This section is structured as follows, we compose the solution for pending request failure and service unavailable in Section 7.1. The pending response failure solution is incorporated in Section 7.2. A working example is presented in Section 7.3. Some general robust service interaction design principals are discussed in section 7.4 and we conclude the chapter in Section 7.5.

## 7.1 Composed solutions: pending request failure and service unavailable

Figure 7.1 shows the process behavior of a synchronous interaction, which is represented by a *receive* activity ("receiveM1") and a *reply* activity ("receiveM2").

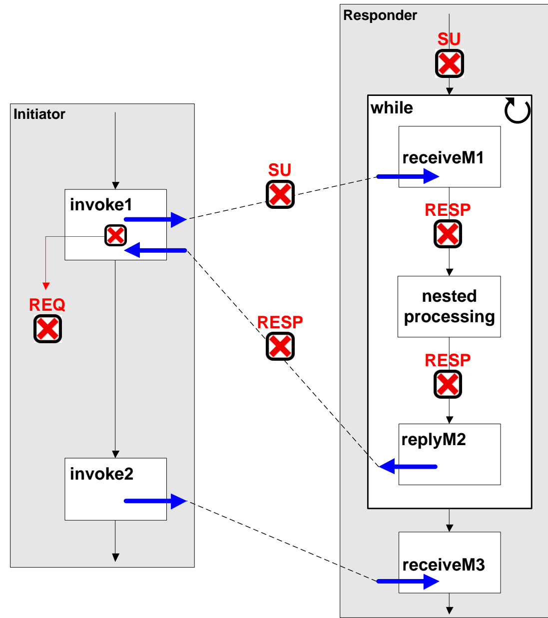



Figure 7.1: Original processes

The synchronous interaction is nested in a while iteration to represent the *n : 1* shared state type that multiple initiator instances may interact with this responder instance. This synchronous interaction is followed by an arbitrary further interaction, represented by a one-way message "receiveM3". The pending request failure, pending response failure and service unavailable are marked as *REQ*, *RESP* and *SU* respectively. *REQ* is the initiator system crashes after the request message is sent and before receipt of the response message. *RESP* is the pending response failure that the responder system crashes after receipt of the request message or the network fails during the sending of the response message. *SU* is the service unavailable failure that the responder system crashes before receipt of the request message. If we apply the solution for pending request failure (Section 4.3), the robust process is shown as Figure 7.2. On the initiator side, the recovery is to make it resend the request whenever this failure happens. An *invoke* activity of the initiator can be transformed as shown in Figure 7.2 to achieve a request resend behavior as long as service is unavailable (i.e., the response message is not received). Inside a *while* iteration called

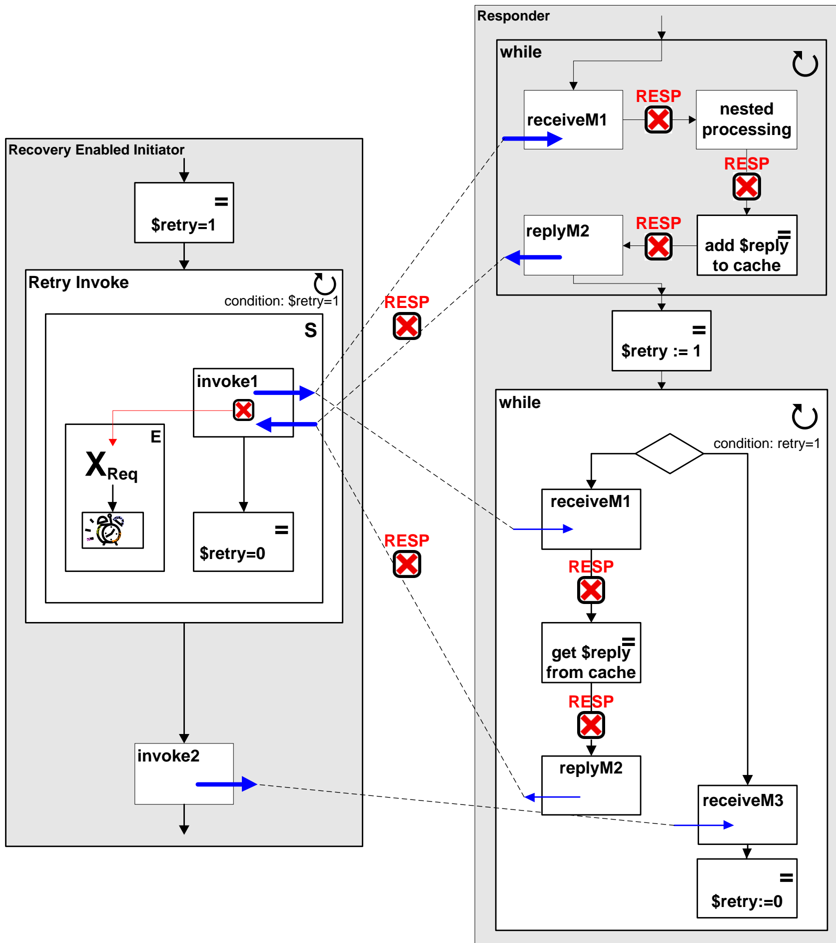



Figure 7.2: Robust processes, recoverable from pending request failure

"Retry Invoke", we put a *scope* activity (a box annotated with *S*) with a exception handler (a box annotated with *E*). The exception handler is implemented with a *wait* activity. When the target service is not available, the *invoke* activity of the initiator will throw an exception, which is caught by the exception handler. The exception handler of the initiator will delay the execution of the

process. The outside *while* iteration will repeat the sending of the request until the response message has been received. On the responder side, the *receive* activity "receiveM3" is replaced by a *while* iteration with a *pick* branch in it. The *while* iteration is used to process the possible request resent by the initiator due to failure. On the left hand side of the *pick* branch, the activities "receiveM1" and "replyM2" are used to respond to the request resent from the initiator, where the response message is read from cache using an *assign* activity. On the right hand side of the *pick* branch, the activity "receiveM3" represents possible processes further interaction. In case that the message for a further interaction is sent and the *receive* activity "receiveM3" is executed, this implies that the initiator has received the previous response. We then assign the process variable $\$retry := 0$ to end the *while* iteration. Furthermore, the solution of service unavailable as described in Chapter 6 is applied as well.

## 7.2 Composed solutions: pending response failure

The solution presented in Figure 7.2, can recovery from pending request failure and service unavailable, however, it cannot recovery from pending response failure, marked as *RESP*. We apply the solution of pending response failure (Section 5.3) and the robust responder process is shown as Figure 7.3. The activities that correspond to the synchronous interaction (the activities between *receiveM1* and *replyM2*) are replaced by a *pick* activity where the recoverable solution for pending response failure is applied. The original synchronous interaction is split into an asynchronous request to send the original order message, and a request response pattern to query the response, thus the result of the original request. There are two steps corresponding to handling the asynchronous request and handling the request-response to query the response to the original request. In the first step, the responder receives a message *receiveM1* (the left branch of the *pick* activity in the *while* iteration in Figure 7.3). This is a one-way message, so that the responder does not send a response, thus pending request or response failure are avoided. After receiving the request message *M1*, an *if* activity is used to check whether *M1* is cached. If *M1* is cached, this implies that the message has been processed before and this is a resent message due to failure. In this case the responder does nothing (*empty* activity). If the order is not cached, the responder processes the request message *M1* and adds the result to the cache. In the second step, the responder receives the query message from the initiator. If the request message *M1* is cached, the responder will use the cached response message as a reply. If the request message is not

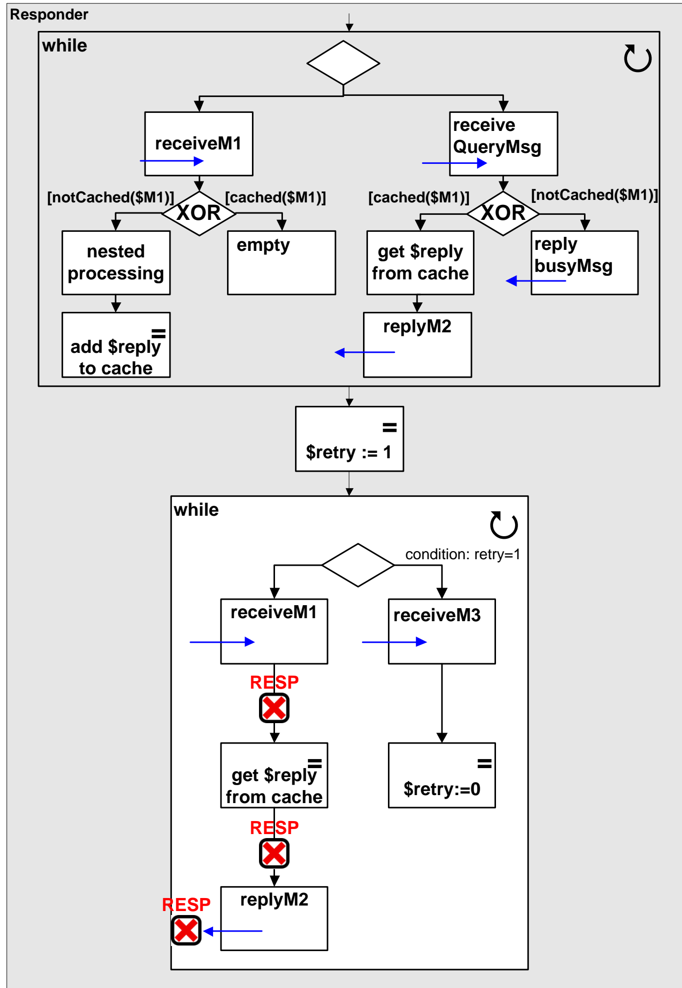



Figure 7.3: Robust responder process with recovery solutions combined

cached, the responder sends a message $BusyMsg$ to indicate to the initiator that the processing is not finished. The two steps are placed via a pick activity in a while iteration to support the interaction with multiple client instances and retries per instance.

What should be mentioned is the cache related activities. As the same cache capabilities are used both in pending request failure and pending response failure, the composed solution use one cache, which is shared for all cache related activities.

We can see the responder process mainly consists of two *while* iterations. The above *while* iteration is used mainly to recover from pending response failure. The left side is an asynchronous interaction to send the request parameters. The right side is a synchronous interaction, which is a query operation in order to get the operation result. The bottom *while* iteration is used to recovery from pending request failure. However, in the bottom *while* iteration, the left side has a read cache operation and uses this cache to generate a reply. We do not further apply the recovery solution for pending response failures of this synchronous interaction marked as *RESP* in Figure 7.3. If pending response failure happens, the initiator will resend the request message to recover. Furthermore, we assume that a cache query operation takes less time than processing the request, which will lower the chance that the pending response failure happens.

After this step of the process transformation, an adapter process should be applied between the initiator and the responder, because the messages sequences and formats does not match each other. The same adapter from 5.3 can be applied here.

## 7.3 An example scenario

We illustrate our approach with a scenario shown in Figure 7.4 of a simple procurement process in a virtual enterprise. It contains three business partners: a buyer, an accounting department and a logistics department. The accounting department gets a “getQuote” message from the buyer and returns a “quote” message. The “order” information (“deliver” message) is forwarded by the accounting department to the logistic department. The logistic department then confirms the receipt (“deliverConf” message with expected delivery date and parcel tracking number) to the accounting department. The accounting department forwards a “delivery” message to the buyer. Furthermore, the buyer can decide to track the status or terminate the process (messages “getStatus”, “status”, or “terminate”). The process definition of the accounting department is shown in Figure 7.5.

The scenario starts by receiving a synchronous invocation “getQuote” message requiring the quote information. After the accounting process replies

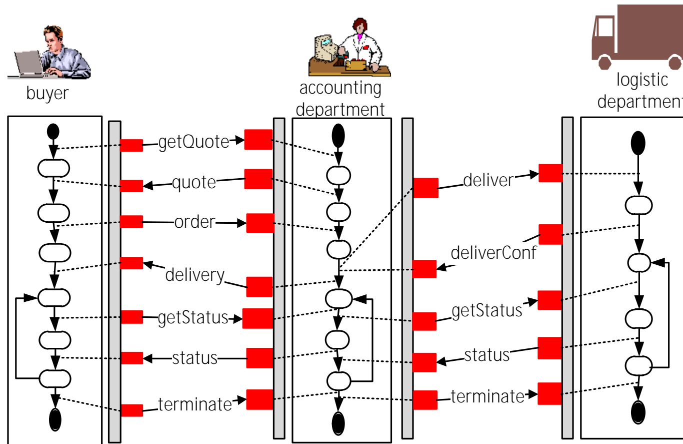



Figure 7.4: Overview of the example scenario

with a "quote" message, the buyer sends the "order" message, which is forwarded to the logistic process by the accounting process. The logistic process then replies with a "deliverConf" message to the accounting process. The message is forwarded to the buyer via a "delivery" message afterwards. Since the buyer is allowed to do parcel tracking arbitrarily often, this step is embedded in a *while* iteration within the accounting process. More precisely, the accounting department may receive a "getStatus" message sent by the buyer, which is then followed by a synchronous invocation of the logistics "getStatusLOP" operation and the reply of the respective status back to the buyer (via a "status" message). Alternatively, the buyer may decide to terminate the accounting and the logistics process at some point by sending a "termination" message to the accounting process, which is forwarded to the logistic process.

### 7.3.1 Collaborative processes interaction failure analysis

All possible failure points are depicted out in Figure 7.5. The process begins with a "getQuote" *receive* activity and a "quote" *reply* activity. If the process crashes after receiving the request message "getQuote" and before sending a response, or the network fails of sending the response message, this should be a pending response failure, which is marked as a $RESP$ between the "getQuote" and "quote" activities. For similar reasons, in the "parcel tracking" *while* iteration

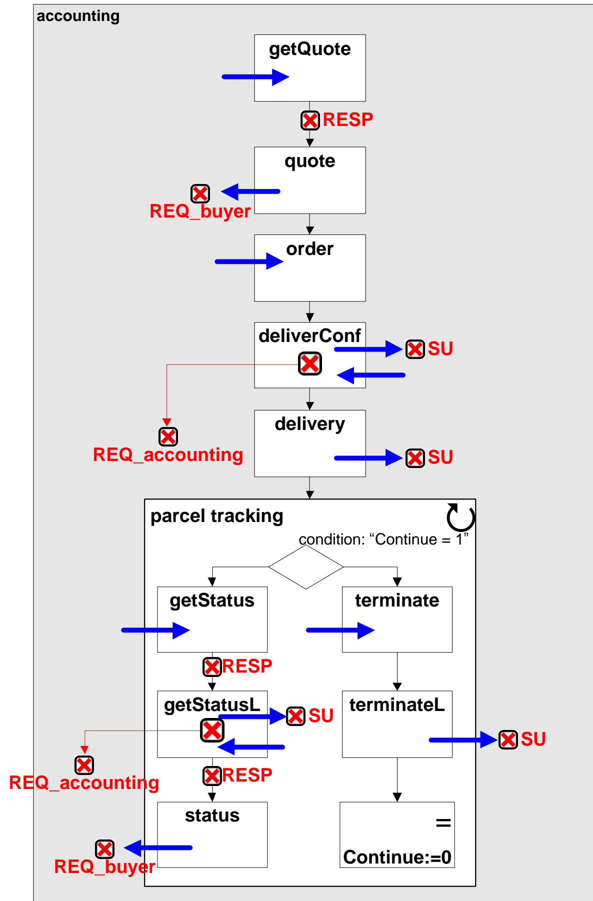



Figure 7.5: Accounting process

tion, if the accounting process crashes after the *receive* activity "getStatus" and before the "status" *reply* activity, a pending response failure will occur. We can notice that there is a nested *invoke* activity in between.

For the "quote" *reply* activity, if the buyer process which should receive this reply fails after sending the request, a pending request failure will happen, which is marked as a $REQ_buyer$. A pending request failure will happen at the point of "status" reply for the same reason. If the accounting process crashes

after the “quote” activity and before receiving the “order” information, the buyer process that sends the “order” information will have a service unavailable failure. However, we don’t need to transform the accounting process for the recovery of this failure. For the same reason, the “parcel tracking” *pick* activity is not marked with service unavailable failure if the process crashes before this activity. For the *invoke* activity “deliverConf”, which invokes the “deliverOP” operation provided by the logistics process, if the logistics process is not available before the invocation, a service unavailable failure will happen to the accounting process, which is marked as a $SU$. If the accounting process crashes after the request message has been sent but the response message has been received, a pending request failure will happen, which is marked as a $REQ_accounting$. For similar reasons, a service unavailable failure will happen to the activities “delivery”, “getStatusL” and “terminateL”, and a pending request failure will happen to the activity “getStatusL”.

### 7.3.2 Accounting process transformation

As is shown in Figure 7.6 and Figure 7.7, in order to make the accounting process recovery enabled, we apply all necessary process transformations. For the *reply* activity “quote” where a pending request failure may occur, we do the transformation to put a conditional branch (*pick* activity) in a *while* iteration. If the buyer process crashes and resends the “getQuote” message, the accounting process will reply with “quote” information in the *while* iteration, until “order” information is received.

For pending response failures (marked as $RESP$) that occur between “getStatus” and “status”, in order to avoid a nested “getStatusL”, we split the operation that the parameters of “getStatus” request sent asynchronously with a query operation to obtain the processing result.

## 7.4 General process design principles

In this section, we are going to discuss some process design guidelines to avoid a few interaction failures.

**(1) Use asynchronous interactions as much as possible**

A synchronous interaction will block the initiator while the responder processes the request. If during this time a system crash happens on either side or if a the network failure occurs, then the interaction will also fail (i.e., it cannot be completed). On the other hand, an asynchronous interaction will not block

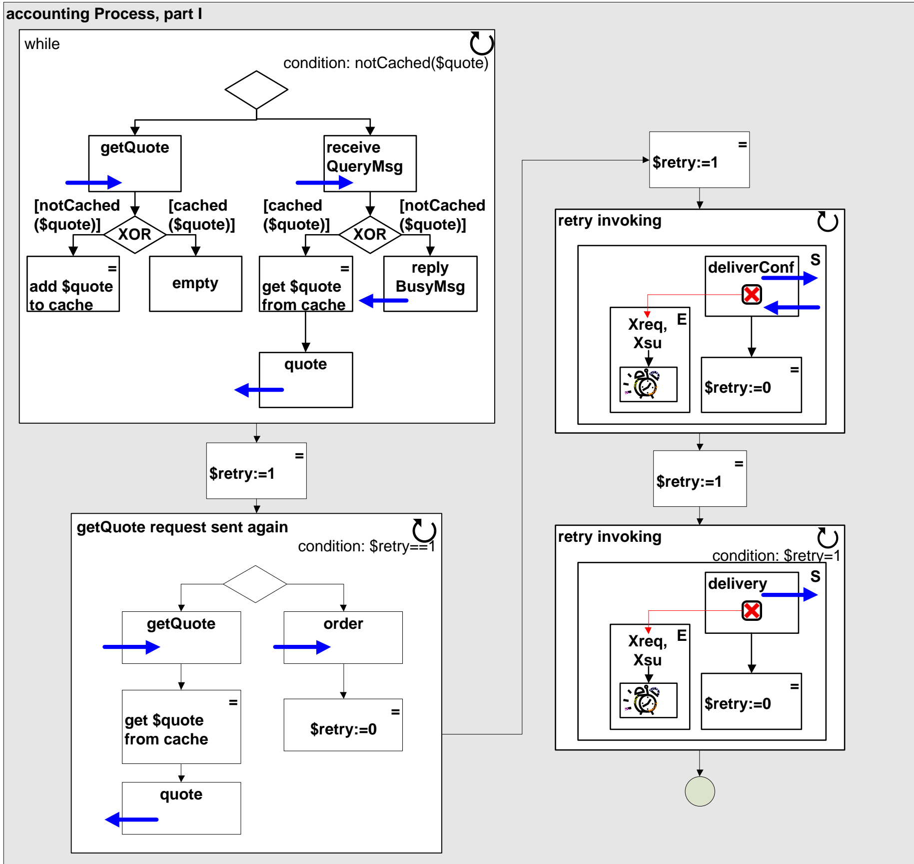



Figure 7.6: Transformed accounting process, part I

the initiator, thus asynchronous interactions can avoid pending request failure and pending response failure. If the system of either party crashes, for asynchronous interactions, the only interaction failure that has to be recovered is service unavailable failure. As an example, in [100], all services that may result in a state change use asynchronous interactions.

**(2) Finalized by asynchronous interaction**

Our basic idea is based on message resending. If the final interaction is synchronous, the responder instance may terminate itself after sending the fi-

Figure 7.7: Transformed accounting process, part II

nal response message. However, if this response message gets lost, the resent request message will not be accepted by the terminated responder instance. If the final interaction is asynchronous, the responder will not terminate itself until the final request message is received.

**(3) Make messages distinguishable from each other**

The responder needs to distinguish a resent message from a next (new) message. If by process design, the same message is sent multiple times, our solution does not work. However, most of the cases, even if copies of the identical messages are sent multiple times, they can be distinguished by including a field with a unique value. For example, a client may submit duplicate orders for the same product, however, these order messages normally have different order ids or different timestamp values.

## 7.5 Conclusions

The composition of the solutions in previous chapters shows solutions for different interaction failures. In this chapter, we have shown how the solutions can work together. We have followed the steps below to derive a robust process to recovery from all the three types of interaction failures:

- apply the solution of pending request failure, then the solution of service unavailable.
- based on the above step, incorporate the solution to the pending response failure to generate a solution that is able to recover from all possible interaction failures.

Although in this chapter we only illustratively show how the solutions are composed, we demonstrated that the multiple solutions can be composed as described in the above steps. If a single interaction failure occurs, the composed solution can recover from the failure using the corresponding part of the transformed process. However, recovery from multiple interaction failures that happen at the same time is a topic for future work.

CHAPTER 8

# Evaluation

In this chapter, we evaluate our solutions presented in sections 4, 5 and 6 in three aspects: their correctness, their performance overhead and the complexity of the process transformation.

## 8.1 Correctness validation

The correctness validation aims to show that the solutions presented in sections 4, 5 and 6 provide robust interactions for collaborative services with regards to system crashes and network failures. The core of the validation is to define robust interaction criteria such that the failed interactions recovery can be automatically evaluated. Process interactions are achieved by exchanging messages. From state coordination point of view, one service changes its state, sends a message to relevant parties, and if the other parties have successfully received the message, then further state changes is possible. Thus, we have to check whether it is possible that the services can apply further state changes before the message exchange has been completed. The indication for an additional state change is exchanging additional (different) messages. Successful message exchanges can be verified by comparing the states of the services involved for all possible executions.

### 8.1.1 Validation procedure

Fig. 8.1 shows the correctness validation in three steps. First, we prove that collaborative business processes always pass the correctness criteria when no failure happens. Second, we prove that the business processes cannot pass the criteria if interaction failure happens Finally we prove that the transformed business processes fulfill correctness criteria when interaction failure happens.

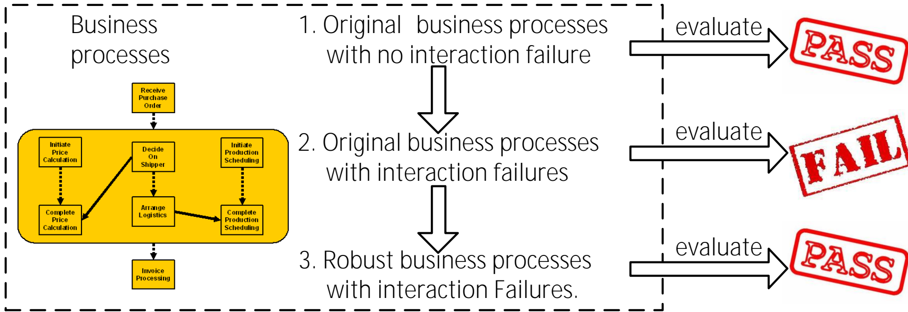



Figure 8.1: Evaluation Procedure

In the last two steps, each of the three interaction failures is checked, one at a time.

In each step of validation, we start from modeling the collaborative business processes as a Petri net. Then the Petri net is transformed into occurrence graph (automata model). We finish the correctness proof by checking that the occurrence graph (automaton model) of the transformed processes is subsumed by the correctness criteria automata. The subsumption checking algorithm [87] is implemented as a program to check correctness.

### 8.1.2 Notion of state

Our solutions have been formalized by Petri nets, which forms a basis for correctness validation. The states of a message sending and receiving are presented by the markings of the Petri Nets and the transitions between them [101]. An *occurrence graph* represents all possible states and state changes derived from a Petri net. Formally, an occurrence graph is an automaton $< Q, \Sigma, \delta, q_0, F >$, where

- $q_0$ is the initial state;
- $Q$ is the set of all states reachable from $q_0$;
- $F$ is the set of final states;
- $\Sigma$ is the set of Petri net transition labels, which represents the sending and receiving of messages, etc.;

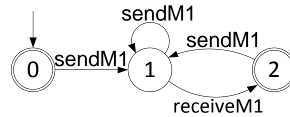



Figure 8.2: Criteria automaton for a single message sending and receiving

- $\delta$ is the automaton transition function: $Q \times \Sigma \rightarrow Q$ representing the occurrences of a Petri net transition from an input state to an output state labeled with the Petri net transition responsible for this state change.

### 8.1.3 Correctness criteria for state synchronization

In the following we present the correctness criteria, in the form of finite state automata, to evaluate the correctness of our proposed solutions. We consider two correctness criteria: the correctness criterion related to sending or receiving a single message in the context of an asynchronous message exchange, and a correctness criterion for synchronous request and response messages.

#### Single message

The exchange of a single message is successful if and only if it synchronizes the state of the sending process and the receiving process. This is the case if the message sent by the sending process is actually received by the receiving process, possibly after resending the message multiple times.

This criterion can be represented as an automaton and is depicted in Figure 8.2: for any message $M1$, we use the Deterministic Finite Automaton (DFA) $< Q, \Sigma, \delta, q_0, F >$ to formalize the criteria. The global message sending and receiving status are modeled as the state set $Q = \{0, 1, 2\}$. The alphabet $\Sigma = \{sendM1, receiveM1\}$. $sendM1$ ($receiveM1$) models the sending (receiving) of message $M1$. $q_0 = 0$ is the initial state and $F = \{0, 2\}$ is the set of final states. The transitions rules $\delta$ are visualized as Figure 8.2.

- A transition $sendM1$ from state $0$ to state $1$ models the sending of message $M1$.

- A transition from state $1$ to itself models that the message $M1$ is sent multiple times, until it is received.

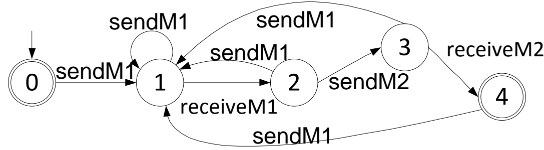



Figure 8.3: Criteria automaton for synchronous request and response messages

- A transition $receiveM1$ from state $1$ to state $2$ represents represents that the message has been received.
- A transition from state $2$ to state $1$ represents that in the original process definition, the message could be sent multiple times, e.g., in a *while* iteration.

After message $M1$ has been sent for the first time, the rest of the interaction has to be completed (i.e., message *M1* has to be received at the receiving end) before any further interaction is allowed. This is reflected in Figure 8.2, such that in state $1$ the only outgoing transitions are either sending message $M1$ again or receiving message $M1$.

### Synchronous request and response messages

The synchronous interaction criteria should take into consideration both request and response messages. Informally, the idea is presented as follows.

1. A request may be sent multiple times until received;
2. A response message must be sent afterwards. No interaction is allowed after the synchronous request message is received, except the sending of the response message.
3. The sequence of 1) and 2) can be repeated multiple times until the response message is received.

The criteria are formalized using the automaton shown in Figure 8.3. *M1* is the request message and *M2* is the response message. In state $2$ the only allowed transitions are to send the response message ($sendM2$) or a resend of the request message ($sendM1$). In case there is an error during the sending or receiving of response message $M2$, the request message $M1$ may have to be

resent to enable the resending of the response message M2. Thus, Figure 8.3 shows a transition labeled as $sendM1$ from state $3$ to state $1$.

The transition from state $4$ to state $1$ represents that the synchronous request message can be sent multiple times, e.g., in a *while* iteration.

### 8.1.4 Correctness validation

The aim of the proof is to show that the correctness criteria defined before are always guaranteed by our proposed approach. First we prove that collaborative business processes always pass the correctness criteria when no failure happens, then we prove that the business process cannot pass the criteria if interaction failure happens. Finally we prove that the transformed business process fulfills the correctness criteria when interaction failure happens.

In each step of the correct validation, we use the Petri nets model of the collaborative processes as the basis. We transform the Petri net model into the occurrence graph (automaton model). Then we extend the criteria automata with messages from the automation model of collaborative processes to make it complete. We finish the correctness proof by checking that the occurrence graph (automaton model) of the transformed processes is subsumed by the correctness criteria automata, which verifies whether the specified criteria are guaranteed by the occurrence graph and consequently by the business processes. The subsumption checking algorithm [87] is implemented as a program to check correctness.

We use the solution of pending request failure presented in section 4.2 as an illustration of the correctness proof procedure. The solutions of pending response failure and service unavailable presented in chapters 5 and 6 follow a similar proof procedure. The first step is to simulate the Petri net model of our solution for the pending request failure in Figure 8.4 to generate an occurrence graph. The resulting occurrence graph (automaton model) has 46 states and 70 transitions. We then simplify the automaton, by replacing transition labels not related to interactions by empty transitions (epsilon transitions) and minimize the automaton. The minimization algorithm [87] is implemented as a program. The resulting automaton is depicted in Figure 8.5. In order to make the transition labels consistent with the correctness criteria automata defined above, we have renamed them. The messages in this occurrence graph are (*M1, M2, M3*). *M1* is the request message, *M2* is the response message and *M3* is the next request message. The correspondence between this occurrence graph and the Petri net (Fig. 8.4) is as follows:

Figure 8.4: Petri net model of pending request failure recovery

- $t_inv1 \rightarrow sendM1$
- $t_rec \rightarrow receiveM1$
- $t_rep \rightarrow sendM2$
- $t_inv2 \rightarrow receiveM2$
- $t_inv_next \rightarrow sendM3$
- $t_rec_m3 \rightarrow receiveM3$

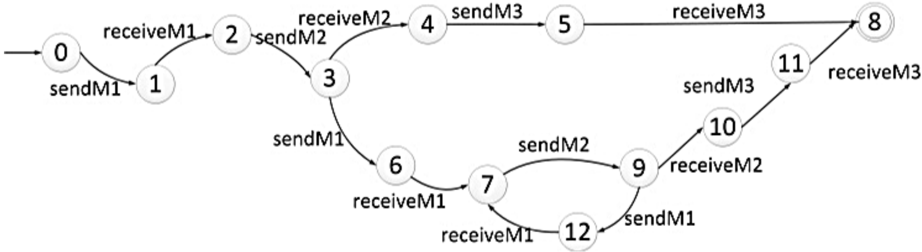



Figure 8.5: Occurrence graph of pending request failure recovery

- $t_rec_m6 \rightarrow receiveM3$

The following properties must hold to prove correctness:

1. For each message $M1$, $M2$ and $M3$ in the occurrence graph, the sequence of sending and receiving of messages meets the correctness criterion for a single message exchange proposed in subsection 8.1.3.

2. The sequences of sending and receiving of synchronous request and response messages $M1$ and $M2$ meet the criterion for exchanging a request and a response message (synchronous interaction) proposed in subsection 8.1.3.

### Single message sending and receiving

In this thesis we only present the proof for message $M1$. The proof for the other two messages $M2$ and $M3$ is similar. The occurrence graph and the correctness automaton is the same except that in the corresponding criteria automaton (Figure 8.2), *M1* is replaced by *M2* and *M3* respectively. The first step is to extend the criterion automaton of Figure 8.2 with messages contained in the occurrence automaton of Figure 8.5. These are additional transition labels $sendM2$, $receiveM2$, $sendM3$, and $receiveM3$. Further, an error state (state 3) is added to the criterion automaton, since the original automaton only contains the correct message sequences. To complete the automaton, the following transitions are added:

- Transitions from state 0 to state 0 labeled $\{sendM2, receiveM2, sendM3, receiveM3\}$, representing that in the initial state, the sending and receiving of other messages does not affect the state of sending and receiving message $M1$.

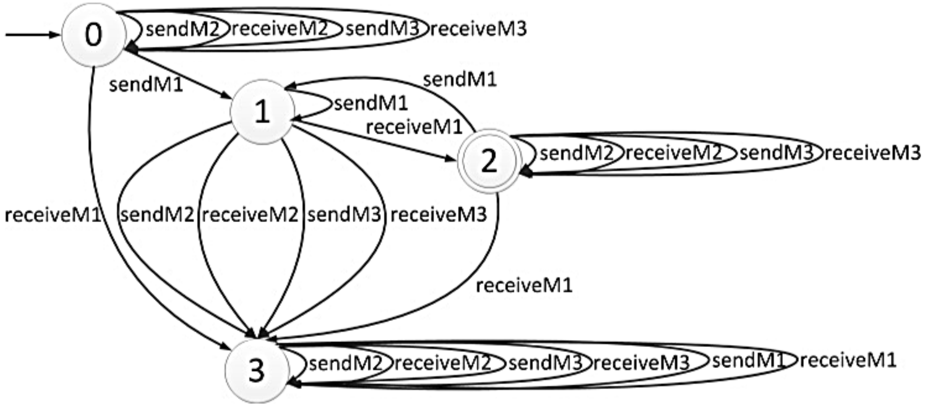



Figure 8.6: The extended criterion automaton

- Transition from state 0 to state 3 labeled $receiveM1$, representing that in the initial state, receiving an message $M1$ that has not been sent before is an error.
- Transitions from state 1 to state 3 labeled $\{sendM2, receiveM2, sendM3, receiveM3\}$, representing that once message $M1$ is sent, any other interaction is an error.
- Transitions from state 2 to itself labeled $\{sendM2, receiveM2, sendM3, receiveM3\}$, representing that once message $M1$ is received, no other interaction affects the state of sending and receiving message $M1$.
- Transition from state 2 to state 3 labeled $receiveM1$, representing that receiving $M1$ multiple times without sending is an error.
- Transitions from state 3 to itself labeled with all messages, representing that an error cannot be compensated.

Figure 8.6 shows the criterion automaton obtained with these extensions.

The next step is to test whether the occurrence automaton of Figure 8.5 is subsumed by the extended criterion automaton. We implement the subsumption algorithm [87] as a program and it turned out that the occurrence graph (automaton model) was indeed subsumed by the extended criteria automaton. Therefore, the criterion is fulfilled for all possible executions contained in the occurrence graph.

### Synchronous message sending and receiving

Since messages $M1$ and $M2$ are the request and response of a synchronous message exchange, the correctness criteria for synchronous messages proposed in subsection 8.1.3 must also hold. As in the previous case, the criterion automaton must be extended by the messages contained in the occurrence automaton, that is, $sendM3$ and $receiveM3$. The extension consists of an error state (state 5) and the following transitions:

- Transitions from state 0 to state 5 labeled $\{receiveM1, sendM2, receiveM2\}$, representing that in the initial state, the receiving of the request message and the sending and receiving of the response message are errors.
- Transitions from state 0 to itself labeled $\{sendM3, receiveM3\}$, representing that in the initial state, the sending and receiving of message M3 does not affect the state of the synchronous request and response.
- Transitions from state 1 to state 5 labeled $\{sendM2, receiveM2, sendM3, receiveM3\}$, representing that in state 1, the synchronous message is sent once or multiple times but has not been received yet. According to the principle proposed in subsection 8.1.3, the synchronous request message has to be accepted before any other message exchange, thus any other interaction is an error.
- Transitions from state 2 to itself labeled $\{sendM3, receiveM3\}$, representing that under the condition that the synchronous request message has been received, the sending and receiving of the message $M3$ does not affect the sending and receiving of the synchronous message.
- Transitions from state 2 to state 5 labeled $\{sendM1, receiveM1, receiveM2\}$, representing that under the condition that the synchronous message $M1$ has been accepted, send or receive $M1$ again is an error. Meanwhile, under the condition that the synchronous response message has not been sent, receiving the response message is an error.
- Transitions from state 3 to state 5 labeled $\{receiveM1, sendM2, sendM3, receiveM3\}$, representing that after the synchronous response message has been sent but before it has been received, the initiator process may have received the response message ($receiveM2$). A possible process crash or network error will cause the loss of the response message and then the request message will be resent ($sendM1$). All other transitions are erroneous in this case.

- Transitions from state $4$ to itself labeled $\{sendM3, receiveM3\}$, representing that in the final state, no further interactions (sending and receiving of message $M3$) affect the synchronous interaction.
- Transitions from state $4$ to state $5$ labeled $\{receiveM1, sendM2, reciveM2\}$, representing that as the synchronous interaction is finished, receiving the request message, sending the response message or receiving the response message is an error.
- Transitions from state 5 to itself labeled $\{sendM1$, $receiveM1$, $sendM2$, $receiveM2$, $sendM3$, $reciveM3\}$.

The correctness criterion is fulfilled if the extended criteria automaton subsumes the occurrence automaton. We implement the subsumption algorithm [87] as a program and it turned out that the occurrence graph (automaton model) was indeed subsumed by the extended criteria automaton. Therefore, the criterion is fulfilled for all possible executions contained in the occurrence graph. This proves that the sequences of sending and receiving of synchronous request and response messages $M1$ and $M2$ are correct.

## 8.2 Performance evaluation

Below we investigate the performance overhead of our solutions. In case the infrastructure (software, hardware and network configuration) is the same, performance depends on the process design and the workload.

We evaluated the performance overhead of our solutions with different workloads. The requests sent per minute by the simulation client comply to a Poisson distribution [102]. We collect performance under two workloads, namely two mean message arrival rates $\lambda = 5$ and $\lambda = 10$ (messages per second). We use these workloads because according to our tests under the available hardware and software configurations, higher workload exhausts the server resources. Each test run lasted for 60 minutes, but only the response times during the 30 minutes in the middle of this period have been considered (steady state).

We implemented the original process and the transformed processes with our solutions for pending request failure, pending response failure and service unavailable failure. We assume that the interaction pattern is *send/receive* (synchronous interaction). As a further interaction is required for the pending request failure, the *send/receive* synchronous interaction is followed by an

Table 8.1: Performance overhead analysis

| **Pending request failure, *1 : 1* shared state type** | | | | |
|---|---|---|---|---|
| Determinate further interaction | | | | |
| Workload (messages per second) | Before | After | Overhead | % |
| $\lambda = 5$ | 432 ms | 475 ms | 43 ms | 9% |
| $\lambda = 10$ | 461 ms | 480 ms | 19 ms | 4% |
| Indeterminate further interaction | | | | |
| $\lambda = 5$ | 287 ms | 379 ms | 92 ms | 24% |
| $\lambda = 10$ | 322 ms | 452 ms | 130 ms | 28% |
| **Pending request failure, *n : 1* shared state type** | | | | |
| Workload (messages per second) | Before | After | Overhead | |
| $\lambda = 5$ | 313 ms | 375 ms | 62 ms | 17% |
| $\lambda = 10$ | 256 ms | 440 ms | 184 ms | 42% |
| **Pending response failure, *1 : 1* shared state type** | | | | |
| Workload (messages per second) | Before | After | Overhead | |
| $\lambda = 5$ | 432 ms | 680 ms | 248 ms | 36% |
| $\lambda = 10$ | 461 ms | 681 ms | 220 ms | 32% |
| **Pending response failure, *n : 1* shared state type** | | | | |
| Workload (messages per second) | Before | After | Overhead | |
| $\lambda = 5$ | 645 ms | 1607 ms | 962 ms | 60% |
| $\lambda = 10$ | 892 ms | 5419 ms | 4527 ms | 84% |
| **Service unavailable failure** | | | | |
| Workload (messages per second) | Before | After | Overhead | |
| $\lambda = 5$ | 432 ms | 508 ms | 76 ms | 15% |
| $\lambda = 10$ | 461 ms | 507 ms | 46 ms | 9% |

asynchronous interaction, determinate or in determinate. We evaluated the performance overhead by comparing the response time of the original process and the transformed ones.

Table 8.1 shows the performance results. In most of the cases, the performance overhead is small (within 100 ms), e.g., for the pending request failure and shared state type *1 : 1*, the overhead at the workload $\lambda = 5$ is 43 ms and at the workload $\lambda = 10$ is 19 ms. For the service unavailable failure, the performance overhead is 76 ms under the workload of $\lambda = 5$ and 46 ms under the workload of $\lambda = 10$. However, the performance overhead for the pending response failure is big (more than 200 ms). For the pending response failure and

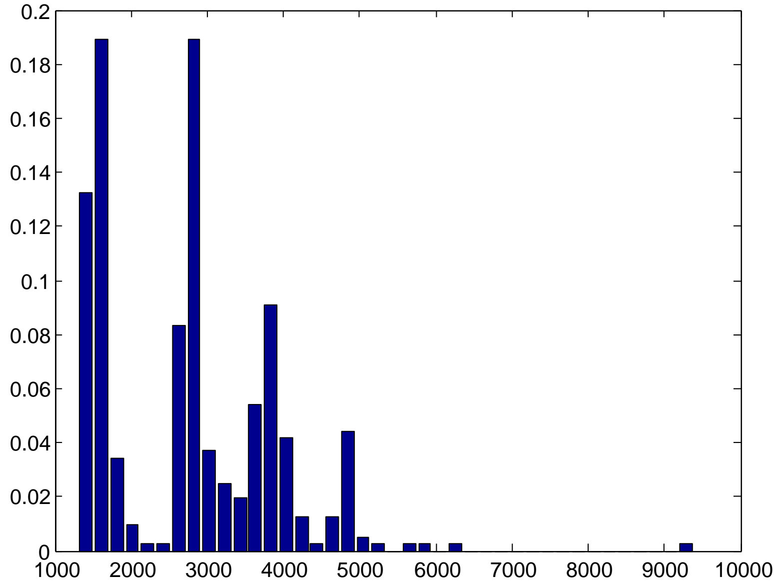



Figure 8.7: Response time distribution (the percentage of response times in unit of millisecond)

shared state type *1 : 1*, the performance overhead is 248 ms under the workload of $\lambda = 5$ and 220 ms under the workload of $\lambda = 10$. The big overhead of the pending response failure is due to splitting of one synchronous interaction into two synchronous interactions and introducing an adapter service.

For the pending response failure and shared state type *n : 1*, the performance overhead is 962 ms under the workload of $\lambda = 5$ and 4527 ms under the workload of $\lambda = 10$. In this solution, some performance overhead is caused by the process specific query intervals in the adapter process. After the responder finishes processing, it can only send the reply after the adapter has queried the result. For example, in our test, the query interval is 1000 ms. In the response time distribution, shown in Figure 8.7, we see that the response time peak interval is near 1000 ms, which is the query interval. Another reason could be the management of the adapter instances. At runtime, each incoming request message triggers a new adapter instance creation and we use the limited EC2 instance type in this evaluation (*t1.micro* with 1 vCpu and 0.613 GiB memory). However, we expect lower performance overhead when the infrastructure is

scalable, like in a cloud environment [103].

## 8.3 Business process complexity evaluation

Our process transformation increases the complexity of the robust business processes. In this section, we investigate the process complexity with respect to the number of activities (in terms of WS-BPEL).

We consider below a process design and our proposed transformation to illustrate the introduced measures. For a responder process, a synchronous interaction followed by a one-way message can be defined as the following WS-BPEL snippet:

```
<receive name="receive1" ... />
  <!-- nested activities (some processing) -->
<reply name="reply1" ... />

<receive name="receive2" ... />
```

By contrast, the transformed responder process with capabilities of pending request failure recovery, can be defined as in the following WS-BPEL process snippet:

```
<receive name="receive1" .../>
  <!-- nested activities (some processing) -->
<reply name="reply1" .../>
<while>
  <condition>$PR=1</condition> <!-- PR is initialized to 1 -->
    <pick>
      <onMessage name="receive1" ... >
        <reply name="reply1" .../>
      </onMessage>
      <onMessage name="receive2" ... >
        <assign ... /> <!-- assign 0 to variable PR -->
      </onMessage>
    </pick>
</while>
```

The original WS-BPEL process definition is composed by message sending and receiving activities while the transformed process contains assignments and structured activities: a *while* iteration activity and a pick activity with two branches. This design corresponds replacing 3 activities by 6 activities.

Table 8.2: Number of activities before and after transformation

| **Pending request failure, state type: private** | |
|---|---|
| Before transformation | After transformation |
| Determinate further interaction | |
| 3 (responder side) | 6 (responder side) |
| Indeterminate further interaction | |
| 3 (responder side) | $2n + 8$ [1] |
| **Pending request failure, state type: shared, static** | |
| 3 (responder side) | 7 (responder side) |
| **Pending response failure, state type: private** | |
| 3 (responder side) | 5 (responder side) |
| **Pending response failure, state type: shared, static** | |
| 3 (responder side) | 10 (responder side) |
| **Service unavailable failure** | |
| 1 (initiator side) | 7 (initiator side) |

Table 8.2 summarizes the number of activities before and after the process transformation for the initiator or responder of an interaction per failure type. Table 8.2 shows that the proposed solutions have increased considerably the complexity of the process design. However, since the transformation has been formally defined and an automatic process transformation tool can be designed based on this, we believe that the increased complexity should not be a big burden for process designers.

## 8.4 Fulfilment of requirements

In Section 1.2, we proposed the requirements of our solution, which are discussed as follows:

- Requirement R0: The solution should function correct. We have proposed the correctness criteria and proved the correctness of our solution in Section 8.1.
- Requirement R1: The process transformation should be transparent for process designers. Currently, the robust process is transformed from the original process manually. However, we have presented a solution that an automatic process transformation could implement.

- Requirement R2: The transformed process should not require additional investments in a robust infrastructure. As presented in Chapters 4, 5 and 6, our solutions do not put additional requirement on the robust infrastructure investment.

- Requirement R3: As a solution at process language level, the process interaction protocols should not be changed. The interaction protocol of the responder process has been changed in the solution of the pending response failure, however, we have presented an adapter process to make this change transparent to the initiator process.

- Requirement R4: The service autonomy should be preserved. This is relevant with the previous requirement. As we do not change the interaction protocol, if one party transforms the process according to our approach and the other party does not, they can still interact with each other, although without being able to recover from system crashes and network failures.

- Requirement R5: Only existing process language specifications could be used. In our solutions, we use WS-BPEL as our implementation language without extending the language, so that the robust WS-BPEL process is independent of any specific engine.

- Requirement R6: The solution should have acceptable performance. The performance of our solution is evaluated in Section 8.2.

## 8.5 Sensitivity of our design

In this section, we discuss the sensitivity [104] of our design. The context of our research work is the following:

- Collaborative services: described by explicit workflow. Our solutions assume the business processes implemented using workflow language like WS-BPEL. We have illustrated our solutions using WS-BPEL, however, the solutions can be applied to other process languages as they support similar workflow patterns [105]. Our solution cannot be applied to services which do not have an explicit workflow. This is a future research work as discussed in subsection 9.4.2.

- Execution environment: standard process engines. We have deployed our solutions to the two process engines: Oracle Business Process Manager and Apache ODE. We expect minor engineering effort of migrating the robust process between different standard process engines.
- Network environment: we assume the TCP/IP environment where services are interacting using HTTP messages. The failure behavior of other network environment will be investigated in future.

## 8.6 Conclusions

In this chapter, we have evaluated the correctness of our solutions against the proposed criteria, the performance overhead of our solutions and the complexity of the robust business process.

The core of the correctness validation is to define robust interaction criteria such that our solution can be automatically evaluated. The robust interactions are achieved by successfully exchanging messages. One service changes its state, sends a message to other relevant partners, and if the other partners has successfully received the message, then further state changes is possible. Thus, we check whether it is possible that the services can apply further state changes before the message exchange has been completed. The indication for an additional state change is exchanging additional (different) messages. Successful message exchanges can be verified by comparing the states of the services involved for all possible executions.

We have shown the considerable performance overhead under limited infrastructures (Amazon EC2 (*t1.micro* with 1 vCpu and 0.613 GiB memory)). However, we expect higher performance under a elastic infrastructure, for example, cloud environment. Our solutions have increased considerably the complexity of the process design. However, since the transformation has been formally defined and an automatic process transformation tool can be designed based on this, we believe that the increased complexity should be a big burden for process designers.

CHAPTER 9

# Conclusions and future work

This chapter presents the conclusions of this thesis and identifies some areas for further research. This chapter is structured as follows: Section 9.1 gives some general conclusions of our research. In Section 9.2, we revisit the research questions, introduced in Chapter 1 and discuss how they have been answered. Our research contributions are presented in Section 9.3 and Section 9.4 identifies areas for further research.

## 9.1 General conclusions

System crashes and network failures are very common events, which may happen in various information systems of these collaborative organizations, e.g., servers, desktops or mobile devices. System crashes and network failures may result in inconsistent states/behaviors of the business processes involved in a collaboration and possibly to a deadlock of these business processes. In this thesis we have presented a solution to recover business processes from system crashes and network failures. We transform the original processes into robust counterparts by incorporating necessary recovery activities. We have investigated the business process behaviors in case interaction failures occur caused by system crashes and network failures. We developed solutions (robust counterparts of the original processes) to recover from these failures. We proposed mechanism to prove the correctness of our solutions and applied the mechanism to our solution to show the correctness of our solutions. The performance and robust process complexity aspect of our solutions are evaluated as well.

## 9.2 Research questions revisited

In this section, we revisit the research questions introduced in chapter 1. We present the research questions and summarize the answers that follow from our research.

- **Main research question: How to recover collaborative processes interaction failures caused by system crashes and network failures?**
  This is a knowledge question that can be decomposed into several subquestions and each question is answered in this thesis.

- **Research question 1: What are the current existing solutions which can be used to recover from interaction failures?**

  A business process execution environment is often built up with multiple abstraction layers, namely application layer, infrastructure layer and integration layer. Interaction failure solutions can be found at each of the layers. Application layer solutions make use of the application programming languages support, such as exception handling features and transactional features. However, these solutions require that the programmer is aware of all possible failures and their recovery strategies. Solutions at infrastructure layer are transparent to application programmers. However, normally these solutions require more infrastructure investment, e.g., more reliable communication channels. We assume system crashes and network failures are rare events that make additional infrastructure support expensive. Furthermore, these solutions may make the implementation specific to process engine, which make the business process difficult to migrate between different process engines. We can conclude there is a need for a solution that is transparent to process designers and require little infrastructure investment.

- **Research question 2: What are the necessary concepts/models in our solution?**

  We have learned that at runtime, one process may have multiple instances while each instance maintains its own state information. The state information can be shared among multiple process instances via interactions realized by message exchanges. This results in the identification of four shared state types, based on the number of initiator instances and the number of responder instances that are involved in a collaboration. The four shared state types are: $1:1$, $1:n$, $n:1$ and $m:n$. We have proposed

Petri net model and Nested Word Automata (NWA) model to provide a formal basis upon which the solution and correctness validation can be based. We have used Web Services Business Process Execution Language (WS-BPEL) as an illustrative language to illustrate our solutions.

- **Research question 3: What are the corresponding behavior and recovery approach for the interaction failures?**

  For this question, we have learned that possible interaction failures are *pending request failure*, *pending response failure* and *service unavailable*. Pending request failure is an initiator system crash after sending the request message. Pending response failure is a responder system crash after receiving the request message or the network fails to deliver the response message. Service unavailable is that the network failed in the request message delivery or responder crashes before receiving the request message. The recovery of pending request failure depends on the shared state types. For $1 : 1$ type of state information, the recovery method is that the initiator resends the request message while the responder uses the previous result as a response without reprocessing the duplicate request message. The $n : 1$ state information is shared between multiple initiator instances and one responder instance. The difficulty is that if one initiator system crashes, the interaction between other running initiator instance and the responder instance may further change the responder state and overwrite the previous interaction result, which make using the previous result as a reply impossible. The recovery method consists of caching the response message when the responder system state changes, and using the cached message as a response for a resent message due to failure. If the state information is $1 : n$, the solution of state type $1 : 1$ can be applied. If the state information is $m : n$, the solution of state type $n : 1$ can be applied, as discussed in Chapter 4. The recovery of pending response failure depends on the four state types. For state type $1 : 1$, to avoid the crash in the middle of a processing nested between receiving a request and replying a response, our recovery is to split one interaction into two. One sends the request parameters and the other asks for the result. For state type $n : 1$, the problem is that if we split one synchronous interaction into two synchronous interactions. The request message for the second interaction from multiple initiators will accumulate at the responder side, thus increasing the possibility of message queue overflow and causing potentially performance problems. Our solution is to parallelize the processing of the request message and the initiator query for the process-

ing result. The transformation adds a caching capability, i.e., the response message for a newly incoming message representing a non-idempotent operation is cached. If the responder receives a resent message from the initiator due to a failure, the responder replies the cached response message and does not execute the operation again. If the state information is $1 : n$, the solution of state type $1 : 1$ can be applied. If the state information is $m : n$, the solution of state type $n : 1$ can be applied, as discussed in Chapter 5. The recovery of service unavailable is based on message resending. Whenever an initiator sends a message, the message sending activity will be repeated until the target is available. We have shown that our solutions provide robust interactions for collaborative services with regards to system crashes and network failures. The core of the validation is to define robust interaction criteria such that our solution can be automatically evaluated.

- **Research question 4: How to combine the recovery solutions for different approach?** We have shown how the solutions are working together. We have followed the following steps to derive a robust process to recovery from all the three types of interaction failures: First, we apply the solution of pending request failure, then the solution of service unavailable is applied as well. Based on the above step, we incorporate the solution of pending response failure to generate a solution that is able to recover from all possible interaction failures. If a single interaction failure occurs, the composed solution can recover from the failure using the corresponding part of the transformed process. However, it is our future work to recover from multiple interaction failures that happen at the same time.

After the sub-questions have been answered, we can answer the main research question. We have proposed solutions for collaborative services interaction failures caused by system crashes and network failures, which is based on message resending and using cached response message as a reply. We have validated the correctness of our process transformations and we implemented a prototype to test the runtime performance the complexity of the robust business process.

## 9.3 Research contributions

Based on the answers to the research questions and the existing solutions, we formulate the following research contributions:

- **recovery solutions transparent to application developers** The robust business process can be derived from the original process with transformations, without the involvement of the application developers. In contrast to the solutions based on language capabilities, e.g., exception handling and transactions, our solution makes the interaction recovery transparent to application developers. The solutions based on exceptions require that the application developers are aware of possible failures and their recovery strategies. Transaction based solutions require that the application developers are aware of the transactions where the ACID properties apply. However, we aim at relieving the application developer from concerns that have to do with interaction failures recovery.

- **reduce reliable infrastructure investment** Our solutions are at process level that are independent from the infrastructure layer. This makes our solution easy to migrate between different process engines. Additionally, we assume that system crashes and network failures are rare events that make the investment in reliable infrastructures expensive. Our solution at process level does not require infrastructure support.

- **stable business partners** Our solution assumes the same business partners during runtime and does not require dynamically changing business partners at runtime. This contracts with approaches that attempt to dynamically change the business partner whenever an interaction failure prevents to continue the collaboration with the current business partner.

## 9.4 Future work

This section discusses some subjects for further research.

### 9.4.1 Automatic process transformation

In our solution, if a business process is given, we transform it into a robust counterpart, which is able to recover from interaction failures. However, most of the process transformation work is done manually. As a next step, we expect

Table 9.1: Failure scheme

| | **Type of failure** | **Description** |
|---|---|---|
| Inside Scope | Crash failure | A server halts, but is working correctly until it halts. |
| | Omission failure | A server fails to respond to incoming requests. |
| | Receive omission | A server fails to receive incoming messages. |
| | Send omission | A server fails to send messages. |
| Outside Scope | Timing failure | A server response lies outside the specified time interval. |
| | Response failure | A server response is incorrect. |
| | Value failure | The value of the response is wrong. |
| | State transition failure | The server deviates from the correct flow of control. |
| | Arbitrary failure | A server may produce arbitrary responses at arbitrary times. |

to implement an automatic process transformation module that incorporates our recovery solution can be incorporated.

### 9.4.2 General software system interaction failures

This thesis investigates interaction failures of collaborative services, which are described as automated business processes. We illustrate our solution by using an standardized process implementation language, WS-BPEL. In future, other software systems and languages can be explored, e.g., mobile based applications [106] where network interruptions happen rather frequently. Furthermore, robust RESTful services with regard to system crashes and network failures is also an area worth exploring. We have proposed robust interaction solutions, the technological limitation is how to apply our solutions to other software systems and languages.

### 9.4.3 Other types of failures

In this thesis the interaction failures caused by system crashes and network failures are explored. Although the considered failure types are the most common

failures that we experience everyday, there are more types of failures. Table 9.1 shows a failure classification scheme [7]. Crash failure is referred as *system crashes* in this thesis. Omission failure and timing failure occur when the network fails to deliver messages (within a specified time interval) and are referred as *network failures* in this thesis. However, response failures due to flaws in the process design, e.g., incompatible data formats, and arbitrary failure, also referred to as Byzantine failure, which is more of a security issue, require further exploration.

The interaction failures caused by incorrect design of process interaction protocols is also worth further exploration, e.g., message duplication or message sequence errors or even deadlocks cause by incorrect process design.

# Bibliography


[1] K. Vollmer, M. Gilpin, and S. Rose, "The forrester wave™: Enterprise service bus, q2 2011," *Forrester Research Inc.*, Apr. 2011.

[2] Amazon's Press Releases, "For the eighth consecutive year, amazon ranks #1 in customer satisfaction during the holiday shopping season," http://phx.corporate-ir.net/phoenix.zhtml?c=176060&p=irol-newsArticle&ID=1769785&highlight=, 2012.

[3] V. Palladino, "Amazon sold 426 items per second in run-up to christmas," http://www.theverge.com/2013/12/26/5245008/amazon-sees-prime-spike-in-2013-holiday-season, Dec. 2013.

[4] Microsoft case study, "Movie theatre chain projects $17 million revenue gain from information integration," http://www.microsoft.com/casestudies/Microsoft-Biztalk-Server-Enterprise-2010/AMC-Entertainment/Movie-Theatre-Chain-Projects-17-Million-Revenue-Gain-from-Information-Integration/710000001643, Nov. 2012.

[5] B. Schroeder and G. Gibson, "A large-scale study of failures in high-performance computing systems," *Dependable and Secure Computing, IEEE Transactions on*, vol. 7, no. 4, pp. 337–350, Oct 2010.

[6] Wikipedia, "Blue screen of death," http://en.wikipedia.org/wiki/Blue_Screen_of_Death.

[7] A. S. Tanenbaum and M. van Steen, *Distributed Systems: Principles and Paradigms*, 2nd ed. Upper Saddle River, NJ, USA: Prentice Hall, 2006, ch. 8, pp. 321–375.

[8] "ODE, the Orchestration Director Engine," http://ode.apache.org/index.html, Apache Software Foundation.

[9] L. Gasser, "The integration of computing and routine work," *ACM Trans. Inf. Syst.*, vol. 4, no. 3, pp. 205–225, 1986. [Online]. Available: http://doi.acm.org/10.1145/214427.214429

[10] D. M. Strong and S. M. Miller, "Exceptions and exception handling in computerized information processes," *ACM Trans. Inf. Syst.*, vol. 13, no. 2, pp. 206–233, 1995. [Online]. Available: http://doi.acm.org/10.1145/201040.201049

[11] OASIS Web Services Business Process Execution Language (WSBPEL) TC, *Web Services Business Process Execution Language*, 2nd ed., OASIS, http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html, Apr. 2007.

[12] A. Barros, M. Dumas, and A. Hofstede, "Service interaction patterns," in *Business Process Management*, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2005, vol. 3649, pp. 302–318.

[13] A. R. Hevner, S. T. March, J. Park, and S. Ram, "Design science in information systems research," *MIS Quarterly*, vol. 28, no. 1, pp. 75–105, Mar. 2004.

[14] R. Wieringa, "Design science as nested problem solving," in *Proceedings of the 4th International Conference on Design Science Research in Information Systems and Technology*, ser. DESRIST '09. New York, NY, USA: ACM, 2009, pp. 8:1–8:12.

[15] "Oracle SOA Suite," http://www.oracle.com/technetwork/middleware/soasuite/overview/index.html, Oracle Corporation.

[16] Apache ODE, "Create a process," https://ode.apache.org/creating-a-process.html#in-memory-execution.

[17] SOA Technology for beginners and learners, "Transient vs. durable bpel processes," http://ofmxperts.blogspot.nl/2012/11/transient-vs-durable-bpel-processes.html, Nov. 2012.

[18] Oracle, "Oracle bpel process manager quick start guide," https://docs.oracle.com/cd/E12483_01/integrate.1013/b28983/intro.htm.

[19] Apache Software Foundation, "Ode architectural overview," http://ode.apache.org/developerguide/architectural-overview.html.

[20] J. B. Goodenough, "Structured exception handling," in *Proceedings of the 2Nd ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, ser. POPL. New York, NY, USA: ACM, 1975, pp. 204–224. [Online]. Available: http://doi.acm.org/10.1145/512976.512997

[21] ——, "Exception handling: Issues and a proposed notation," *Commun. ACM*, vol. 18, no. 12, pp. 683–696, Dec. 1975. [Online]. Available: http://doi.acm.org/10.1145/361227.361230

[22] A. Borgida, "Language features for flexible handling of exceptions in information systems," *ACM Trans. Database Syst.*, vol. 10, no. 4, pp. 565–603, 1985. [Online]. Available: http://doi.acm.org/10.1145/4879.4995

[23] B. S. Lerner, S. Christov, L. J. Osterweil, R. Bendraou, U. Kannengiesser, and A. E. Wise, "Exception handling patterns for process modeling," *IEEE Transactions on Software Engineering*, vol. 36, no. 2, pp. 162–183, 2010.

[24] N. Russell, W. Aalst, and A. Hofstede, "Workflow exception patterns," in *Advanced Information Systems Engineering*, ser. Lecture Notes in Computer Science, E. Dubois and K. Pohl, Eds. Springer Berlin Heidelberg, 2006, vol. 4001, pp. 288–302. [Online]. Available: http://dx.doi.org/10.1007/11767138_20

[25] D. K. Chiu, Q. Li, and K. Karlapalem, "A meta modeling approach to workflow management systems supporting exception handling," *Information Systems*, vol. 24, no. 2, pp. 159 – 184, 1999, meta-Modelling and Methodology Engineering. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0306437999000101

[26] B. Randell, P. Lee, and P. C. Treleaven, "Reliability issues in computing system design," *ACM Comput. Surv.*, vol. 10, no. 2, pp. 123–165, 1978. [Online]. Available: http://doi.acm.org/10.1145/356725.356729

[27] S. Modafferi and E. Conforti, "Methods for enabling recovery actions in ws-bpel," in *On the Move to Meaningful Internet Systems: CoopIS, DOA, GADA, and ODBASE*, ser. Lecture Notes in Computer Science, R. Meersman and Z. Tari, Eds. Springer Berlin Heidelberg, 2006, vol. 4275, pp. 219–236, http://dx.doi.org/10.1007/11914853_14.

[28] M. P. Herlihy and J. M. Wing, "Avalon: Language support for reliable distributed systems," in *Proceedings from the Second Workshop on Large-Grained Parallelism*, 1987.

[29] D. Detlefs, M. Herlihy, and J. Wing, "Inheritance of synchronization and recovery properties in Avalon/C++," *Computer*, vol. 21, no. 12, pp. 57–69, Dec 1988.

[30] S. K. Shrivastava, G. N. Dixon, and G. D. Parrington, "An overview of the arjuna distributed programming system," *IEEE Software*, vol. 8, no. 1, pp. 66–73, 1991.

[31] G. D. Parrington, "The evolution of c++," J. Waldo, Ed. Cambridge, MA, USA: MIT Press, 1993, ch. Reliable Distributed Programming in C++: The Arjuna Approach, pp. 235–248. [Online]. Available: http://dl.acm.org/citation.cfm?id=168501.168523

[32] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz, "Aries: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging," *ACM Trans. Database Syst.*, vol. 17, no. 1, pp. 94–162, Mar. 1992. [Online]. Available: http://doi.acm.org/10.1145/128765.128770

[33] R. A. Lorie, "Physical integrity in a large segmented database," *ACM Trans. Database Syst.*, vol. 2, no. 1, pp. 91–104, Mar. 1977. [Online]. Available: http://doi.acm.org/10.1145/320521.320540

[34] R. Agrawal and D. J. Dewitt, "Integrated concurrency control and recovery mechanisms: Design and performance evaluation," *ACM Trans. Database Syst.*, vol. 10, no. 4, pp. 529–564, Dec. 1985. [Online]. Available: http://doi.acm.org/10.1145/4879.4958

[35] J. N. Gray, R. A. Lorie, G. R. Putzolu, and I. L. Traiger, "Readings in database systems (3rd ed.)," M. Stonebraker and J. M. Hellerstein, Eds. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1998, ch. Granularity of Locks and

Degrees of Consistency in a Shared Data Base, pp. 175–193. [Online]. Available: http://dl.acm.org/citation.cfm?id=302090.302122

[36] S. Modafferi, E. Mussi, and B. Pernici, "Sh-bpel: a self-healing plug-in for ws-bpel engines," in *Proceedings of the 1st workshop on Middleware for Service Oriented Computing*. NY, USA: ACM, 2006, pp. 48–53, http://doi.acm.org/10.1145/1169091.1169099.

[37] A. Charfi, T. Dinkelaker, and M. Mezini, "A plug-in architecture for self-adaptive web service compositions," in *IEEE International Conference on Web Services*, Jul. 2009, pp. 35–42.

[38] M. Fredj, N. Georgantas, V. Issarny, and A. Zarras, "Dynamic service substitution in service-oriented architectures," in *IEEE Congress on Services - Part I*, Jul. 2008, pp. 101–104.

[39] L. Cavallaro, E. Nitto, and M. Pradella, "An automatic approach to enable replacement of conversational services," in *Service-Oriented Computing*, ser. Lecture Notes in Computer Science, L. Baresi, C.-H. Chi, and J. Suzuki, Eds. Springe Berlin Heidelberg, 2009, vol. 5900, pp. 159–174.

[40] O. Moser, F. Rosenberg, and S. Dustdar, "Non-intrusive monitoring and service adaptation for ws-bpel," in *Proceedings of the 17th international conference on World Wide Web*. NY, USA: ACM, 2008, pp. 815–824.

[41] F. Moo-Mena, J. Garcilazo-Ortiz, L. Basto-Diaz, F. Curi-Quintal, S. Medina-Peralta, and F. Alonzo-Canul, "A diagnosis module based on statistic and qos techniques for self-healing architectures supporting ws based applications," in *International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery*, Oct. 2009, pp. 163 –169.

[42] F. Moo-Mena, J. Garcilazo-Ortiz, L. Basto-Diaz, F. Curi-Quintal, and F. Alonzo-Canul, "Defining a self-healing qos-based infrastructure for web services applications," in *11th IEEE International Conference on Computational Science and Engineering Workshops*, Jul. 2008, pp. 215 –220.

[43] S. Todd, F. Parr, and M. Conner, *A Primer for HTTPR*, http://www.ibm.com/developerworks/webservices/library/ws-phtt/, IBM, Mar. 2005.

[44] OASIS Web Services Reliable Exchange (WS-RX) TC, *Web Services Reliable Messaging (WS-ReliableMessaging)*, http://docs.oasis-open.org/ws-rx/wsrm/200702/wsrm-1.2-spec-os.html, OASIS Standard, Rev. 1.2, Feb. 2009.

[45] J. Gray, "The transaction concept: Virtues and limitations (invited paper)," in *Proceedings of the Seventh International Conference on Very Large Data Bases - Volume 7*, ser. VLDB '81. VLDB Endowment, 1981, pp. 144–154. [Online]. Available: http://dl.acm.org/citation.cfm?id=1286831.1286846

[46] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger, "The notions of consistency and predicate locks in a database system," *Commun. ACM*, vol. 19, no. 11, pp. 624–633, Nov. 1976. [Online]. Available: http://doi.acm.org/10.1145/360363.360369

[47] J. Gray, "Notes on data base operating systems," in *Operating Systems*, ser. Lecture Notes in Computer Science, R. Bayer, R. Graham, and G. Seegmüller, Eds. Springer Berlin Heidelberg, 1978, vol. 60, pp. 393–481. [Online]. Available: http://dx.doi.org/10.1007/3-540-08755-9_9

[48] M. T. Ozsu, *Principles of Distributed Database Systems*, 3rd ed. Upper Saddle River, NJ, USA: Prentice Hall Press, 2007.

[49] G. Coulouris, J. Dollimore, T. Kindberg, and G. Blair, *Distributed Systems: Concepts and Design*, 5th ed. USA: Addison-Wesley Publishing Company, 2011.

[50] J. G. Mitchell and J. Dion, "A comparison of two network-based file servers," *Commun. ACM*, vol. 25, no. 4, pp. 233–245, Apr. 1982. [Online]. Available: http://doi.acm.org/10.1145/358468.358475

[51] B. Liskov, "Distributed programming in Argus," *Commun. ACM*, vol. 31, no. 3, pp. 300–312, 1988. [Online]. Available: http://doi.acm.org/10.1145/42392.42399

[52] B. Liskov, D. Curtis, P. Johnson, and R. Scheifer, "Implementation of Argus," *SIGOPS Oper. Syst. Rev.*, vol. 21, no. 5, pp. 111–122, Nov. 1987. [Online]. Available: http://doi.acm.org/10.1145/37499.37514

[53] B. Liskov, M. Day, M. Herlihy, P. Johnson, and G. Leavens, "Argus reference manual," Cambridge, MA, USA, Tech. Rep., 1987.

[54] B. Liskov and R. Scheifler, "Guardians and actions: Linguistic support for robust, distributed programs," *ACM Trans. Program. Lang. Syst.*, vol. 5, no. 3, pp. 381–404, Jul. 1983. [Online]. Available: http://doi.acm.org/10.1145/2166.357215

[55] Z. Tari and O. Bukhres, "Object transaction service," *Fundamentals of Distributed Object Systems: The CORBA Perspective*, pp. 316–341, 2001.

[56] OASIS Web Services Transaction (WS-TX) TC, *Web Services Atomic Transaction (WS-AtomicTransaction)*, http://docs.oasis-open.org/ws-tx/wstx-wsat-1.2-spec.html, OASIS Standard, Rev. 1.2, Feb. 2009.

[57] ——, *Web Services Business Activity (WS-BusinessActivity)*, http://docs.oasis-open.org/ws-tx/wstx-wsba-1.2-spec-os/wstx-wsba-1.2-spec-os.html, OASIS Standard, Rev. 1.2, Feb. 2009.

[58] ——, *Web Services Coordination (WS-Coordination)*, http://docs.oasis-open.org/ws-tx/wstx-wscoor-1.2-spec-os/wstx-wscoor-1.2-spec-os.html, OASIS Standard, Rev. 1.2, Feb. 2009.

[59] ANSI, *Database Language SQL*, Standard, Rev. ISO/IEC 9075-1, 2011.

[60] T. Haerder and A. Reuter, "Principles of transaction-oriented database recovery," *ACM Comput. Surv.*, vol. 15, no. 4, pp. 287–317, Dec. 1983. [Online]. Available: http://doi.acm.org/10.1145/289.291

[61] C. H. Papadimitriou, *The Theory of Database Concurrency Control*. New York, NY, USA: W. H. Freeman & Co., 1986.

[62] J. E. Moss, *Nested Transactions: An Approach to Reliable Distributed Computing*. Cambridge, MA, USA: Massachusetts Institute of Technology, 1985.

[63] H. Garcia-Molina and K. Salem, "Sagas," in *Proceedings of the 1987 ACM SIGMOD International Conference on Management of Data*. New York, NY, USA: ACM, 1987, pp. 249–259. [Online]. Available: http://doi.acm.org/10.1145/38713.38742

[64] H. Garcia-Molina, D. Gawlick, J. Klein, K. Kleissner, and K. Salem, "Coordinating multi-transaction activities," College Park, MD, USA, Tech. Rep., 1990.

[65] J. Gray, "Why Do Computers Stop and What Can Be Done About It?" in *Symposium on Reliability in Distributed Software and Database Systems*, 1986, pp. 3–12. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.59.6561

[66] G. Weikum, "Principles and realization strategies of multilevel transaction management," *ACM Trans. Database Syst.*, vol. 16, no. 1, pp. 132–180, Mar. 1991. [Online]. Available: http://doi.acm.org/10.1145/103140.103145

[67] B. Lindsay, P. Selinger, C. Galtieri, J. Gray, R. Lorie, F. Putzolu, I. Traiger, and B. Wade, "Single and multi-site recovery facilities," *Distributed Data Bases*, pp. 247–284, 1980.

[68] C. Mohan, B. Lindsay, and R. Obermarck, "Transaction management in the r* distributed database management system," *ACM Trans. Database Syst.*, vol. 11, no. 4, pp. 378–396, Dec. 1986. [Online]. Available: http://doi.acm.org/10.1145/7239.7266

[69] X. Li, Z. Liu, and H. Jifeng, "A formal semantics of uml sequence diagram," in *Proceedings of the Software Engineering Conference. Australian*, 2004, pp. 168–177.

[70] B. W. Lampson, "Chapter 11. atomic transactions," in *Distributed Systems — Architecture and Implementation*, ser. Lecture Notes in Computer Science, D. Davies, E. Holler, E. Jensen, S. Kimbleton, B. Lampson, G. LeLann, K. Thurber, and R. Watson, Eds. Springer Berlin Heidelberg, 1981, vol. 105, pp. 246–265. [Online]. Available: http://dx.doi.org/10.1007/3-540-10571-9_11

[71] B. Lampson and H. Sturgis, *Crash recovery in a distributed data storage system*. Xerox Palo Alto Research Center Palo Alto, California, 1979.

[72] P. A. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[73] Wikipedia, "Three-phase commit protocol," http://en.wikipedia.org/wiki/Three-phase_commit_protocol, Aug. 2014.

[74] I. Jerstad, S. Dustdar, and D. Thanh, "A service oriented architecture framework for collaborative services," in *14th IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprise*, Jun. 2005, pp. 121–125.

[75] D. Booth, H. Haas, F. McCabe, E. Newcomer, M. Champion, C. Ferris, and D. Orchard, "Web services architecture, w3c working group note," *World Wide Web Consortium, article available from: http://www. w3. org/TR/ws-arch*, Feb. 2004.

[76] Open Source Workflow and BPM Blog, "How to distinguish between a business process and a process instance," http://www.processmakerblog.com/bpm\discretionary{-}{}{}2/business\discretionary{-}{}{}process\discretionary{-}{}{}and\discretionary{-}{}{}process\discretionary{-}{}{}instance/, Apr. 2010.

[77] Pinkesh Jain, "Lakozy toyota," http://www.processmaker.com/lakozy-toyota.

[78] C. Atkinson and P. Bostan, "Towards a client-oriented model of types and states in service-oriented development," in *IEEE 13th International Enterprise Distributed Object Computing Conference (EDOC)*, Sep. 2009, pp. 119–127.

[79] A. Barros, M. Dumas, and A. Hofstede, "Service interaction patterns," in *Business Process Management*, W. Aalst, B. Benatallah, F. Casati, and F. Curbera, Eds. Springer Berlin Heidelberg, Sep. 2005, vol. 3649, pp. 302–318.

[80] Eclipse Foundation, "Bpel designer project," https://eclipse.org/bpel/, May. 2012.

[81] W. van der Aalst, A. ter Hofstede, and M. Weske, "Business process management: A survey," in *Business Process Management*, ser. Lecture Notes in Computer Science, W. van der Aalst and M. Weske, Eds. Springer Berlin Heidelberg, 2003, vol. 2678, pp. 1–12. [Online]. Available: http://dx.doi.org/10.1007/3-540-44895-0_1

[82] W. van der Aalst, "Three good reasons for using a petri-net-based workflow management system," in *Information and Process Integration in Enterprises*, ser. The Springer International Series in Engineering and Computer Science, T. Wakayama, S. Kannapan, C. Khoong, S. Navathe, and J. Yates, Eds. Springer US, 1998, vol. 428, pp. 161–182.

[83] C. Ouyang, et al., "Formal semantics and analysis of control flow in ws-bpel," *Science of Computer Programming*, vol. 67, no. 2–3, pp. 162–198, Jul. 2007.

[84] C. Stahl, "A petri net semantics for bpel," *Humboldt-Universität zu Berlin, Mathematisch-Naturwissenschaftliche Fakultät II, Institut für Informatik*, 2005.

[85] S. Hinz, K. Schmidt, and C. Stahl, "Transforming bpel to petri nets," in *Business Process Management*. Springer Berlin Heidelberg, 2005, vol. 3649, pp. 220–235.

[86] N. Lohmann, "A feature-complete petri net semantics for ws-bpel 2.0," in *Web Services and Formal Methods*. Springer Berlin Heidelberg, 2008, vol. 4937, pp. 77–91.

[87] J. E. Hopcroft, *Introduction to Automata Theory, Languages, and Computation*, 3rd ed. Pearson Addison Wesley, 2007.

[88] R. Alur and P. Madhusudan, “Adding nesting structure to words,” *J. ACM*, vol. 56, no. 3, pp. 16:1–16:43, May 2009.

[89] L. Wang, A. Wombacher, L. Ferreira Pires, M. J. van Sinderen, and C.-H. Chi, “A state synchronization mechanism for orchestrated processes,” in *IEEE 16th Intl. Enterprise Distributed Object Computing Conference (EDOC)*, 2012, pp. 51–60.

[90] ——, “An illustrative recovery approach for stateful interaction failure of orchestrated processes,” in *IEEE 16th International Enterprise Distributed Object Computing Conference Workshops (EDOCW)*, Sep. 2012, pp. 38–41. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=6406252&tag=1

[91] L. Wang, L. Pires, A. Wombacher, M. van Sinderen, and C. Chi, “Robust collaborative processes interactions under system crashes and network failures,” in *SERVICE COMPUTATION, The Seventh International Conferences on Advanced Service Computing*. IARIA, Mar. 2015.

[92] L. Wang, A. Wombacher, L. Ferreira Pires, M. J. van Sinderen, and C.-H. Chi, “Robust client/server shared state interactions of collaborative process with system crash and network failures,” in *10th IEEE Intl. Conference on Services Computing (SCC)*, 2013.

[93] ——, “Robust collaborative process interactions under system crash and network failures,” *International Journal of Business Process Integration and Management*, vol. 6, no. 4, pp. 326–340, 2013.

[94] CPN Tools, *CPN Tools*, 3rd ed., AIS Gruop, Eindhoven University of Technology, the Netherlands, http://www.cpntools.org, Jun. 2012.

[95] L. Wang, A. Wombacher, L. Ferreira Pires, M. J. van Sinderen, and C.-H. Chi, “A colloborative processes synchronization method with regards to system crashes and network failures,” in *the 29th Symposium On Applied Computing (SAC)*, 2014.

[96] M. Hawkins and F. Piedad, *High Availability: Design, Techniques and Processes*, 1st ed. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2000.

[97] R. Colville and G. Spafford, “Top seven considerations for configuration management for virtual and cloud infrastructures,” *Retrieved from Gartner database*, 2010.

[98] IBM, “Top seven considerations for configuration management for virtual and cloud infrastructures,” comet.lehman.cuny.edu/cocchi/CIS345/LargeComputing/05_Availability.ppt, 2006.

[99] A. Hesseldahl, “Amazon’s cloud crashed overnight, and brought several other companies down too,” http://allthingsd.com/20110421/amazons-cloud-crashed-overnight-and-brought-several-other-companies-down-too/, 2011.

[100] E. Kaldeli, E. U. Warriach, A. Lazovik, and M. Aiello, "Coordinating the web of services for a smart home," *ACM Trans. Web*, vol. 7, no. 2, pp. 10:1–10:40, 2013. [Online]. Available: http://doi.acm.org/10.1145/2460383.2460389

[101] C. A. Petri and W. Reisig, "Petri net," *Scholarpedia*, vol. 3, no. 4, p. 6477, 2008.

[102] F. A. Haight, *Handbook of the Poisson distribution*, ser. Publications in operations research. New York, NY: Wiley, 1967.

[103] L. Wang, L. Pires, A. Wombacher, M. van Sinderen, and C. Chi, "Stakeholder interactions to support service creation in cloud computing," in *2010 14th IEEE International Enterprise Distributed Object Computing Conference Workshops (EDOCW)*, Oct. 2010, pp. 173–176.

[104] R. J. Wieringa, *Design science methodology for information systems and software engineering*. London: Springer Verlag, 2014.

[105] W.M.P. van der Aalst, A.H.M. ter Hofstede, B. Kiepuszewski, and A.P. Barros, "Workflow patterns," *Distributed and Parallel Databases*, vol. 14, no. 1, pp. 5–51, Jul. 2003.

[106] X. Hu, L. Wang, Z. Sheng, P. TalebiFard, L. Zhou, J. Liu, and V. C. Leung, "Towards a service centric contextualized vehicular cloud," in *Proceedings of the Fourth ACM International Symposium on Development and Analysis of Intelligent Vehicular Networks and Applications*, ser. DIVANet. New York, NY, USA: ACM, 2014, pp. 73–80. [Online]. Available: http://doi.acm.org/10.1145/2656346.2656351

# List of Acronyms

CPN Coloured Petri Net.

DFA Deterministic Finite Automaton.

EDI Electronic Data Interchange.

HTTP Hypertext Transfer Protocol.

IMA Inbound Message Activity.

NFA Non-Deterministic Finite State Automaton.

NWA Nested Word Automata.

ODE Orchestration Director Engine.

OMA Outbound Message Activity.

REST Representational state transfer.

SOAP Simple Object Access Protocol.

TCP Transmission Control Protocol.

WS-BPEL Web Services Business Process Execution Language.

WS-TX Web Services Transaction.

WSDL Web Services Description Language.

XML Extensible Markup Language.

# About the author

Lei Wang has been pursuing his PhD degree in the database group, University of Twente, the Netherlands since 2010. His research interest is to build robust collaborative services interactions without modifying the infrastructure (servers, operating systems or network protocols). The major concern is the interaction failures caused by system crashes and network failures. As the outcome of his research, he has published several research papers in prestigious conferences and journal such as EDOC, SCC, IJBPIM, SAC, etc.

Before going to Twente, Mr. Wang earned his M.Sc degree in School of Software from Tsinghua University, China in 201 During his master study, he was the team leader of a research project: IDAAS(IDentity As A Service) in collaboration with NTT Japan, where the motivation is to achieve a standalone authentication service which is easy to integrate with existing (legacy) information systems. He received his Bachelor degree in Computer Science and Technology from Harbin Institute of Technology, China in 2007.

## List of Publications


L. Wang, L. Pires, A. Wombacher, M. van Sinderen, and C. Chi, "Robust collaborative processes interactions under system crashes and network failures," in *SERVICE COMPUTATION, The Seventh International Conferences on Advanced Service Computing*. IARIA, Mar. 2015.

L. Wang, A. Wombacher, L. Ferreira Pires, M. J. van Sinderen, and C.-H. Chi, "A collaborative processes synchronization method with regards to system crashes and network failures," in *the 29th Symposium On Applied Computing (SAC)*, 2014.

X. Hu, L. Wang, Z. Sheng, P. TalebiFard, L. Zhou, J. Liu, and V. C. Leung, "Towards a service centric contextualized vehicular cloud," in *Proceedings of the Fourth ACM International Symposium on Development and Analysis of Intelligent Vehicular Networks and Applications*, ser. DIVANet. New York, NY, USA: ACM, 2014, pp. 73–80. [Online]. Available: http://doi.acm.org/10.1145/2656346.2656351

L. Wang, A. Wombacher, L. Ferreira Pires, M. J. van Sinderen, and C.-H. Chi, "Robust collaborative process interactions under system crash and network failures," *Interna-*

*tional Journal of Business Process Integration and Management*, vol. 6, no. 4, pp. 326–340, 2013.

——, “Robust client/server shared state interactions of collaborative process with system crash and network failures,” in *10th IEEE Intl. Conference on Services Computing (SCC)*, 2013.

——, “A state synchronization mechanism for orchestrated processes,” in *IEEE 16th Intl. Enterprise Distributed Object Computing Conference (EDOC)*, Sep. 2012, pp. 51–60.

——, “An illustrative recovery approach for stateful interaction failure of orchestrated processes,” in *IEEE 16th International Enterprise Distributed Object Computing Conference Workshops (EDOCW)*, Sep. 2012, pp. 38–41. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=6406252&tag=1

L. Wang, L. Pires, A. Wombacher, M. van Sinderen, and C. Chi, “Stakeholder interactions to support service creation in cloud computing,” in *2010 14th IEEE International Enterprise Distributed Object Computing Conference Workshops (EDOCW)*, Oct. 2010, pp. 173–176.