Multiagent Systems for Robust IoT Services

Kemas Muslim Lhaksmana
Department of Social Informatics
Kyoto University, Japan

Multiagent Systems for Robust IoT Services

Doctoral Thesis Series of
Ishida & Matsubara Laboratory
Department of Social Informatics
Kyoto University

Copyright © 2016 Kemas Muslim Lhaksmana

Abstract

The objective of this thesis is to provide a multiagent systems (MAS) design and implementation methods to support the development of service-based Internet of Things (IoT) systems, or IoT services for brevity, that are robust against changes and failure. The need to realize IoT systems has been increasing due to the growing interest of IoT research and initiatives, which drive the rapid growth of smart objects, i.e. devices having computing and communication capability, that soon will become more pervasive. The main feature of IoT is the possibility to combine different functionalities of these pervasive smart objects and network-accessible applications as combining services by means of service composition. Despite the advantages of IoT, reliability is still one of the challenges to realize its full potential.

The reliability issue is a result of the dynamic nature of IoT due to the mobility of smart objects, distributed nature of IoT applications, and the dynamic of IoT environment. First, some smart objects may be mobile, such as mobile devices, wearable devices, or smart devices embedded to vehicles. Such smart objects may join (leave) to (from) different networks while they are moving across different locations. Consequently, connection with the other required smart objects may become lost. Second, different smart objects and

applications are developed and maintained independently. When a smart object or an application is modified, the other smart objects and applications that require its functionality may not be aware of the modification. Finally, smart objects may be situated in dynamic and unpredictable environment, which require them to respond accordingly upon changes.

This thesis tackles the reliability issue by modeling robust IoT services as self-organizing MAS. Self-organizing systems have autonomous and dynamic characteristics to allow them changing their behavior at runtime. Self-organizing systems, which consist of multiple autonomous systems, are suitable to be modeled as systems of autonomous agents as in MAS. This is also the case in IoT services, which comprise a massive number of smart objects and applications. The robustness of IoT services is realized by introducing two kinds of self-organization capability: *failover flexibility* and *functional flexibility*. Failover flexibility is agent capability to perform failover actions by switching among substitutable agents upon internal or external failures. By contrast, functional flexibility is agent capability to respond against external (in the environment) or internal (in the MAS) changes and failures that require the agent to adjust their behavior to perform different actions.

Against these background, this thesis proposes design and implementation methods to realize flexible IoT services as self-organizing MAS. The contributions of this work are as follows:

1. Provide an IoT services topology that is robust against failure.
   In this direction, the interconnected smart objects and applications in IoT is modeled as a large-scale service network. In the service network, a service node represents a functionality provided by a smart

object or network-accessible application that can be invoked by other smart objects or applications, while a link between two service nodes represents dependency between them. This work analyzes service-network tolerance to cascading failure, which occurs when failure of a service disrupts the other dependent services. The analysis reveals that service network tolerance to cascading failure can be significantly improved by embedding failover flexibility into IoT services to respond upon failure. At the same time, it also suggests service network to have sufficient number of substitutable services such that failover flexibility could efficiently reduce the risk of cascading failure.

2. Introduce a role-modeling method for designing functional flexibility of IoT services.

   Self-organizing MAS is suitable to represent the collection of flexible IoT services as agents since they share common characteristics, such as autonomous, adaptive, decentralized, etc. A modeling method based on role notion is proposed for modeling functional flexibility in MAS. Role notion provides a high-level representation of agent behavior such that agents, i.e. IoT services, can change their behavior at runtime by switching from a set of roles to another to respond upon changes. In contrast to other modeling methods, role modeling separates the design of agent behaviors and agent behavior adaptation (flexibility). The separation improves maintainability and provides designers a convenience way to design agent flexibility. The proposed role-modeling method is suitable for self-organizing MAS since it does not require the MAS to maintain global knowledge as in existing MAS methodologies.

3. Provide a role-based language to implement functional and failover flexibility of IoT services as MAS.

   Developing IoT services with functional and failover flexibility is a complex task since they consist of multiple components and are expected to be situated in dynamic environment. Therefore, the flexibility of IoT services should be implemented as MAS before they are implemented in the actual environment. While self-organizing MAS can be developed using common programming languages, agent-oriented programming (AOP) language is more suitable since the agents in MAS can be directly mapped as agents in the program. However, existing AOPs do not provide a feature to implement higher-level agent behaviors, such as roles, which can be used to easily implement agent flexibility. To solve this issue, Jason, a well-known AOP is extended by introducing role and other related notions in role-modeling method to implement role-based agent flexibility. The proposed language allows agent mental states to be defined in roles such that agents can change their behavior by adopting and releasing roles at runtime, according to role-transition models. Similar to the role-modeling method, the use of role notion also improves maintainability since it separates the implementation of agent behavior and agent flexibility.

These contributions are beneficial for system designers to design self-organizing MAS in IoT services or other domains with similar characteristics.

Acknowledgements

This thesis and my work during Ph.D. study would not be possible without Allah's permission and the help of many people. Firstly and foremost, words are not enough to express my gratitude to my supervisor professor Toru Ishida who has given me the opportunity to forge myself as a Ph.D. student in Ishida & Matsubara Laboratory, Kyoto University. Thank you very much for giving so much persistent guidance that is very valuable for my future career as a lecturer. You also have taught me the skills and attitude that shape me to become a good researcher in the future.

I also would like to express my gratitude to my adviser professor Akihiro Yamamoto. I always received insightful comments from different perspective that enriched my work. I also thank my adviser professor Hirokazu Tatano who always gave fundamental advice that significantly improve the quality of my research.

I also sincerely express my gratitude to associate professor Yohei Murakami who gave me so much encouragement and thoughtful advice for my work. I also thank assistant professor Donghui Lin who guided me to adapt to Ph.D. program and research process, especially during the beginning of my Ph.D. study.

I sincerely thank the faculty at Ishida & Matsubara Laboratory for the wonderful environment and support to learn how to do high quality research: associate professor Shigeo Matsubara, Masayuki Otani, and Takao Nakaguchi. I also would not forget to thank the coordinators of the laboratory: Terumi Kosugi and Hiroko Yamaguchi who gave so much help for both study-related matters and my personal matters as a foreigner living in Japan. I also thank Yoko Kubota who helped me so much for my admission to the Ph.D. program and various matters during my early years. I also thank those who were part of this laboratory for helping me adapting with the laboratory, giving positive contribution to my work, and giving a wonderful research environment: associate professor David Kinny, associate professor Hiromitsu Hattori, and assistant professor Yuu Nakajima.

I sincerely express my gratitude to my fellow laboratory members and past members for being good friends and supporting each other. Special thanks to Shinsuke Goto who gave much help for both my academic and personal matters. Special thanks to Mairidan Wushouer for giving so much advice for my work and how to live as a muslim in Japan. Special thanks to Amit Pariyar, Trang Mai Xuan, Xin Zhou, and Nguyen Cao Hong Ngoc who accompanied the journey throughout Ph.D. study. Special thanks to Arbi Haza Nasution for being able to discuss in my native language. Special thanks to Arif Bramantyo, the first person who connected me to this laboratory and introduced me to professor Toru Ishida. Sincere thanks to Andrew W. Vargo, Hiroaki Kingetsu, Xun Cao, Mondheera Pituxcoosuvarn, Akihiko Itoh, and many others. I also thank past students for their kindness and support: Ari Hautasaari, Chunqi Shi, Huan Jiang, Hiromichi Cho, Taketo Sasaki, and many others. I am grateful to know you all.

My Ph.D. study and life in Kyoto were supported by the scholarship from MEXT, The Ministry of Education, Culture, Sports, Science & Technology, Japan. The research throughout my Ph.D. study was supported by the Grant-in-Aid for Scientific Research from the Japan Society for the Promotion of Science under Grant (S) 24220002 (2012-2016).

I sincerely thank Telkom University, Bandung, Indonesia for their financial support, especially during the last semester of my Ph.D. study. I also thank the then dean of Faculty of Science, Suwandi, for giving me the chance to do Ph.D. study. Special thanks to the School of Computing, Telkom University, more importantly the vice dean Adiwijaya who closely supported me during my study in Japan.

I would like to express sincere thank professor Iping Supriana Suwardi of Department of Informatics, Institute of Technology Bandung, Indonesia who always supported me in applying to a doctoral program. I also thank professor Guido Bakema and Chris Scholten of HAN University of Applied Sciences, Arnhem, Netherlands, who introduced me to research in computer science. I also owe a great debt to all of my teachers who have helped me to grow to become what I am.

Finally, I would not achieve anything without the support of my family and relatives, especially my parents, Kemas Azis Yusuf and Tintin Kartini who always support me through prayers and many ways. I also express my gratitude to my only brother, Kemas Novar Imran P. P. for his support. The most important, I also extend my gratitude to my wife, Nur Hadiyani, and my children, Kemas Abdullah Azzam As-Sajjad and Kemas Muhammad Al-Fatih, for their patient throughout the years and, at the same time, giving me more motivation to finish my Ph.D. study.

Contents

Abstract **i**

Acknowledgements **v**

1 Introduction **1**

1.1 Overview 1

1.2 Objectives 4

1.3 Overview of Solutions 7

1.3.1 Failover Flexibility for Cascading-Failure Tolerant IoT Services 8

1.3.2 Role-Modeling Method for Designing Functional Flexibility of IoT Services 9

1.3.3 Role-Based Language for Implementing Flexible Behavior in IoT Applications 10

1.4 Thesis Outline 10

2 Background **11**

2.1 The Internet of Things 12

2.1.1 An Introduction to the Internet of Things 12

2.1.2 Reliability Issues in the Internet of Things 13
2.1.3 The Internet of Things Services 15
2.1.4 The Internet of Things as Self-Organizing Multiagent Systems 17
2.2 Self-organizing Multiagent Systems 18
2.2.1 An Overview of Self-Organization 19
2.2.2 Self-Organization in MAS 24
2.3 Role Modeling in MAS Engineering Methodologies 29
2.3.1 MAS Engineering Methodologies 29
2.3.2 Generalization of Role Modeling Activities 41

3 Analyses on Cascading Failure in IoT Services 47
3.1 Background 47
3.2 Cascading Failure 50
3.3 Service Network 55
3.3.1 Service Network Model 56
3.3.2 Service Network Properties 58
3.3.3 Publicly Accessible Service Networks 64
3.4 Service Network Generation 70
3.4.1 Artificial Service Network 70
3.4.2 Real Service Networks 75
3.5 Cascading-Failure Simulation and Analysis 77
3.5.1 The Effect of Network Topology on Cascading Failure 80
3.5.2 In-degree Evolution during Network Failure 80
3.5.3 The Effect of Degree of Dependency on Cascading Failure 81

3.5.4 The Effect of Degree of Alternative on Cascading Failure 82
3.6 Summary 83

4 Role Modeling Method for Designing Functional Flexibility 85
4.1 Background 85
4.2 Role Modeling in MAS Engineering Methodologies 87
4.2.1 Role Notion in MAS Engineering Methodologies . . 89
4.2.2 Role Modeling Process 90
4.2.3 Behavior Adaptation Design 91
4.3 Designing Agent Behavior 93
4.3.1 Identifying and Elaborating Roles 94
4.3.2 Designing Relationship and Interaction Between Roles 95
4.3.3 Assigning Roles to Agent Classes 96
4.3.4 Designing Behavior Adaptation 97
4.3.5 Identifying Potential Issues 99
4.4 Case Study: Flexibility in a Traffic Intersection System . . . 101
4.4.1 Identifying and Elaborating Roles 103
4.4.2 Designing Relationship and Interaction Between Roles 106
4.4.3 Assigning Roles to Agent Classes 107
4.4.4 Designing Behavior Adaptation 109
4.5 Summary 111

5 Role-Based Language for Implementing Flexible Behavior in IoT Applications 113

5.1 Background . . . 113
5.2 Modeling IoT Applications as Multi-Agent Systems . . . 115
5.3 Agent-Oriented Programming . . . 117
5.4 Extending AgentSpeak(L) and Jason Interpreter . . . 119
5.4.1 Agent Mental States in Jason . . . 120
5.4.2 Adopting Role Modeling in Jason . . . 120
5.4.3 Extending Jason Grammar . . . 122
5.5 Case Study . . . 124
5.5.1 Description . . . 124
5.5.2 The Implemented Jason Agents . . . 127
5.5.3 Implementation in Jason . . . 129
5.5.4 Implementation in the Extended Jason . . . 131
5.6 Evaluation . . . 135
5.6.1 Features of the Extended Jason . . . 135
5.6.2 Maintainability . . . 137
5.7 Summary . . . 138

6 Conclusion and Discussion **140**
6.1 Contributions . . . 142
6.2 Future Direction . . . 144

Publications **146**

Bibliography **148**

List of Tables

2.1 Role definitions in MAS engineering methodologies 44

3.1 The Language Grid (LG) and ProgrammableWeb (PW) service network properties 64

4.1 Comparison of existing MAS engineering methodologies . . 88
4.2 The role modeling process 91
4.3 Potential issues in playing multiple roles 98
4.4 Roles in the traffic intersection system 105

5.1 New features in the extended Jason compared to the original Jason 136

List of Figures

1.1 The research framework 7

2.1 The AGR framework 31

3.1 An IoT-based traffic intersection system 53
3.2 Service network representation of the traffic intersection system 57
3.3 In-degree distribution of scale-free, exponential, and random service networks 58
3.4 Two abstraction level of service composition in The Language Grid 61
3.5 In-degree distribution of The Language Grid and ProgrammableWeb service networks 65
3.6 In-degree distribution of scale-free, exponential, and random service networks 68
3.7 Cascading failure in service networks with different degree of dependency $\langle dep \rangle$ and different degree of alternative $\langle alt \rangle$ 71
3.8 The in-degree of each random failed service and the average in-degree of the remaining active services during cascading-failure simulation 76

3.9 The out-degree distribution of scale-free and ProgrammableWeb service networks 77
3.10 The effect of degree of dependency $\langle dep \rangle$ and degree of alternative $\langle alt \rangle$ on cascading failure in service networks . . . 78

4.1 The metamodel of the role-modeling method 93
4.2 The metamodel of role-relationship model 94
4.3 The metamodel of role-interaction model 95
4.4 The metamodel of role-enactment model 97
4.5 The metamodel of role-transition model 97
4.6 The IoT-based traffic intersection system in normal operation 100
4.7 The IoT-based traffic intersection system with a vehicle sensor failure 102
4.8 The role-relationship model of the traffic intersection system 106
4.9 The role-interaction model of the traffic intersection system . 107
4.10 The role-enactment model of the the agent classes in the traffic intersection system 108
4.11 Different types of enactment and role association in role-enactment model 108
4.12 The role-transition model of Congestion Predictor agent class 109
4.13 The role-transition model of Image Processor agent class . . 110
4.14 The role-transition model of Sidewalk Camera agent class . 110

5.1 Diagram of some core classes of the extended Jason 121
5.2 Architecture of the extended Jason interpreter 124

Chapter 1

Introduction

1.1 Overview

In the near future, the Internet of Things (IoT) vision will become reality, in which everyday devices, sensors, and actuators are embedded with computing and communication capability enabling them to interact with each other, surrounding environment, and Internet applications. Smart objects, i.e. devices with computing and communication capability, will become more pervasive since sensors, mobile devices, and embedded devices become smaller, cheaper, yet having better performance [Guinard et al., 2010a]. As one of the emerging technology nowadays [LeHong and Fenn, 2013], IoT is predicted to comprise more than 25 billion devices by 2020 [Middleton et al., 2014, Evans, 2011], in which more than one-third (above 10 billion) are mobile devices [Cis, 2015, GSM, 2015]. By 2019, the number of smartphones and mobile M2M devices are predicted to reach 70%

among IoT mobile devices, while the number of wearable devices are growing significantly. The rapid growth of mobile devices is also driven by the improvement of mobile network connection speed. In average, connection speed is expected to increase more than twofold by 2019 (4.0 Mbps) within 5 years (1.7 Mbps in 2014) [Cis, 2015].

One of the main advantages of IoT is the possibility to combine different functionality from different smart objects and network-accessible applications in the Internet. This will become more possible due to the rapid growth of publicly accessible services and the adoption of service-oriented computing (SOC) into IoT. Much research effort has been done for the adoption of SOC, such as abstracting IoT devices as services [Guinard et al., 2010a] and allow them to interact by means of service-based technology such as RESTful and SOAP [Haller, 2010]. The adoption of SOC enables combining different functionality from smart devices and publicly accessible services by means of service composition [Geyik et al., 2013, Song et al., 2010, De Souza et al., 2008]. In this thesis, such service-based IoT systems are called IoT services for brevity.

Behind the promise of IoT, some research challenges still need to be solved to realize its full potential. Reliability, robustness, and fault tolerance are, among others, the fundamental issues in IoT [Miorandi et al., 2012, Abdelwahab et al., 2014]. These issues are the consequence of the dynamic nature of IoT due to the mobility of smart objects, distributed characteristic of IoT applications, and the dynamic and unpredictable environment where the IoT applications are situated.

First, the mobility of smart objects could raise reliability issue when, for instance, the interacting smart objects suddenly lost connection and experi-

ence failure because of changing to different network while moving to different location. Without proper failover method, such failure would be common since smartphones and mobile M2M devices are predicted to become more pervasive and dominant in IoT. These mobile smart objects may continuously join (leave) to (from) different networks while they are moving, and thus dynamically change their connections with different smart objects and different networks.

Another factor of the reliability issue is the distributed characteristic of IoT applications since they comprise a collection of smart objects interacting with each other. Different smart objects may be maintained by different companies, and thus modification of a smart object may not be realized by the other smart objects that require its functionality and interact with it. The smart object may remove some of its functionality, change the required parameters, or change the usage policy. This may cause the other smart objects fail to execute the functions that were provided before it is modified.

Finally, IoT applications are expected to be situated in dynamic and uncertain environment [Geyik et al., 2010, Ding et al., 2013, Steine et al., 2015]. Therefore, smart objects should capable of responding to changes in the surrounding. In addition, smart objects interact with other smart objects and environment by using different kind of communication protocols. Low power network communication protocols based on IEEE 802.15.4, such as 6LoWPAN and ZigBee, have been considered to be widely implemented for IoT [Guinard et al., 2010b, Cirani et al., 2014, Catarinucci et al., 2015]. While they are dedicated for short-range and low data-rate M2M communication, they are more likely to experience interference and failure compared to those consume higher power such as IEEE 802.11 WiFi.

1.2 Objectives

This thesis aims to provide a solution for the reliability issue in IoT by proposing multiagent systems (MAS) design and implementation methods to develop robust IoT services. The proposed solution consists of fault tolerant network topology, a modeling method, and a programming language for implementing flexible IoT services, i.e. service-based IoT systems whose components are capable of changing the way it operates at runtime to respond upon changes and failure.

Flexible IoT services can be developed as self-organizing MAS, where each IoT service is modeled as an adaptive agent. The idea to embed runtime flexibility into computing systems has been addressed in autonomic computing [Kephart and Chess, 2003] and self-organizing systems [Ishida et al., 1992, Kamboj and Decker, 2006, Kota et al., 2012]. Autonomic computing, self-organization, and other self-* solutions [de Lemos et al., 2013], are proposed in response to the growing complexity of computing systems. The trend of computing systems have shifted from large-centralized systems to the integration of multiple smaller systems and the utilization of cloud services. This is also the case in IoT systems, which comprise a massive number of smart objects connected to each other and to Internet applications.

The robustness of self-organizing MAS against changes and failure is realized by embedding the flexibility to change behavior at runtime. Two kinds of flexibility are introduced: failover flexibility and functional flexibility. Failover flexibility is agent capability to perform failover actions by switching among substitutable agents upon external failure in the environment or internal failure in the MAS. By contrast, functional flexibility is the agent

capability to change its behavior to enable different actions. Throughout this thesis, agent functional flexibility is also called agent behavior adaptation. Behavior adaptation is mainly triggered by external changes in the environment where the self-organizing MAS are situated. However, internal changes in the MAS such as changes on some agents or in the relations between the agents could also trigger behavior adaptation.

To implement these two kinds of flexibility in developing IoT services, this thesis is directed to achieve the following objectives:

1. To analyze the effect of services dependency on cascading failure in IoT services and how failover flexibility could significantly improve the tolerance to cascading failure.
   The dependency between services in service-based IoT systems may cause cascading failure, i.e. a situation when failure of one service propagates to the other dependent services. To analyze cascading failure, IoT services are modeled as service network, in which each node represents a service and each link connecting two nodes represents dependency between two services. In IoT services, these nodes are smart object functions that wrapped as services and publicly accessible by other smart objects. To design a reliable service network against cascading failure, it is important to analyze how failover flexibility could improve the tolerance to cascading failure.

2. To provide a modeling method for designing flexible IoT services.
   Flexible IoT services should be designed as self-organizing MAS such that they are capable to change their behavior at runtime. The requirement to change behavior at runtime is contrast to traditional engineering methodologies, in which a set of fixed functionalities is

defined at design time. Unfortunately, most of existing MAS engineering methodologies do not provide suitable methods to design agent flexibility. Several MAS engineering methodologies that specifically address self-organization have some drawbacks, such as requiring the agents to have global knowledge and using centralized approach for MAS reorganization. Providing a modeling method for designing agent flexibility will give a contribution for self-organizing MAS design.

3. To provide a programming language for implementing functional and failover flexibility of IoT services.
   The flexibility of IoT services should be implemented in a simulated environment before they are realized in the actual environment. The use of agent-oriented programming (AOP) language would be convenience to implement flexible IoT services as agents since agent-oriented properties, such as agent classes, goals, and actions, can be mapped and implemented directly in the program. However, most of existing AOP languages do not provide a high-level notion to easily implement agent flexibility. In addition, AOP languages do not provide a feature to separate the implementation of agent behavior and agent behavior adaptation (agent flexibility) which is required to improve maintainability and provide convenience feature for designing agent behavior adaptation.

The first objective is directed to provide an analysis on cascading failure and how failover flexibility could improve failure tolerance. The second objective focuses on providing a modeling method for designing functional flexibility. Finally, the third objective is intended to support the implementa-

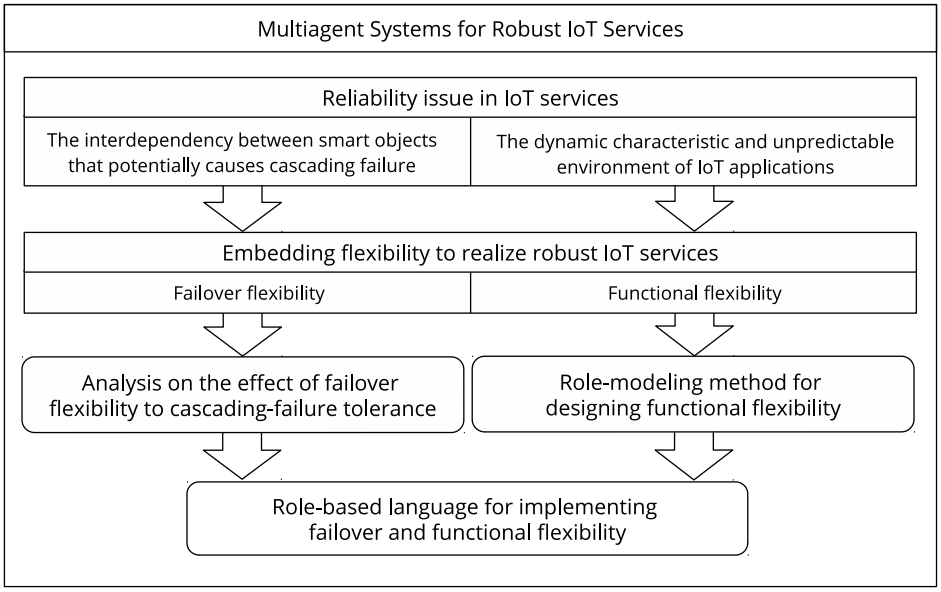


Figure 1.1: The research framework.

tion of both failover and functional flexibility. These three objectives lead to providing a modeling technology to support the design of flexible systems as self-organizing MAS.

1.3 Overview of Solutions

As illustrated in Fig. 1.1, this thesis proposes solutions for reliability issue in IoT services. The robustness of IoT services is realized by embedding functional and failover flexibility as self-organization capability in MAS. The overview of the proposed solutions are given as follows.

1.3.1 Failover Flexibility for Cascading-Failure Tolerant IoT Services

Cascading failure possibly occurs in interdependent systems, including IoT services, where failure of one system component disrupts the other dependent components. Even though much cascading failure research has been done in power network and critical infrastructure domains, it has not been analyzed in IoT and service network. This motivates the research to analyze cascading failure in IoT services that is intended to gain more understanding about IoT service network tolerance to cascading failure. Specifically, the objectives of this research are as follows. First, to analyze how the degree of interdependency between services affects service network tolerance. Second, to analyze how failover flexibility could improve the tolerance. Finally, to analyze cascading failure in different network topology.

To achieve the objectives of this research, a number of experimental service networks were generated based on existing service network topology. These service networks were generated such that they exhibit different topology, degree of dependency, and degree of alternative. The two degree represents the average level of dependency between the services and the average number of substitutable services, respectively. A number of random failures were simulated on these generated service networks, which then triggered cascading failure. The effect of failover flexibility to cascading-failure tolerance was analyzed by calculating the number of services that failed due to cascading failure.

1.3.2 Role-Modeling Method for Designing Functional Flexibility of IoT Services

Role-modeling method is proposed to design functional flexibility of IoT services as self-organization capability in MAS. Role-based modeling is central in MAS engineering as it is used in most of MAS engineering methodologies. The notion role has been widely used in sociology and was introduced to object oriented software engineering and MAS engineering [Kendall, 1998]. In MAS, a role is a set of functionalities, a position of duty, or an aggregation of behaviors to be played by agents. As the notion role has been extensively used to explain the behaviors of individuals in a society [Biddle, 1986], role modeling is considered suitable to design agent flexibility since it provides a natural way to model agent behavior in MAS.

The proposed role-modeling method is a set of modeling activities that guide MAS designers to model functional flexibility of agents, i.e. to model how agents change their behaviors at runtime to enable different actions. The designers conduct a series of modeling activities to create four model: (i) role relationship model (RRM), (ii) role interaction model (RIM), (iii) role enactment model (REM), and (iv) role-transition model (RTM). The first two models design the relationship and interaction between roles. In the third model, REM, the designers assign roles to agent classes and defines the configuration of the roles in the agent class. Finally, the core of the proposed role-modeling method is RTM which is used to design how agents change their behaviors at runtime by making transition from a set of roles to another.

1.3.3 Role-Based Language for Implementing Flexible Behavior in IoT Applications

The proposed role-modeling method (Section 1.3.2) provides a way to design functional flexibility using role notion. Additionally, failover flexibility is also possible to be designed using role-modeling method. Having designed agent flexibility using role-modeling method, designers need suitable AOP to implement the design. To this end, a programming language is proposed based on AgentSpeak(L) [Rao, 1996] that implemented on Jason interpreter [Bordini and Hübner, 2006]. In line with the role-modeling method, the proposed language supports role notion and allows the separation of agent behavior and agent behavior adaptation.

1.4 Thesis Outline

This thesis is organized into six chapters. Chapter 2 presents the background of this thesis in more detailed. IoT is reintroduced by addressing its vision, current status, issues, and challenges. Since IoT services are modeled as self-organizing MAS, this chapter also provides a brief introduction to self-organization, MAS, and some existing self-organization solutions in MAS. Additionally, it also explains why IoT can be modeled as self-organizing MAS. Finally, a concise introduction to SOC is provided to explain how IoT systems can be modeled as service network and show some related work in this direction. The next two chapters following Chapter 2 correspond to the proposed solutions as briefly introduced in Section 1.3. Finally, this thesis is concluded and future directions is addressed in Chapter 6.

Chapter 2

Background

This chapter introduces IoT and addresses some related subjects such as MAS and SOC to put the thesis into context. This thesis is motivated by IoT issues and challenges in the near future. As a newly emerging techonology [Middleton et al., 2014], IoT offers promises and opportunities by enabling physical objects around us to interact with each other, with other network-accessible applications, and with the environment including people. Despite the rapid growth of smart objects, some research issues and challenges should be solved to realize its full potential, such as reliability issue, which is the focus of this thesis.

2.1 The Internet of Things

2.1.1 An Introduction to the Internet of Things

The term "Internet of Things" was firstly coined by Kevin Ashton in 1999 to support automation of supply chain by utilizing RFID for object identification. Even though the idea was intended for industrial application, the general idea is still relevant until today, i.e. to integrate physical objects to the Internet and allow them to automatically generate data and perform more automation [Ashton, 2009]. Besides this idea, some other relevant effort has been done by other researchers. In a paper published in 1991, Mark Weiser envisioned ubiquitous computing where computing systems, which he referred as silicon-based information technology, should become part of the environment around us. However, in contrast with the pervasiveness of today's computing systems that are very apparent, he argued that ubiquitous computing should be seamlessly integrated into the world [Weiser, 1991].

Some of the notable work on connecting physical objects to network and Internet was also realized around those years. One of the first was a network-accessible Coke vending machine that allowed people to check its status through a network application [Nichols, 1990]. In 1990, a toaster that could be remotely operated is also regarded as the first machine connected to the Internet [Zakon, 1997]. The Trojan Room Coffee Pot, originally developed in 1991, consisted of a camera which periodically captured the image of the coffee pot. A client-server application was implemented to provide a continuously updated image of the pot to the users [Stafford-Fraser, 2001]. Despite its simplicity as compared to today's objects technology, these im-

plementations introduce some of the key ideas in IoT, such as connecting physical objects to the Internet, providing network-accessible information of the objects, and allowing them to be remotely controlled.

The advance of Internet technology and other related progress in computing have driven smart objects to become more pervasive. The development of IPv6 has made possible to assign approximately 3.4×10^{38} addresses to connected devices. This is much more than its predecessor (IPv4) that only holds approximately 4.3 billion addresses. In the future, IoT devices will be dominated by mobile and M2M devices. By the end of 2014, the number of mobile-connected devices is believed to exceed the number of people on earth (more than 7 billion), and it will reach more than 10 billion by 2020 [Cis, 2015, GSM, 2015]. In 2014, 29% of these devices were smartphones, while M2M devices were only 7%. However, in 2019 these two are expected to reach 40% and 28%, respectively. In addition, wearable devices, which were only 109 million (2014), will become more popular and are expected to reach 578 million in 2019. The rapid growth of mobile devices is also driven by the mobile network connection speed, which in average is expected to increase more than twofold by 2019 (4.0 Mbps) from 1.7 Mbps in 2014 [Cis, 2015].

2.1.2 Reliability Issues in the Internet of Things

Apart from its promises, IoT is facing some issues and challenges, such as security and privacy, reliability, robustness, and fault tolerance [Miorandi et al., 2012, Abdelwahab et al., 2014]. This thesis focuses on reliability issue, which arises from the dynamic nature of IoT due to the mobility of

smart objects, distributed characteristic of IoT applications, and the dynamic and unpredictable environment where the IoT applications are situated.

Mobile smart devices including smartphones, wearable devices, and mobile M2M devices are becoming more pervasive. Mobile devices are capable to stay operational and connected while moving through different locations. However, mobility could raise reliability issue when, for instance, the interacting smart objects suddenly loss connection and experience failure because of changing to different networks while one of them moving to different locations. In such scenario, mobile smart objects need to dynamically change their connections with different smart objects and different networks because of joining (leaving) to (from) different networks.

As systems that consist of a number of smart objects, the distributed characteristic of IoT applications is also a factor that cause reliability issue. Even though some smart objects interact with each other, they may be developed by different companies. When one smart object is modified, other smart objects that interact with it may not be aware of the modification. Some modifications, such as changing the required parameters, the usage policy, or even the functionality, may cause the dependent smart objects fail to keep the interaction. Another factor is the dynamic and uncertain environment where IoT applications are situated [Geyik et al., 2010, Ding et al., 2013, Steine et al., 2015]. This requires IoT applications to have the capability to respond upon environment changes. The dependency between the interacting smart objects and between smart objects and services may also cause another issue which is called cascading failure. This kind of failure occurs when a smart object or a service fails and the dependent smart objects also experience problems because of this failure.

Since the use of smart devices and other IoT-related technology have been more pervasive, these reliability issues become more important to be solved. Some IoT-based systems require higher degree of reliability compared to others. For example, traffic intersection systems that utilize smart devices and other kinds of IoT objects should have high degree of reliability since they involve cars and pedestrians. Failure in such systems can cause problems from congestion to accident. Therefore, tackling the reliability issue in IoT applications is of importance. To this end, this thesis proposes solutions for the reliability issue by embedding failover and functional flexibility into IoT. To address failover flexibility, dependency between smart objects is modeled as a service network. Cascading failure in IoT service network is simulated to investigate the effect of failover flexibility on cascading-failure tolerance. As for functional flexibility, the interacting smart objects are modeled as MAS with self-organization capability. These service network and MAS abstractions are explained in the following.

2.1.3 The Internet of Things Services

The core advantages of IoT are the pervasiveness of Internet-connected smart objects and the capability of the smart objects to interact with each other, with the environment, and with accessible applications in the Internet. The possibility to interact smart objects with any other network-accessible resources in the Internet allows people to create new smart objects and services that combines the functionality of the existing ones. This is in line with service composition in SOC that allows a new service to compose several existing services.

The SOC paradigm was proposed to realize networks of cooperating services by combining distributed applications as network-accessible services. Different applications should be easily assembled to combine different functionalities regardless of their heterogeneity in computing platforms, environments, etc. [Papazoglou et al., 2008]. The adoption of SOC technology to IoT also brings solution to the heterogeneity issue in IoT due to differences in hardware, platform, and communication protocols of smart objects [Vasseur and Dunkels, 2010, Im et al., 2013]. To make their heterogeneity becomes transparent, smart objects should wrap their functionalities as services so that they can be invoked using existing service communication protocols.

The idea to adopt service composition for combining smart object functionalities has been addressed in some research that leads to the following directions.

1. Abstracting IoT objects as services
   The abstraction of IoT objects as services, as it has been addressed in some related work [Geyik et al., 2013, Zeng et al., 2011, Stecca et al., 2015], can be made possible since IoT objects and services share common characteristics. IoT objects and services comprise multiple independent components that communicate to each other by exchanging data [Vasseur and Dunkels, 2010].

2. Utilizing SOC technology for object communication
   IoT objects are heterogeneous in that they are of different hardware and employ various communication protocols. The main purpose in abstracting IoT objects as services is to allow these heterogeneous objects to easily communicate with each other [Geyik et al., 2013, Zeng

et al., 2011, Stecca et al., 2015]. To this end, commonly used service technology, such as RESTful and SOAP, could be utilized [Haller, 2010].

3. Enabling the interaction between IoT objects and web services
   The capability of IoT objects to interact (and to be accessed) as services opens the possibility to combine IoT objects and web services. Some related work uses the term "Internet of Services" as the vision of the future Internet, in which everything, including IoT objects, can be accessed as services and cooperate with each other [Lizcano et al., 2008, Guinard et al., 2010a, Moreno-Vozmediano et al., 2013, Schroth and Janner, 2007].

Despite its potential, combining functionalities of smart objects also brings reliability challenge since the availability of composite services depends on the availability of the required services they combined. If one of the required services is unavailable, the composite services that depends to it also becomes unavailable. The chain of dependency between composite services and the composed services forms a service network where each node represents a service and each link represents the dependency between two services.

2.1.4 The Internet of Things as Self-Organizing Multiagent Systems

In addition to IoT services, IoT systems can also be considered as self-organizing MAS since they exhibit similar characteristics. First, a collection

of smart objects in IoT applications are analogous to autonomous agents in MAS since they interact with each other without external control. Second, IoT applications have decentralized characteristic because the interaction between smart objects are performed without the need to have centralized control. To address this further, the discussion on self-organizing MAS is provided next.

2.2 Self-organizing Multiagent Systems

Current development of computing systems, including the rapid growth of smart objects and the trend of IoT, increases the need to integrate heterogeneous systems. This leads to a new level of complexity that cannot be handled by human interventions [Kephart and Chess, 2003]. Inspired by nature and social phenomenon, autonomic computing and self-organizing systems have been proposed to solve this problem by embedding automation and self-governing capability into computing systems [Kephart and Chess, 2003, Kota et al., 2012].

Self-organizing systems have several unique characteristics, such as autonomous, decentralized, adaptive, dynamic, and having emergent behavior where the individual behavior gives rise to the novel behavior of the system at the global level. The first two characteristics are also common in MAS where the agents are also autonomous, responsive, and proactive [Jennings et al., 1998]. This suggest that it is quite natural to model and design self-organizing systems as MAS, i.e. as self-organizing MAS. Moreover, there are many MAS engineering methodologies that may be utilized for the de-

velopment of self-organizing systems.

Apart from the similarity between self-organizing systems and MAS characteristics, there are some characteristic that may not be supported by existing MAS engineering methodologies. Self-organizing systems designer should also consider the fact that MAS engineering methodologies are not intended to design such systems. For example, self-organizing systems are adaptive and dynamic where the behavior and the configuration of the systems may change at runtime. To realize such systems, instead of defining a set of fix functionalities at design time, the designers should design how the system may adapt itself. In other words, the development of self-organizing systems requires an engineering methodology that provides modeling methods to design how agents adapt their behavior at runtime.

2.2.1 An Overview of Self-Organization

Definitions

The term "self-organizing systems" was firstly coined by Ashby, who defined "organizing" as "changing from unorganized to organized" or "changing from a bad organization to a good one" [Ashby, 1947, Ashby, 1962]. The prefix "self" indicates that the organizing process is performed autonomously by the system itself. Self-organization is coherence to the IBM's vision of autonomic computing which was motivated by the increasing need to integrate heterogeneous computing systems, which leads to a new level of complexity that soon cannot be handled by human interventions [Kephart and Chess, 2003]. Autonomic computing [Kephart

and Chess, 2003] and self-organize systems [Kota et al., 2012] have been proposed to solve this challenge by embedding self-governing capability into computing systems. Self-managing capability of autonomic computing consists of, but not limited to, four aspects: self-configuration, self-optimization, self-healing, and self-protection [Kephart and Chess, 2003]. In this thesis we use both terms "self-organization" and "self-organizing systems". The former refers to the mechanism, while the latter refers to the systems which employ self-organization.

Another term that is related to self-organization and has been widely used is "self-adaptive". Self-organizing systems are expected to be situated in a dynamic and uncertain environment where the characteristics of the environment may change [Kota et al., 2012]. A number of self-organizing systems in nature, as well as in human societies, have adaptive characteristic. The adaptation capability is needed to respond against perturbations, as well as internal and external changes. Self-adaptive systems are capable to adapt itself in response to their perception of the environment and the system itself [Weyns et al., 2012, Cheng et al., 2009].

Classifications

To date, researchers have proposed classifications of self-organization. According to the existence of centralized control to perform reorganization process, self-organizing systems are categorized into *strong self-organizing systems* and *strong self-organizing systems* [Serugendo et al., 2006]. In the former, reorganization is performed without any explicit and centralized control. By contrast, the latter perform reorganization using internal

centralized control and planning.

Classification of self-organization in wireless cellular systems have been proposed, in which self-organization mechanisms are divided into three phases: *self-configuration*, *self-optimization*, and *self-healing* [Aliu et al., 2013]. Even though it was intended for specific domain, it is also applicable for other fields. Self-configuration focuses on the process of applying the right settings to enable the systems to achieve expected performance and to comply with expected situation and environment. The objective of self-optimization is to allow the systems to organize itself to achieve best performance. Finally, self-healing is the process to recover the systems from failure. A classification of self-healing systems has been proposed in a survey which categorizes self-healing systems into three paradigms: maintenance of health, detection of systems failure, and system recovery, each of which is hierarchically categorized further [Ghosh et al., 2007].

Most of self-organization mechanisms are inspired by self-organization in nature [Di Marzo Serugendo et al., 2004]. Since humans are self-organize beings, their social behavior also inspired self-organizing systems [Hassas et al., 2006]. This brings another notable classification in which self-organization is classified into *bio-inspired*, *social-based*, and *artificial-based* mechanisms.

Characteristics

Recently, self-organization trends toward decentralization, openness, imitation of nature, and reliance on simulation [Parunak and Brueckner, 2011]. Self-organization, both in nature or artificial mechanisms, share common

characteristics. Serugendo et al. listed a number of self-organization characteristics [Di Marzo Serugendo et al., 2005]. We list the most important characteristics as follows:

1. Autonomous

   Autonomy is the ability to act and respond to external and internal stimuli without external control. By definition, a self-organizing system is a system that organize itself autonomously. Therefore, every self-organizing system is autonomous but not an autonomous is not necessary self-organizing [Parunak and Brueckner, 2011].

2. Decentralized

   Decentralization levels can be seen as a continuum that span from centralized to purely decentralized [Parunak and Brueckner, 2011]. Decentralization in self-organization mostly ranging from intermediate to purely decentralized. Even though decentralization is not a mandatory characteristic [Di Marzo Serugendo et al., 2005], existing self-organization mechanisms show that the agents act and make decision mainly based on local knowledge. An example of a decentralization in complex system is an ants colony, in which each ant acts autonomously without centralized control [Gordon, 1996]. Decentralization also leads to better system performance by avoiding bottleneck and centralization overhead especially in a system where the number of agents are high.

3. Adaptive

   Self-organizing MAS, as well as self-organizing systems in nature, are likely to be situated in uncertain and dynamic environment. The system must be capable of responding against external and internal

changes and perturbations. Handling changes and perturbations autonomously is one of the prime motivations of self-organizing systems. The robustness of a self-organizing system is determined by the adaptive capability of the system.

4. Dynamic

   Autonomy and adaptation in a self-organizing system drive the system to evolve. In fact, biologists believe that evolution plays important role in developing self-organization behavior in nature. A self-organizing system may have multiple stable states, and thus dynamically makes transition from one state to another.

5. Having emergent properties

   Many self-organizing systems show macro level behaviors resulting from individual behaviors at micro level. In emergence system, the behavior at the macro level is novel and cannot be deduced from the behavior at the micro level. For example, foraging ants behavior that uses pheromone trails generates shortest path between their nest and food source [Deneubourg et al., 1990].

These characteristics are compatible with MAS which consist of a number of autonomous agents. These agents are flexible agents since they should capable to perceive and respond to their environment (*responsive*), to take the initiative to act in opportunistic and goal-oriented manner (*proactive*), to interact with each other and with other entities in the environment (*social*) [Jennings et al., 1998]. Therefore, it is quite natural to model and design self-organizing systems as MAS.

Some notable work puts more emphasize on the adaptive characteris-

tic [Weyns et al., 2012, Cheng et al., 2009, Qureshi and Perini, 2008]. This characteristic can be considered as the core characteristic, since self-organizing systems are built to embed the capability to change itself, i.e. to perform self-adaptation. An adaptive and autonomous system is self-organizing if it is capable of maintaining the system as expected despite of the dynamic of its environment [Aliu et al., 2013].

2.2.2 Self-Organization in MAS

Due to the compatibility between the characteristics of self-organizing systems and MAS, and the increasing needs to embed self-* capability into computing systems, research in self-organizing MAS is growing. Existing work proposes various self-organization, from primitive mechanism which based on duplication and merging agents, until more complex mechanisms such as those involve diagnostic systems and learning. Here, we briefly explain some of the notable work in this field.

Organization Self-Design

The self-* terms have been addressed in distributed artificial intelligence (DAI) and MAS. Corkill and Lesser introduced organizational self-design component that is responsible to initially develop and modify organizational structure in a distributed problem solving network [Lesser and Corkill, 1980, Corkill and Lesser, 1983]. Also named as organization self-design (OSD), Ishida et al. proposed a self-organization mechanism for MAS based on two primitives: composition and decomposition [Ishida et al., 1992].

Composition is a combination of two agents into one to release computing resources and reduce communication overhead between agents. On the other hand, decomposition divides an agent into two to improve parallelism and increase the capability to handle more concurrent requests.

Kamboj and Decker extended the OSD approach by incorporating TÆMS (task analysis, environment, modeling, and simulation) as the underlying problem representation and introducing robustness [Kamboj and Decker, 2006]. TÆMS is a formal, domain-independent computational framework to describe task structures of agents [Horling et al., 1999]. The root node of the structure represents a high-level goal, the sub-nodes are its subtasks and methods, and the leaf nodes are executable methods. The adoption of TÆMS to OSD allows the representation of a larger, more general class of problems and quantitative reasoning over task structures. Their work is also based on the work of Shehory et al. [Shehory et al., 1998] on agent cloning and merging. In agent cloning, the cloned agents have the same roles and responsibilities as its original. It differs with agent spawning (decomposition) where the spawned agents are specialized on a subpart of the spawning agent's task structures [Kamboj and Decker, 2006].

Diagnostic System Approach

Horling et al. proposed a diagnostic subsystem within agent architecture that responsible to detect faulty conditions and inefficiencies in the organization [Horling et al., 2001]. The diagnostic subsystem is build based on blacboard-based design. It comprises three layers: symptoms, diagnoses and reactions. This layering allows one to easily trace the symptoms of

the diagnoses, as well as the diagnoses of the reactions. Moreover, these layers are divided by time to provide access to the history of activities on each layer. Symptoms are contained in the lowest level of the blackboard. It is generated by two classes of components: observers and modelers. Observers monitor various aspects of the agents and generate appropriate symptoms. Modelers use different approach. It builds or learns models and uses it as a basis for comparison.

The diagnosis process used in the subsystem is based on the causal model. The model is represented as a directed acyclic graph where the nodes represent diagnoses and the edges show causative relations between the related diagnoses. Causal model allows more detailed diagnoses to be determined from a rather general diagnosis. However, diagnostic does not always resulting detailed and specific diagnoses. Sometimes it can only reach an initial hypothesis with low level of confidence due to limited evidence.

Another mechanism that also identifies inefficiencies in the network is the work of Hoogendoorn [Hoogendoorn, 2007] that uses max flow network [Ford and Fulkerson, 1956], a widely used network model in graph theory. The work proposed two algorithms to perform organizational adaptation. The goal of both algorithms is to increase the max flow of the network. The first adapts organizational structure by modifying the bottleneck of the network. The bottleneck of the network is identified and then the capacity is improved to increase the organization's performance by identifying paths from source to sink that can be increased by one. The second method works by adding organizational elements such as roles and group links to the organization. Intuitively, the algorithm finds the nodes and edges representing the roles of which the copy would improve the max flow most. If no

such nodes and edges found, it identifies the nodes and edges representing the role having most connection in the organization.

Structural Adaptation

An extant work of Kota et al. argues that most of the self-organization mechanisms may not be applicable due to several reasons [Kota et al., 2012]. First, the structural adaptations which based on agent spawning and composition [Ishida et al., 1992, Kamboj and Decker, 2006] require the modification of the internal characteristics of the agents. This may not be applicable on all situations because of physical and accessibility limitations. They also argued that the use of a diagnostic subsystem [Horling et al., 2001] requires the designers to identify all the states of the system. Against these issues, they proposed a kind of network of agents in which the relationship between agents are categorized into several levels of interaction, and thus structural adaptations are performed by changing the *relationship level* from one to another. The relationship levels from the lowest interaction frequency to the highest, are *stranger*, *acquaintance*, *peer*, and *superior-subordinate*. Whenever an agent receive a task request, it will try to perform the task by itself, or to delegate the task to one of its subordinates if it is not capable. If there is no subordinate that capable to perform the task, then the agent will try to delegate the task to one of its peers, and then to its superior if no peer is capable. Finally, the superior will do the same steps to perform or delegate the task. In this kind of organization, it is clear that the organization performance is influenced by the relationship configuration between the agents. The work proposed structural adaptation that modify the relationship level between agents by calculating the best relationship level of

each relation [Kota et al., 2012].

Team Formation

Gaston and desJardins proposed dynamic formation of an agents organization that works solely based on rewiring connections of agents, i.e. it does not allow the creation of new network connections, neither removal of existing connections [Gaston and desJardins, 2005]. The work defines agent-organized network (AON) as: an organizational network structure, or agent-to-agent interaction topology, that is the result of local rewiring decisions made by the individual agents in a networked MAS. A MAS is a networked if it has an explicit set of interactions or interdependencies among the agents. The model used in the work is motivated by other previous works on agent team formation [Abdallah and Lesser, 2004, Kraus et al., 2003].

Tasks are generated periodically and broadcasted to agents in the organization. Teams are formed to accomplish the tasks. The agent teams are formed in completely decentralized and distributed manner. The decision making process of each participating agent in forming the team solely depends on its local information. Within the organization, an agent is part of a social network. To become a team member, an agent must have a social connection with at least an existing team member.

Learning for Self-Organization

To date, learning has been utilized in some self-organization mechanisms for MAS. The first notable work that addresses learning is the work of Ab-

dallah and Lesser that adopts multiagent reinforcement learning (MARL) technique to allow the agents to self-organize the underlying network using the information obtained from learning [Abdallah and Lesser, 2007]. As well as the other mechanisms that we have addressed, the self-organization mechanism in their work is performed by rearrange the network topology to achieve better network organization. In doing so, the mechanism consists of deciding which neighbor to add or remove, when to stop adding or removing neighbor, and how to update agent's knowledge after the adaptation.

2.3 Role Modeling in MAS Engineering Methodologies

2.3.1 MAS Engineering Methodologies

Within the last two decades, a number of well-known MAS engineering methodologies have been proposed, such as Gaia [Wooldridge et al., 2000], SODA [Omicini, 2001], ADELFE [Bernon et al., 2003], Prometheus [Padgham and Winikoff, 2003a], INGENIAS [Pavón and Gómez-Sanz, 2003], Tropos [Bresciani et al., 2004], PASSI [Cossentino et al., 2005], ASEME [Spanoudakis and Moraitis, 2011], and O-MaSE [DeLoach and García-Ojeda, 2010]. Role notion is used by most of the aforementioned engineering methodologies, except Prometheus and ADELFE, to represent agents behavior. Prometheus uses another concept to avoid confusion as the term is defined differently between different MAS engineering methodologies [Padgham and Winikoff, 2003a]. ADELFE purposely

omits the notion to allow the designers to focus on designing cooperative agents [Bernon et al., 2005]. In this section, first we give the overview of role notion and the use of role in MAS engineering. Next, we briefly address the existing MAS engineering methodologies. Next, we address the engineering process and particularly the role modeling in MAS engineering methodologies. We focus on role modeling because it is central in most of the engineering methodologies to design agents behavior, and thus can be exploited to design adaptive behavior for self-organizing MAS.

The term role has been commonly used in social sciences. Despite a number of definitions of role in sociology, it is believed that the theory of role is a theatrical metaphor of social life [Biddle, 1986]. A role is the expected characteristic behavior of an actor who plays a part based on a script. In this sense, in social context the concept of role consists of three components: (i) pattern and characteristic of social behaviors, (ii) parts or identities that are assumed by social participants, and (iii) scripts or expectations for behaviors [Biddle, 1986].

Role and role modeling have also been introduced in object oriented software engineering. Role models are complementary to class models [Kendall, 1999]. A role model identifies and describes the interactions of objects with each other in terms of roles. A role represents a position characterized by a set of responsibilities to be assigned to objects [Kendall, 2001].

The use of role notion in existing MAS engineering methodologies is somewhat influenced by AGR modeling framework. This model was proposed by Ferber and Gutknecht as core concepts to model organizations as MAS in AALAADIN [Ferber and Gutknecht, 1998] and organization centered multi-

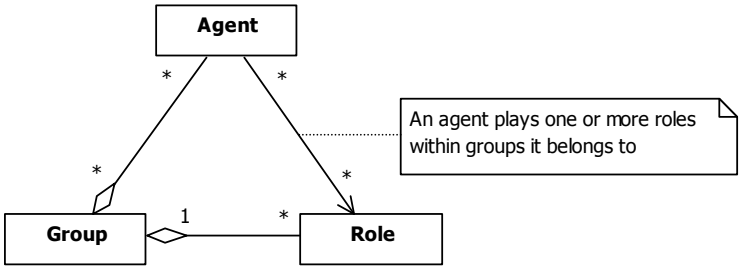


Figure 2.1: The relationship of agent, group, and role in AGR (adapted from the work of Ferber and Gutknecht [Ferber and Gutknecht, 1998])

agent system (OCMAS) [Ferber et al., 2004] metamodels. The relationship between agent, group, and role in AGR is illustrated in Fig. 2.1. A group is an aggregation of agents, and each agent is a member of one or more groups. A group consists of one or more roles that is local to the group. These notions allow one to model an organization in terms of relations between groups and roles, and defined agents as active communicating entities that play one or more functions (roles) in the organizations [Ferber and Gutknecht, 1998]. This is a natural way to model MAS as it can be seen in daily life where a person plays one or more roles within one or more societies. For example, a researcher can also be a lecturer, a head of a department in a university, and a member of some scientific associations.

Currently, none of the MAS engineering methodologies provide modeling method to design how agents change their role at runtime. However, related work shows that it is possible to use roles to model dynamic behavior of agents. Such feature has been introduced in *behavior graph* [Weyns et al., 2005]. However, in behavior graph, an agent is limited to play only one role at a time. Behavior graph is also comparable to state-transition diagram

that is commonly used to model system state transitions. However, both behavior graph and state-transition diagram are not modeling methods since they do not provide step-by-step guidelines to define the roles (states) and transitions. Therefore, they cannot be used independently to model dynamic behavior of agents.

Gaia

Role modeling is central in Gaia methodology since the notion role is used throughout the activities. A role is characterized by responsibilities, permissions, activities, and protocols. Among other attributes, responsibility is the prominent attribute which define the functionality of the role. Permissions define the role's rights to access and use available resources. Activities of a role are private actions that can be performed, i.e. actions that does not require interaction with other agents. Lastly, the ways to interact with other roles are defined as protocols. Analysis phase in Gaia comprises a series of activities which model roles. The activities start with identifying roles which usually represent individuals within an organization, departments, or organization itself. Next, each role is defined and elaborated further in terms of its properties [Wooldridge et al., 2000].

Typically, analysis phase is followed by design phase where the resulting analysis model is detailed further to be easily implemented. Gaia methodology mainly covers analysis phase in MAS engineering. Gaia is not an end-to-end engineering methodology that can be use by the designers to perform the whole engineering phases from analysis until implementation. The design phase in Gaia is intended to provide a low level of abstraction that can

be applied by the designers using other conventional engineering methodologies to realize the MAS. In the design phase, the roles are grouped into agent types, and the interactions between roles (services model) and between agent types (acquaintance model) are defined [Wooldridge et al., 2000].

SODA

SODA is a MAS engineering methodology that specifically tailored for analyzing and designing Internet-based systems [Omicini, 2001]. The distinction of SODA compares to other methodologies is its clear separation between intra-agent and inter-agent aspects. The modeling of inter-agent aspect in SODA allows the designers to put more focus on modeling the society of agents and the environment in which the agents are situated. SODA uses the notions that are commonly used by other methodologies, such as task, role, group, agent class, and resource. These notions are used throughout the engineering process that consists of two phases: *analysis* and *design*.

The *analysis* phase in SODA defines the problem domain and the application goals. This phase consists of three models: *role model*, *interaction model*, and *resource model*. Task is the primary notion in SODA that represents the system's goals to be achieved. A role is defined by the assigned tasks, permissions to access resources, and interaction protocols. In the *role modeling* activity, the tasks are identified and defined in terms of involved responsibilities, required competences, and required resources. Next, these tasks are assigned to roles and groups based on its type. There are two kinds of task: *individual task* and *social task*. The former is the responsibility of

an individual role, whereas the latter is assigned to group since it requires a number of different competences and access to several resources. Finally, the interaction of roles, groups, and resources are defined in the *interaction model*. In the *design* phase, based on the result of the *analysis* phase, the designers create abstract models that can be directly realized to be the actual system. This phase consists of *agent model*, *society model*, and *environment model*. The association of roles to the agent classes is performed in *agent model* where each agent class is defined as a set of roles.

INGENIAS

INGENIAS is a MAS engineering methodology equipped with a set of tools. In this methodology, the designers model MAS from five viewpoints: agent, organization, goals/tasks, interactions, and environment. The functionality of the agents is defined in agent viewpoint in terms of responsibilities and capabilities. Similar as in Gaia, responsibilities define the goals that must be achieved by the agent. Capabilities are the abilities of the agent to perform tasks. Agents have mental states to be able to make decisions based on information. The interaction between agents, either human agents or machine agents, are defined in the interaction viewpoint. The environment entities that interact with the MAS are defined in the environment viewpoint that consists of resources and applications. Finally, the organization viewpoint defines the structure of the MAS in terms of agents, environment entities, and goals/tasks. As well as other methodologies, INGENIAS also uses the notions of roles and groups. The functionality of the agents can be organized into roles. An organization consists of a number of groups in which agents, roles, resources, or applications are aggregated

[Pavón and Gómez-Sanz, 2003]. INGENIAS follows unified software development process (USDP) that consists of inception, elaboration, and construction phases. Analysis and design activities are conducted within each phase [Pavón et al., 2005]. Recently, there are a number of developments in INGENIAS to improve the productivity by using model-driven approach [García-Magariño et al., 2011, Gómez-Sanz et al., 2010, García-Magariño et al., 2009b, García-Magariño et al., 2009a].

Tropos

Tropos is a MAS engineering methodology which specifically includes requirement analysis phase adopted from *i** [Yu, 1997] requirement engineering. The development phases of Tropos consists of: *early requirements*, *late requirements*, *architectural design*, *detailed design*, and *implementation*. The first two phases constitute the requirement analysis phase. In the *early requirement* phase, the designers model the domain stakeholders as social actors and define the dependency relationship among them. In the next phase, the dependencies are refined to specify the system's functional and non-functional requirements [Bresciani et al., 2004]. Tropos allows the designers to develop agents which have mentalistic states based on belief, desire, and intention (BDI) architecture [Rao and Georgeff, 1998].

Tropos engineering methodology is based on a number of concepts, such as: actor, goal, plan, resource, dependency, capability, and belief. An actor represents an entity that has goals and intention such as a physical, social, or software agent. Tropos also uses the concept of roles and position adopted from Actor Dependency model [Yu and Mylopoulos, 1994]. A role is an

abstraction of the social actors, whereas a position is an aggregation of a set of roles.

Recently the development of Tropos is directed to extend the methodology to develop self-* systems [Giorgini et al., 2008] and socio-technical systems (STS). Some efforts to incorporate self-* and runtime reorganization to Tropos exist [Dalpiaz et al., 2008, Qureshi and Perini, 2008, Dalpiaz et al., 2009] but have not yet integrated.

PASSI

PASSI [Cossentino et al., 2005] consists of 12 activities (subphases) from requirement phases until implementation and deployment. These activities are grouped into five models: *system requirement model*, *agent society model*, *agent implementation model*, *code model*, and *deployment model*. The engineering approach in PASSI is a combination of sequential and iterative approach. As well as other MAS engineering methodologies, PASSI uses the notions agent, role, goal, task, etc.

A role represents an agent behavior in terms of a goal to be achieved and a functionality or a service it provides [Cossentino, 2003]. It has a number of tasks, each of which is a unit of individual or interactive behavior. PASSI has two modeling activities that model roles: *role identification* (R.Id.) and *role descriptions* (R.D.). R.Id. is conducted within the *system requirement* phase. In *agent society modeling*, the designers conduct R.D. modeling activity using class diagrams to describe roles to be played by agents.

The engineering process in PASSI starts with describing the functionality of the system using use case diagrams. The use cases are grouped into

packages where each package represents an agent and its use cases are the agent's functionalities. The relations between use cases in different packages are considered as communications between the agents. Based on these communication paths, the scenarios of agent interactions are defined using an extensions of sequence diagram, where objects in the diagram represent roles. This is where the roles are identified by the designers. An agent may play different roles in different scenarios. Next, the modeling focuses on the capabilities of each agent in task specification (T.Sp.). A use case diagram is created for each agent to describe its capabilities where each use case represents a task. The role modeling in R.D. is conducted based on the results of R.Id. and T.Sp. to create class diagrams describing the agent classes and the roles they play, the tasks involve, the communications, messaging, and dependencies among them. PASSI allows the agents to change roles at runtime, which is also modeled in R.D [Cossentino, 2005].

ASEME

Agent systems engineering methodologies (ASEME) [Spanoudakis and Moraitis, 2011] is the MAS engineering methodology that applies end-to-end model-driven engineering (MDE) approach throughout its engineering phases. MDE is applied by employing automatic model transformation from requirement analysis to code generation. Metamodels used in ASEME methodology are based on agent modeling language (AMOLA) [Spanoudakis and Moraitis, 2008]. ASEME engineering process consists of six phases: *requirement analysis*, *analysis*, *design*, *implementation*, *verification*, and *optimization*. ASEME consists of three level of abstractions: *society level*, *agent level*, and *capability level* [Paraschos et al.,

2012]. The primary notions used in ASEME, among others, are actor, goal, and role.

In the *requirement analysis* phase, the main actors of the system and its goals are defined using *system actors and goals* (SAG) model. The role modeling in ASEME is conducted in the *analysis* phase. Here, the SAG model previously defined is transformed into *system use case* (SUC) model by transforming actors into roles and goals into use cases. The uses cases represent the capabilities of the roles. In this modeling activity, the designers may add or modify roles and uses cases. Besides the SUC modeling, the *analysis* phase also includes modeling the *agent interaction protocol* (AIP) model and *system roles model* (SRM). The former defines how to coordinate the use cases played by two or more cooperative roles, while the latter defines the protocols and the liveness of each role [Paraschos et al., 2012]. In SRM, a role is defined in terms of activity, capability, and protocol. The next models following SRM are *inter-agent control* (EAC) model and *intra-agent control* (IAC) model. The AIP model is transformed into the former, while the SRM is transformed into the latter, one IAC for each role. Both models are basically statechart diagrams to model events, conditions, and actions in describing role's process [Spanoudakis and Moraitis, 2011, Paraschos et al., 2012].

O-MaSE

O-MaSE [DeLoach and García-Ojeda, 2010] is an organization-based extension of MaSE engineering methodology [DeLoach et al., 2001]. O-MaSE supports open system where the agents can join by registering

themselves and their capabilities at runtime. The methodology is based on OMACS framework [DeLoach, 2009] which is also based on AGR model [Ferber and Gutknecht, 1998]. The core concepts of OMACS comprise agents (A), goals (G), roles (R), and capabilities (C). The ability of a role to achieve a goal is defined by the function *achieves*: $G \times R \rightarrow [0..1]$. The capabilities required to play a role is defined by the function *requires*: $R \rightarrow P(C)$ which returns a set of capabilities. An agent can play roles if it possesses the capabilities to play the roles which are defined by the function *possesses*: $A \times C \rightarrow [0..1]$. In OMACS, reorganization is performed by finding the optimal organization by going through every combination of agents, goals, and roles. Therefore, the reorganization process typically requires high computation cost when the number of agents, goals, and roles are high.

O-MaSE engineering phases can be grouped into three: *requirements analysis* (*problem analysis* and *solution analysis*), *design* (*architecture design* and *low level design*), and *implementation* (code generation). These phases can be applied sequentially or iteratively. In the *problem analysis*, the designers analyze the problem in terms of the purpose of the system and the environment in which the system will be situated by using a *goal model* and a *domain model*, respectively. Next, the *solution analysis* phase describes the behavior of the system and how the system interacts with the environment based on the models defined in the previous phase. The behavior is defined in terms of roles in the *role model*, while the interactions are described in the *organization model*. In the *design* phase, the designers translate the models from the *requirement analysis* phase to produce lower-level models of the system that can be implemented in code generation phase.

Role modeling in O-MaSE is conducted particularly in the *solution analysis*. First, the designers define a number of roles with respect to the leaf goals defined in the *organization goal* model. Each leaf goal must be covered by at least one role, and each role must be assigned by at least one leaf goal. Next, the designers define the interactions among the roles and the interactions between the roles and external actors by using *role model*. Each role is elaborated in terms of the goals to be achieved, the required capabilities, constraints, and plans in *role description document* and *role goal model*. In the *design* phase, the designers model the agent classes, and then roles, capabilities, or both are assigned to the agent classes.

Other Methodologies

Other well known methodologies that worth to be noted are ADELFE [Bernon et al., 2003] and Prometheus [Padgham and Winikoff, 2003a]. These methodologies do not use role and role modeling. ADELFE is based on AMAS framework [Capera et al., 2003] that is specifically tailored to develop adaptive MAS consisting of cooperative agents. In Prometheus, the unit of agent behaviors is defined in terms of functionality. A functionality is defined as narrow as possible by focusing on single aspect or goal of the system. Next, the functionalities are grouped and this grouping is used to identify what kinds of agents will exist in the system [Padgham and Winikoff, 2003a]. Therefore, in Prometheus the aggregations of the agent behaviors are performed by grouping the functionalities, and the assignments of behaviors to the agents are directly based on these groups.

2.3.2 Generalization of Role Modeling Activities

Even though all of the methodologies use role as one of the notions to model agents behavior, the role-modeling methods are different from each other. The methodologies define roles using different properties between each other (Table 2.3.2). Different diagrams are also used. Some methodologies adopt commonly used diagrams from UML and AUML, whereas others define their own diagrams. The methodologies also define different modeling process with each other. Despite these differences, the role modeling process can be generalized into four activities: (i) identification, (ii) elaboration, (iii) interaction design, and (iv) association. We discuss these activities in more detailed in the following subsections. These generalized role modeling activities can be used as the basis to extend role modeling for self-organizing MAS development.

Identification

In some methodologies, the roles are derived from entities in other previous models, whereas in some others the roles are identified solely based on the designers decisions. Gaia suggests to identify roles based on individuals, departments, or organizational entities. However, the later version of Gaia identifies roles based on identifying the basic skills (functionalities and competences) required for the organization. Even though SODA is influenced by Gaia, roles in SODA are identified based on tasks grouping. In INGENIAS, roles are identified by organizing use cases. Both Tropos and ASEME use the notion actor. In the former, a role is a special kind of actor, while in the latter roles are derived from actors. In O-MaSE, roles are iden-

tified based on leaf goals. However, in Tropos, subgoals are identified from each actors, and then more actors are created to handle these newly created subgoals. In PASSI, use cases are grouped into packages, where each package is identified as an agent, and the roles will eventually created based on the scenario of each agent.

Based on the way roles are identified, role modeling in MAS engineering methodologies can be categorized into three [Tran and Low, 2005]: (i) goal oriented (roles are identified based on goals), (ii) behavior oriented (roles are identified based on system's behaviors and tasks), and (iii) organization oriented (roles are identified based on system's organizational structure). However, goals in some engineering methodologies are basically similar with functionalities in others which express what the system is expected to do. Therefore, we group SODA, INGENIAS, PASSI, and O-MaSE into the behavior oriented approach. Even though roles in O-MaSE are based on leaf goals, the goals also represent functionalities that express what the system should do. ASEME belongs to organization oriented approach, while Gaia and Tropos belong to both behavior oriented and organization oriented approaches.

Most of the methodologies identify roles in analysis phase. An exception of this is Tropos in which the modeling activities focus on actor notion, and role is used to describe the behavior of social actor. The identification of roles in analysis phase indicates the significance of role modeling in MAS engineering as it is used in the early phase of engineering process. From role modeling perspective, this also shows that roles identification drives the whole role modeling activities.

Elaboration

Having identified the roles to be played by the agents, the designers define each role in more detailed in terms of its properties. Table 2.3.2 lists role definitions according to each methodology. Even though all of the methodologies that we have discussed use roles to abstract or aggregate behaviors and characteristics, they differ in defining the properties of the roles. For example, Gaia characterizes roles in terms of responsibilities, permissions, activities, and protocols. This is somewhat followed by SODA that also uses permissions and protocols to define roles. However, as a methodology that uses task as its main notion, SODA also assigns tasks to roles so that the goals to be achieved by the roles are expressed through the tasks. Roles in PASSI and O-MaSE are also characterized by the goals to be achieved, while ASEME uses activity, capability, and protocol to define a role. In INGENIAS, a role is an organization of agents' functionality. Tropos uses the notion role in relation to actor, agent, and position notions.

The elaboration of roles can be performed within a particular modeling activity as in Gaia, ASEME, and O-MaSE. In Gaia, a preliminary role model is designed in analysis phase, and then it is transformed into a complete role model in architectural design phase. In ASEME, roles are defined in more detailed within SRM in terms of name, liveness, activity, capability, and protocol. Role modeling in O-MaSE produces three kinds of artifacts: *role model*, *role description document*, and *role goal model*. *Role description document* elaborates roles in terms of the capabilities required, the goals to be achieved, constraints, and plans. In *role goal model*, the goals to be achieved by the roles are decomposed further into capabilities to achieve the corresponding goals. Other methodologies elaborate roles during identify-

Table 2.1: Role definitions in MAS engineering methodologies

Engineering methodology	Role definition
Gaia	An abstraction of expected functions and competences A position of duty such as an office Characterized in terms of responsibilities, permissions, activities, and protocols
SODA	An aggregation of tasks to be played by the individuals (agents) in the society (MAS) defined in terms of permissions to access resources and the corresponding protocols A task represents a system's goal to be achieved defined in terms of responsibilities, required competence, and required resources
INGENIAS	A role is an organization of the functionality of the system to be played by an agent
Tropos	The notions actor, agent, role, and position are strongly related An actor represents an entity that has goals and intention, such as a physical, social, or software agent, as well as a role or a position A role is an abstraction of the behavior of a social actor A position is an aggregation of a set of roles
PASSI	A behavior of an agent in terms of a goal to be achieved, functionality or service it provides, and the tasks it has
ASEME	An abstraction of agent behavior defined in terms of activity, capability, and protocol
O-MaSE	An abstraction of behaviors to be played by agents A role is defined in terms of goals to be achieved, required capabilities, constraints, and plans

ing roles or assigning the roles to the agent types. For example in PASSI, R.D. modeling activity uses class diagram to associate roles to be played by the agents, and also defined the involving tasks, communications, messaging, and its dependencies.

Interaction Design

Agents interact with each other and with other entities in the environment. The behaviors in doing interaction may be defined in roles. Additionally, role modeling also includes defining the relationship between roles which may or may not be required for designing the interactions. The definition of role and the properties that characterize role also influence the way the interactions and relationships to be modeled in the methodology.

In Gaia, SODA, and ASEME, the ways to interact with other roles is defined in protocol. The role modeling activities in Gaia and SODA also include defining the relationships between roles and between roles and resources. In Gaia, the relationship between roles and resources within the environment is defined in terms of permissions. Permissions are also used in SODA to define roles, along with tasks as its primary notion. The interactions between agents in PASSI are defined by modeling the interactions of roles using an extension of sequence diagram.

The interactions and relationships between roles and between roles and other entities are modeled using commonly used diagram from UML and AUML or using the methodology's diagrams. In Tropos, the interaction between roles are reflected from the interactions between agents in *interaction diagram* which is based on AUML sequence diagram. INGENIAS allows

the designers to use AUML protocol diagram, UML collaboration diagram, or its own diagram called GRASIA to specify interactions between agents and between roles. PASSI and ASEME also adopt commonly used diagram for roles interaction design, which respectively are *sequence diagram* and *statechart diagram*.

Association

The influence of AGR model to the methodologies can be seen in the use of role notion together with the other related notions such as agent class (Gaia, PASSI, O-MaSE) and group (INGENIAS). PASSI uses class diagram to map the roles to agent classes, while O-MaSE uses agent class model which is similar to class diagram. Tropos uses the notions position and actor in relation to roles and agent. Agent type, agent class, and group aggregate and categorize agents, while a position can be seen as an aggregation of roles and an intermediate notion between agent and roles.

Both O-MaSE and ASEME explicitly allow multiple roles to be played by an agent, either concurrently or sequentially change one role to another at runtime. Such ideas have been mentioned in the earlier methodology such as Gaia [Wooldridge et al., 2000], but the methodology has not fully supported this feature. OMACS framework, the foundation of O-MaSE, provides a mechanism to reorganize a MAS including reorganize agents' roles to achieve better organization structure [DeLoach, 2009].

Chapter 3

Analyses on Cascading Failure in IoT Services

3.1 Background

The future Internet will be populated with a massive number of publicly available services integrated to the Internet [Lizcano et al., 2008]. The rapid growth of services and their mashups is driven by the extensive use of SOC, cloud computing, and the availability of APIs provided by the well-known Internet companies [Huang et al., 2013]. On the other hand, IoT vision also pushes the number of smart devices which is predicted to reach more than 25 billion devices by 2020 [Middleton et al., 2014, Evans, 2011]. As one of the emerging technology nowadays [LeHong and Fenn, 2013], IoT attracts a number of research to realize its full potential. One of the key issues is the heterogeneity of IoT, where the smart devices are designed for dif-

ferent purposes and environment, accessible using different communication protocols, implemented on different hardware and platforms, etc. [Atzori et al., 2010, Miorandi et al., 2012, Stankovic, 2014]. Against this challenge, international standardization institutions and IoT-based projects have initiated the adoption of services computing paradigm and technology into IoT [Zanella et al., 2014].

The SOC paradigm was proposed to realize networks of cooperating services by combining distributed applications as network-accessible services. Different applications should be easily assembled to combine different functionalities regardless of their heterogeneity in computing platforms, environments, etc. [Papazoglou et al., 2008]. The idea to combine different services to create a new service, which is called service composition, has been addressed to be applied in IoT [Geyik et al., 2013, Issarny et al., 2011, Ortiz et al., 2014]. Combining different functionalities from different smart devices as combining services can be made possible by providing a service-oriented middleware [De Souza et al., 2008, Song et al., 2010, Teixeira et al., 2011], enabling the smart devices as service platforms [Guinard et al., 2010a], and employing service-based technology, such as RESTful and SOAP [Haller, 2010]. Consequently, these cooperating services form a large-scale service network, where the nodes are the services, and the links are the dependency between composite services and their required component services.

Dependency between interacting services may cause cascading failure, where the failure of a service triggers the failure of one or more its dependent services. To the best of our knowledge, this type of cascading failure has not been studied. Understanding cascading failure in service net-

work is significant since real networks have such dependency between the nodes and potentially experience the same type of cascading failure [Little, 2002, Gupta et al., 2008]. The tolerance of a service network to cascading failure is determined by the topology of the network and the degree of interdependency between the services. Against this background, the objectives of this research are as follows: (i) to analyze the effect of interdependency between services to the tolerance, (ii) to investigate which topology has better tolerance to cascading failure, and (iii) to analyze how in-degree evolution during network failure affects cascading failure.

The contributions of this work are as follows. First, regarding the effect of service interdependency, we investigate how the average number of required services (degree of dependency) and the average number of substitutable services (degree of alternative) contribute to the tolerance. We find that the number of cascade failed services, i.e. the services that fail cascade-wise, is somewhat linear to the degree of dependency and inversely proportional to the degree of alternative. The latter suggests that adding a few more substitutable services for each component service significantly improves the network tolerance to cascading failure when the degree of alternative is low; this is not so effective if the degree of alternative is already high. Second, we find that scale-free topology provides better tolerance, and then subsequently followed by exponential and random topology. Finally, we also find that the number of cascade failed services for each random failed service decreases over time as a result of the in-degree evolution during the network failure.

The discussion in this chapter is organized as follows. Section 3.2 briefly discusses cascading failure and related work. In Section 3.3, we present

the service network model, some relevant properties of service networks, and two publicly accessible service networks. Next, the algorithm to generate service networks is presented in Section 3.4. We generate service networks with different topology and different service interdependency, on which cascading-failure simulations are performed. The simulation settings and analysis are presented in Section 3.5. Finally, we conclude our analysis in Section 3.6.

3.2 Cascading Failure

In distributed systems, failure is classified into several types according to its effect on the systems. The first one is called *crash failure*, or also known as *fail-stop*, i.e. the type of failure that causes a system to halt. *Omission failure* occurs when a system is unable to respond to incoming requests. *Timing failure* occurs when a system fails to respond within the specified time interval. *Response failure* is experienced by a system when it gives incorrect respond upon incoming request. Finally, *arbitrary failure*, is also sometimes called *Byzantine failure*, is the type of failure when the system responses arbitrarily in case of failure [Tanenbaum and Steen, 2006].

Any of these types of failure may be cascaded from one system to another neighboring system, or from one component to another. From network point of view, cascading failure is a situation where the failure of a node (link) cascades to one or more nodes (links) in the network. Cascading failure has been studied in some domains, especially in the study of critical infrastructure, where cascading failure is defined as a disruption in one infrastructure

that causes disruption in another [Rinaldi et al., 2001]. Cascading failure has gained more interest since major cascading failure events [Little, 2002] and power grid failures affecting large areas occurred in the past, such as those in Italy [Rosato et al., 2008], North America [Kinney et al., 2005], Australia, and New Zealand [Ash and Newth, 2007]. The Internet industry also have experienced cascading failure. One of the notable failures is the Amazon EC2 outage that disrupted some major Internet companies that were using their infrastructure services [Armbrust et al., 2009].

According to the direction of the cascade, we classify cascading failure into two types: *horizontal cascading failure* and *vertical cascading failure*. Cascading failure is vertical when the failure creates a path along the links to either direction, or to both direction such as in metabolism network [Smart et al., 2008]. The cascade direction on vertical cascading failure is determined by the dependency among the neighboring nodes. By contrast, cascading failure is horizontal when the failure can cascade from one node to another non-adjacent node. This type of cascading failure is widely studied, especially in power networks, where cascading failure is triggered by load-redistribution among the links [Crucitti et al., 2004]. Load-redistribution occurs when a link fails and the load that was handled by the link is redistributed among the available links. However, if the available links cannot handle the additional load, they will also fail because of overloaded.

In service networks, both vertical and horizontal cascading failure are possible. Dependency between services potentially causes vertical cascading failure, while server load redistribution may cause horizontal cascading failure as in power network. Exploring cascading failure in service network is important to gain more understanding about vertical cascading failure. Many

real networks, such as infrastructure networks and the interconnection between components of an aircraft [Little, 2002, Gupta et al., 2008], exhibit similar dependency as in service network, and thus potentially experience vertical cascading failure. However, to the best of our knowledge, vertical cascading failure in service networks has not been studied. This motivates our work to analyze vertical cascading failure in service networks.

To illustrate cascading failure in IoT, an IoT-based traffic intersection system (Fig. 3.1) is provided as a case study in this chapter and also in the following chapters throughout this thesis. IoT-based traffic systems traffic systems have been addressed in smart city projects, such as in Padova, Italy [Zanella et al., 2014], Oulu, Finland [Kostakos et al., 2013], and Santander, Spain [Sanchez et al., 2014]. The traffic intersection system is chosen as the case study because it is suitable to demonstrate flexibility in IoT-based application. The objective of the traffic intersection system in this case study is to control the traffic lights properly, i.e. to minimize the likelihood of congestion, according to the number of vehicles and pedestrians around the intersection. The system consists of some devices and sensors, such as vehicle sensors that detect incoming vehicles. The objects in the figure are only a part of the actual system that may include some more objects and applications. Moreover, these objects are connected to other objects and network-accessible applications in the Internet, and thus they are part of larger network that we refer as a large-scale service network.

To address cascading failure in the example, let us observe Congestion Predictor and Vehicle Sensors. Congestion Predictor predicts the congestion in the intersection according to the number of vehicles detected by the vehicle sensors that are located in each direction. When one of these sensors fails,

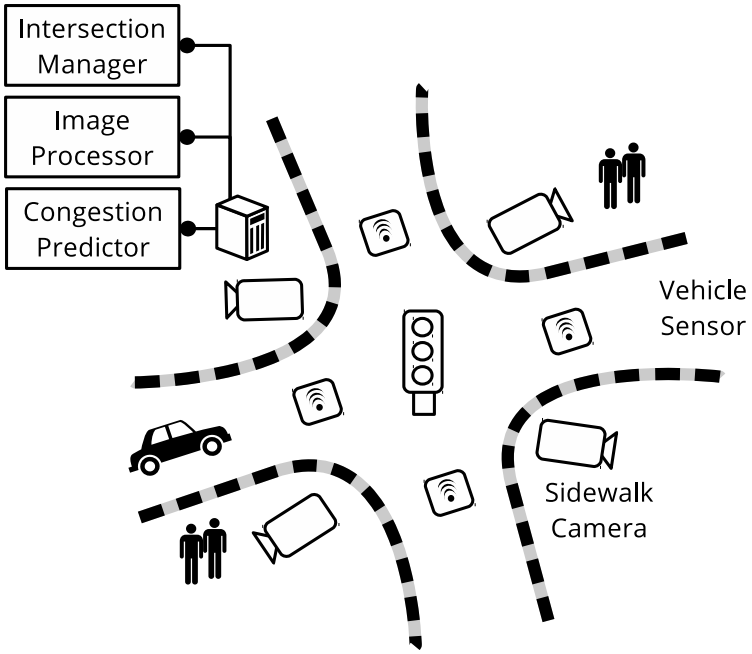


Figure 3.1: An IoT-based traffic intersection system that consists of some interconnected sensors and devices. The traffic lights are controlled by an intersection manager according to traffic situation around the intersection. The traffic situation data are provided by congestion predictor that predicts congestion based on the number of vehicles coming (leaving) to (from) the intersection. Vehicle sensors located in each direction sends these number of vehicles to the congestion predictor. A camera at each corner is installed to monitor pedestrians and the sidewalks. The number of pedestrian is counted by an image processor unit, which is also capable to count the number of vehicles from an image.

either crash failure or another type of failure, Congestion Predictor may not be able to predict congestion accurately. For example, if a sensor experiences crash failure, Congestion Predictor will not receive the number of vehicles that should be detected by the sensor. If it experiences another type of failure, such as arbitrary failure, the sensor may send incorrect number of vehicles. In both cases, Congestion Predictor will not be able to predict accurately the congestion due to incomplete information of number of vehicles in the intersection. This is an example of cascading failure when failure of the vehicle sensor disrupts Congestion Predictor. The failure may be cascaded further to other dependent nodes. In this case, Intersection Manager that controls the traffic light may become unable to set the traffic light timing appropriately because it does not receive congestion alert from Congestion Predictor. To reduce the risk of cascading failure, fail nodes should be restored or the system should provide some redundancy that provide a backup in case of failure.

Object and service discovery has been considered to be one of the key challenges in IoT [Zorzi et al., 2010]. Some IoT research has proposed solutions for service discovery, such as using P2P overlay [Cirani et al., 2014], device profile for web services (DPWS) [Han et al., 2015], social networking for devices [Atzori et al., 2012], and middleware [De Souza et al., 2008, Song et al., 2010, Teixeira et al., 2011]. Service discovery has also been addressed in other related fields, including pervasive computing [Raychoudhury et al., 2013], ambient intelligence [Stavropoulos et al., 2013], and machine-to-machine communication [Villaverde et al., 2014]. However, service discovery solutions cannot guarantee that the required service will always be discovered. In the provided example (Fig. 3.1), consider the case when the

northeast camera in the traffic intersection fails. Although the southwest camera is still available, or the service discovery system may find other intersection cameras in the city, they cannot be used to detect the crowd in the northeast corner of the intersection. In addition, composite service adaptation and service discovery process may take time. During this process, the service is in the state of failure until it finds and binds the required service.

Service network tolerance to cascading failure is determined by the topology of the network and the interdependency between services. However, IoT application designers do not always have full control to decide the topology and the interdependency between services. In some simple IoT applications, e.g. a traffic intersection system, designers can design the network topology and provide some redundancy to improve cascading-failure tolerance. However, providing some redundancy is costly and the number of services could be limited. Analyzing the effect of network topology and dependency between services could give more understanding on how to significantly improve the service network tolerance to cascading failure.

3.3 Service Network

We define service network by presenting the model of service network as a directed network where a node represents a service and a link represents dependency of one service on the other. Next, we address the properties of service network that are relevant to the tolerance against cascading failure. Finally, we address two real service networks to support our analysis on cascading failure.

3.3.1 Service Network Model

A massive number of interdependent services in the future Internet can be modeled as a service network. We model a service network according to the dependency between services in SOC, where the composite services depend on their component services. Formally, a service network is defined as $N = (S, E)$ where the set of services $S = A \cup C$ consists of atomic services A and composite services C. We define the set of links $E = L \cup R$, where L is the set of dependency links and R is the set of alternate links.

The model is illustrated in Fig. 3.2, where a service is represented as a circle, and the dependency of a composite service on a component service is represented as an outgoing link originated from the composite service. First, we explain the dependency links as follows. The set of dependency links that belongs to s_3 is defined as $L_3 = \{l_{3,1}, l_{3,2}, l_{3,3}, l_{3,4}\}$. To quantify the number of component services that will be executed by composite service s_i, we define s_i *degree of dependency* as $dep_i = |L_i|$. Therefore, s_3 degree of dependency is $dep_3 = |L_3| = 4$.

The existence of a substitutable service is illustrated by splitting a dependency link into two or more alternate links, for example in $l_{3,1}$. The set of alternate links on $l_{3,1}$ is $R_{3,1} = \{r^1_{3,1}\, r^2_{3,1}\}$. We define $l_{i,j}$ *degree of alternative* as $alt_{i,j} = |R_{i,j} - 1|$. This represents the number of substitutable services that can substitute another service in case of failure. Therefore, $l_{3,1}$ degree of alternative is $alt_{3,1} = |R_{3,1} - 1| = 1$. If service s_4 (West Vehicle Sensor) is unexecutable, the composite service s_3 is still able to perform its function by invoking service s_5 (Image Processor). We call this type of dependency as *weak dependency*. This is the case when Image Processor counts the num-

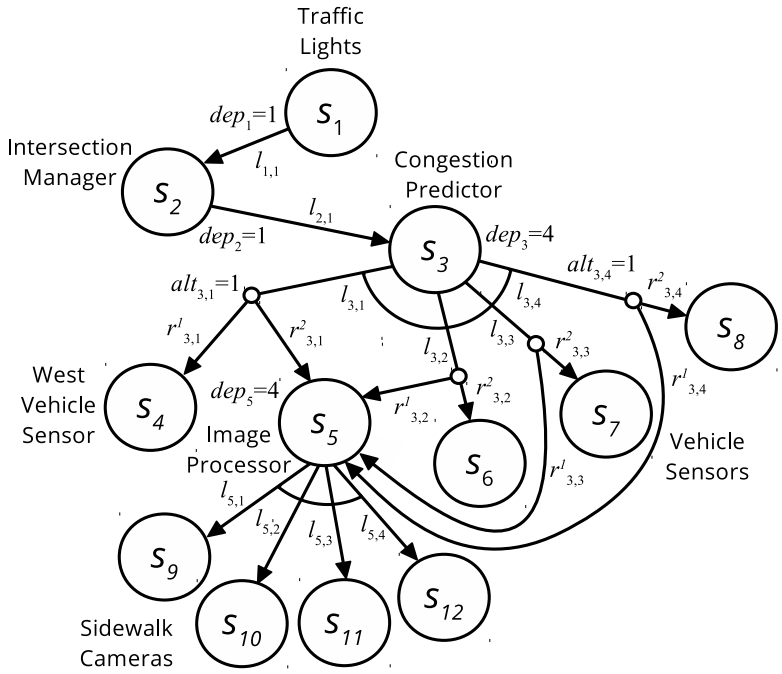


Figure 3.2: A service network represents the dependency of some components in the traffic intersection system. Each component is represented as a service node (circle). Dependency between two services is illustrated by a link (arrow). An arc connecting two or more links indicates that the service depends on all of the services pointed by the links, whereas a small circle indicates that the service requires either service connected by the links.

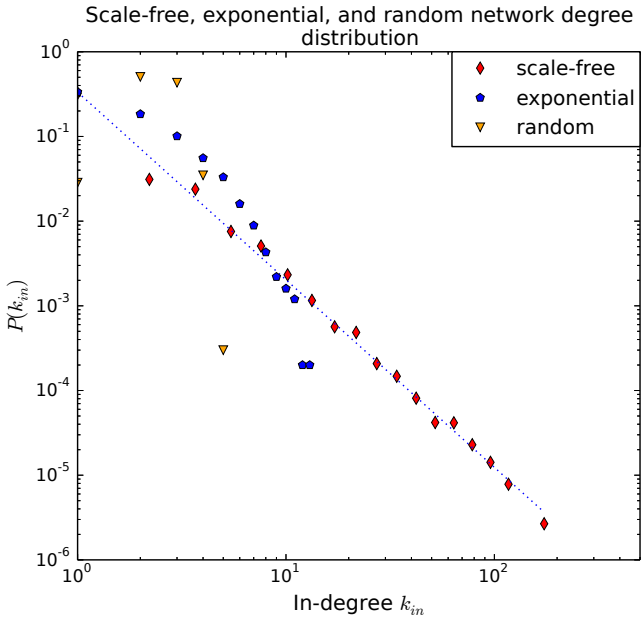


Figure 3.3: In-degree distribution of scale-free, exponential, and random service networks.

ber of vehicles from the images that sent by a nearby sidewalk camera. By contrast, a dependency on a component service is called *strong dependency* when there is no substitutable service available for the component service, such as in $l_{2,1}$. When Congestion Predictor (s_3) fails, Intersection Manager (s_2) may also unable to set the traffic lights timing accurately.

3.3.2 Service Network Properties

We present some relevant properties that contribute to cascading-failure tolerance, which are network topology, the degree of interdependency between services, the depth of composite services, and the proportion of composite services in the network.

Network Topology

Related work has found that network topology affects tolerance to cascading failure [Crucitti et al., 2004]. The topology of complex and large-scale networks is usually classified according to their degree distribution [Dorogovtsev and Mendes, 2002]. For directed networks, including service networks, there are in-degree distribution and out-degree distribution that can be obtained by calculating the distribution of incoming links and outgoing links, respectively. The out-degree of a service is the number of component services that the service depends to. In Fig. 3.2, the out-degree of composite service s_7 is 5. The in-degree of a service is the number of composite services that require this service as one of its components.

Many real networks, including existing service networks, are growing networks in that new nodes are added to the network over time. A growing network that employs preferential attachment, i.e. when new nodes prefer to connect to the existing nodes having high degree, will become a scale-free network. This type of network is recognized by its power-law degree distribution (Fig. 3.3), which formally defined as

$$P(k) \sim k^{-\gamma} \tag{3.1}$$

where $P(k)$ is the probability that a node in the network has k connections, and degree exponent γ is typically in the interval $2 < \gamma < 3$. By contrast, the network degree distribution will become exponential when the connections are made randomly, and thus it is classified as exponential network [Barabási et al., 1999].

Research on scale-free networks has gained more interest since many real networks were identified as scale-free networks, such as the Internet [Faloutsos et al., 1999], E-mail network [Ebel et al., 2002], WWW [Barabási et al., 2000], chapter co-author network [Barabási et al., 2002], and cellular and microbiological networks [Albert, 2005]. Service networks are also predicted to be scale-free since they are growing network and employ preferential attachment [Oh et al., 2008, Kil et al., 2009]. This assumption is also supported by the fact that real service networks, such as the ProgrammableWeb's API network [Huang et al., 2012] and publicly available web services network [Feng et al., 2014], are indeed scale-free. The interest on scale-free topology is also motivated by its high tolerance against random failure despite its vulnerability to directed attack [Albert et al., 2000]. However, the tolerance of scale-free network on random failure does not take cascading failure into account. By contrast, research in power network shows that random topology has better tolerance than scale-free against horizontal cascading failure [Crucitti et al., 2004]. On the other hand, service network tolerance against vertical cascading failure has not been studied.

In addition to the growing network, we also address random network. Random network is assumed to have fixed node number and the connections between the nodes are made randomly [Erdős and Rényi, 1960]. The degree distribution of random networks follows Poisson distribution (Fig. 3.3). Long before it is possible to analyze complex networks computationally, networks were assumed to have random characteristic [Barabási and Albert, 1999]. However, random networks are still relevant to be addressed since some existing networks may also have a rather static number of nodes.

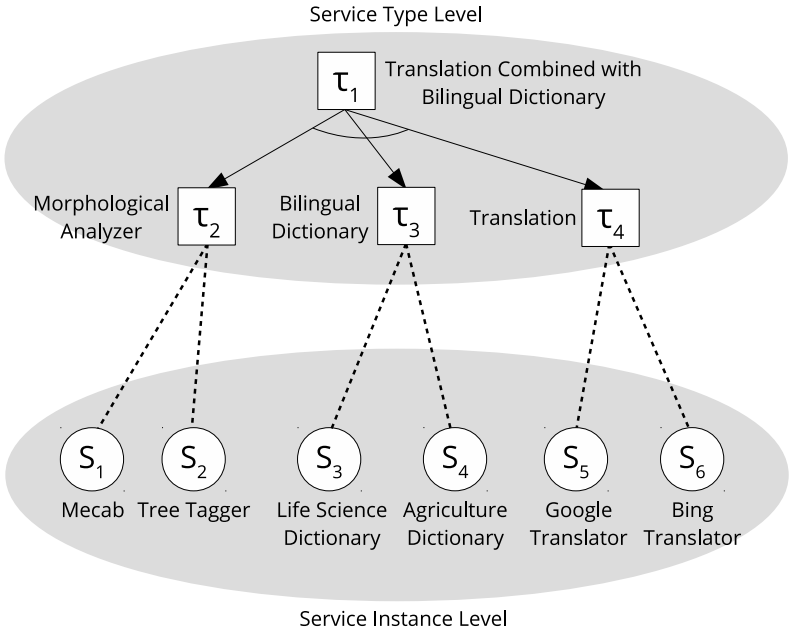


Figure 3.4: Service composition in The Language Grid is modeled into two abstraction level.

In this chapter, we analyze cascading-failure tolerance between scale-free, exponential, and random service networks. Another kind of network is a non-growing network that employs preferential attachment to connect a fixed number of nodes to each other. We exclude this type of network since it can be classified into either scale-free or random network. The degree distribution of such a network initially follows power-law as in scale-free, but eventually becomes Gaussian distribution as the network grows larger [Barabási et al., 1999].

Degree of Interdependency between Services

In Section 3.3.1, we have addressed the *service degree of dependency* dep_i and the *service degree of alternative* $alt_{i,j}$ that reflects the number of required services and the number of substitutable services for a particular composite service s_i, respectively. Here, we define two similar metrics to quantify the interdependency of services for the whole service network. First, we define the *service network degree of dependency* $\langle dep \rangle$, or simply *degree of dependency* for convenience, as the average of dep_i for every composite service $s_i \in C$ in the network. Formally, it is defined as

$$\langle dep \rangle = \frac{1}{|C|} \sum_i dep_i \tag{3.2}$$

where $|C|$ is the number of composite services in the network. Second, we define the *service network degree of alternative* $\langle alt \rangle$, which can be obtained by taking the average of $alt_{i,j}$ for every dependency link $l_{i,j}$. Likewise, we will call this as *degree of alternative* for brevity. The formal definition is as follows

$$\langle alt \rangle = \frac{1}{|L|} \sum_{i,j} alt_{i,j} \tag{3.3}$$

where $|L|$ is the number of dependency links in the network.

Depth of Service Composition

A composite service can combine component services, atomic services, or both. In Fig. 3.2, the composite service s_7 combines three functionalities provided by five atomic services. When a composite service combines one or more composite services, the composition becomes nested. Within a

nested composite service, failure of a component service can cascade outwards to the outermost composite services. The deeper the composition level, the more composite services that potentially experience cascading failure. To measure the composition level of a service s_i, we define its depth as

$$depth_i = \begin{cases} \frac{1}{|S_i|} \sum_{j=0}^{|S_i|-1} [depth_j + 1], & s_i \text{ is a composite service} \\ 0, & \text{otherwise} \end{cases} \tag{3.4}$$

where $S_i \subset S$ is the set of component services belongs to s_i, $|S_i|$ is its cardinality, and $depth_j$ is the depth of a component service $s_j \in S_i$. To quantify the depth of all composite services in the network, we calculate the average of $depth_i$ for every composite service in the network as

$$\langle depth \rangle = \frac{1}{|C|} \sum_i depth_i \tag{3.5}$$

Proportion of Composite Services

The higher the number of composite services in the network, the more composite services that potentially experience cascading failure. The proportion of composite services in the network can be obtained as follows

$$c = \frac{|C|}{|S|} \tag{3.6}$$

In this chapter, we only analyze the effect of the first two properties on cascading failure, which are network topology and degree of interdependency. For the depth of service composition, we assume that the nested composi-

Table 3.1: The Language Grid (LG) and ProgrammableWeb (PW) service network properties

Property	LG	PW
Topology	Fit to neither scale-free nor exponential	Scale-free
Degree of dependency $\langle dep \rangle$	2.81	2.11
Degree of alternative $\langle alt \rangle$	10.92	0
Depth of composition $\langle depth \rangle$	1.29	1.00
Number of services $\|S\|$	138	19,366
Number of composite services $\|C\|$	21	7,513
Proportion of composite services	0.15	0.39

tion in the network is very rare, and thus $\langle depth \rangle = 1$. These assumptions are supported by our observation on publicly accessible service networks, whose depth is close to 1 (Table 3.1). As for the proportion of composite services, we also fix the value according to the same property of the ProgrammableWeb service network at $c = 0.4$. Two publicly accessible service networks are addressed in the following section.

3.3.3 Publicly Accessible Service Networks

To support our analysis on the tolerance of service network to cascading failure, two publicly accessible service networks are addressed. The Language Grid is a typical example of a service network, where users can add new services and combine existing services by means of service composition [Ishida et al., 2011]. In addition, we also address ProgrammableWeb API directory, which is an example of a larger service network consisting of more than 19,000 APIs and mashups. The services published in The

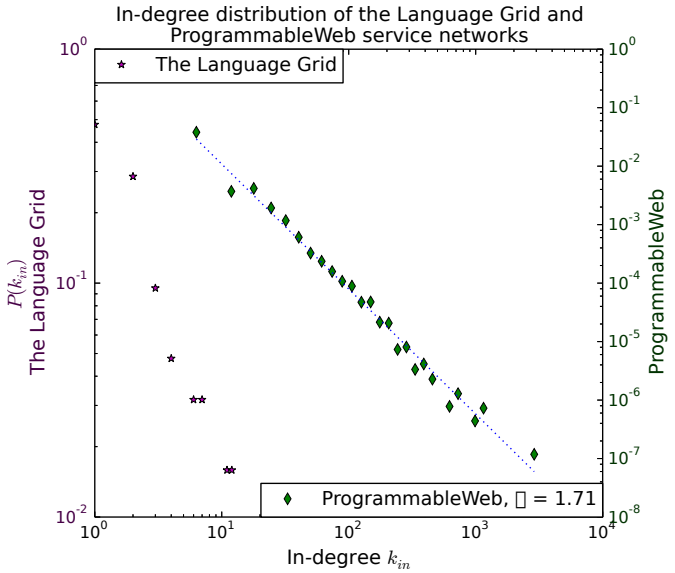


Figure 3.5: In-degree distribution of The Language Grid and ProgrammableWeb service networks.

Language Grid are language-related, while ProgrammableWeb lists more various kinds of APIs, such as mapping, social, searching, etc. To date, the number number of publicly accessible IoT-related services is still very limited. ProgrammableWeb lists around 300 out of 13,000 APIs categorized as IoT-related. However, IoT-related services and service accessible smart devices are expected to become more pervasive as research and initiatives in this direction gain more interest. The properties of these two publicly accessible service networks are listed in Table 3.1.

The Language Grid Service Network

The Language Grid is a service-oriented infrastructure that provides a way for the users to publish their language resources as services, and to com-

pose new services by combining the existing services using service composition [Ishida et al., 2011]. The dependency between composite services and their required component services in The Language Grid represents the typical service composition of a service-oriented environment. In Fig. 3.4, The Language Grid service composition is modeled into two level of abstraction, which are service type level and service instance level.

Service types classify service instances based on their functionalities. Service instances, or we simply call services, belong to the same service type if they share the same functionalities. For example, Google Translator and Bing Translator services are instances of *Translation* service type. Services that belong to the same service type can substitute each other; they are considered as substitutable services. A composite service type is a combination of different service types. An example of a composite service type is *Translation Combined with Bilingual Dictionary* that, at runtime, combines the instances of *Morphological Analyzer*, *Bilingual Dictionary*, and *Translation*.

The Language Grid operation centers have been established in several countries, including Japan (Kyoto), Thailand (Bangkok), Indonesia (Jakarta), and China (Urumqi), which in total provide 188 services. The architecture of The Language Grid allows one to combine services across different operation centers [Murakami et al., 2012]. Here, we limit our analysis on 117 atomic services and 21 composite services provided by Kyoto Language Grid operation center. Even though the proportion of composite services is much less than the proportion of atomic service, they are frequently invoked, which is 47% of all invocations. This indicates that the composite services are vital for The Language Grid users.

The degree distribution of The Language Grid (Fig. 3.5) follows neither power-law nor exponential distribution. We suspect that this is because the relatively small size of the network. When the network is growing larger, we expect it to become a scale-free network as the other service networks [Oh et al., 2008, Kil et al., 2009, Huang et al., 2012, Feng et al., 2014]. This assumption is also supported by the fact that The Language Grid employs preferential attachment in that the composite services are more likely to combine widely used and popular services than less utilized and non-popular services.

Interdependency among The Language Grid services is low. Even though each composite service combines, on average, three component services at $\langle dep \rangle = 2.81$, each component service can be substituted by roughly eleven services at $\langle alt \rangle = 10.92$. As for the depth of service composition, we found that The Language Grid service network has shallow composition at $\langle depth \rangle = 1.29$, which indicates that nested composite services are rare.

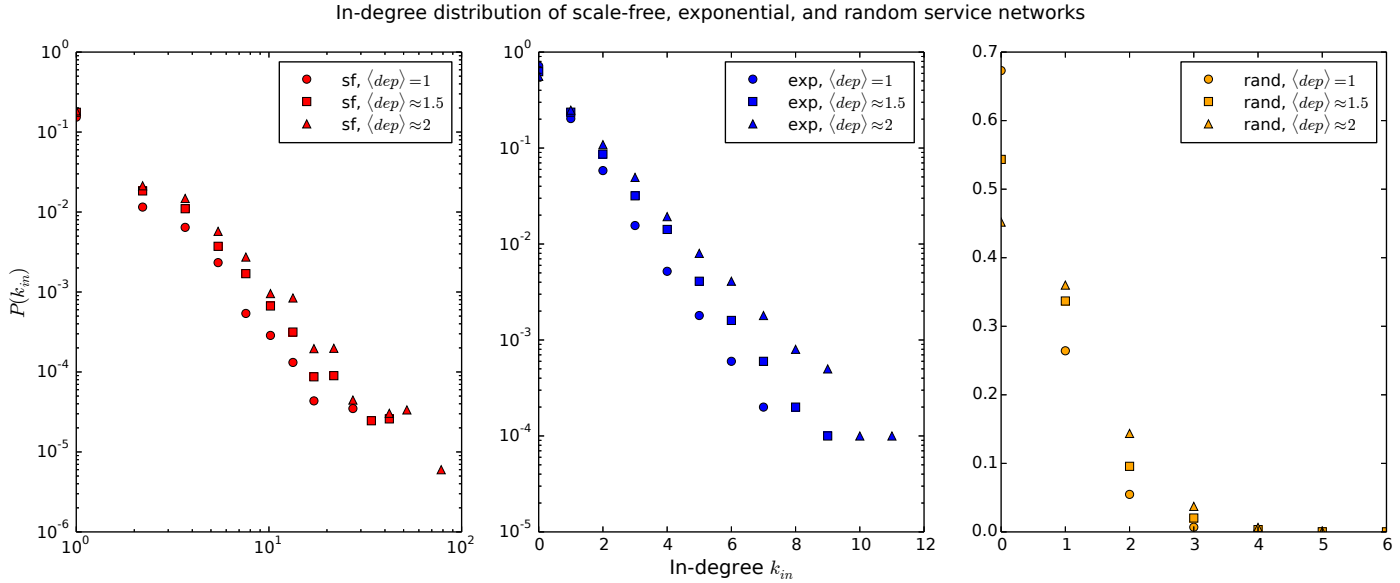


Figure 3.6: In-degree distribution of scale-free (left), exponential (center), and random (right) service networks.

ProgrammableWeb API Network

ProgrammableWeb provides information and directory of publicly available application programming interface (API) and mashups. ProgrammableWeb API directory can be used by interested users and developers to browse and retrieve information about the registered APIs and mashups. The API directory is also an open directory that it allows developers to publish their own APIs and mashups. A mashup, which combines two or more APIs, is analogous to a composite service that combines other services. Therefore, the collection of APIs and mashups also forms a service network where the service nodes represent APIs and mashups, and the links represent the dependency of the mashups on their APIs.

We analyzed 11,853 APIs and 7,513 mashups retrieved on October 15, 2014, and found that, on average, each mashup combines two APIs at $\langle dep \rangle = 2.11$. The directory and retrievable data do not include any information that can be used to identify alternate links or services. Therefore, we assume that the ProgrammableWeb service network has no alternate link, and thus $\langle alt \rangle = 0$. We also calculated the depth of the mashups and found very few nested mashups at $\langle depth \rangle = 1.00$. As for the network topology, the scale-free characteristic of the network is confirmed by its in-degree distribution that fits with power-law at degree exponent $\gamma = 1.71$ (Fig. 3.5). The scale-free characteristic is consistent with a related work that analyzes around 13,000 APIs obtained from ProgrammableWeb in November 2011 [Huang et al., 2012]. These two results show that the service network maintains its scale-free topology as the network grows.

3.4 Service Network Generation

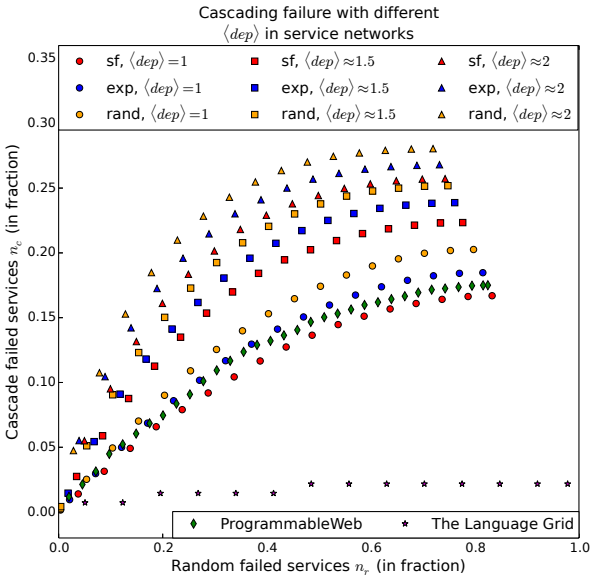
We present the algorithm to generate service networks, on which cascading-failure simulations are performed. The discussion is divided into two groups of service networks: *artificial service network* and *real service network*. The artificial service networks are generated from scratch for different topology, $\langle dep \rangle$, and $\langle alt \rangle$. The real service networks are generated based on real data obtained from The Language Grid services and ProgrammableWeb APIs.

3.4.1 Artificial Service Network

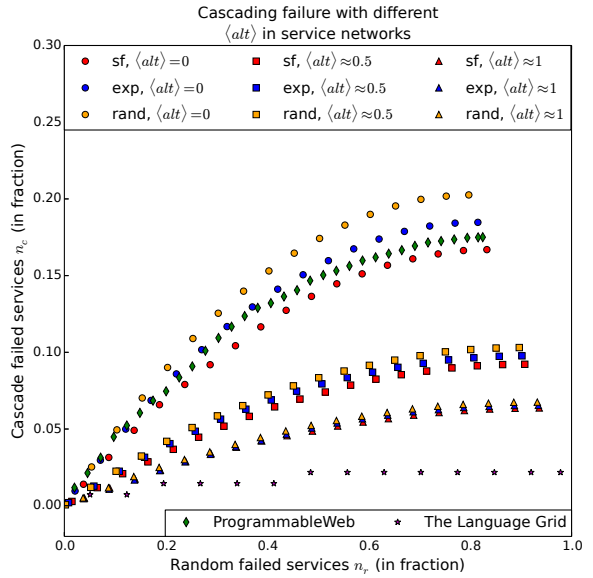
By "artificial" we mean the networks are computationally generated from scratch instead of constructed from an existing service network. The artificial service networks to be generated are of scale-free, exponential, and random topologies. Having service networks with these three topologies allows us to analyze which topology has better tolerance to cascading failure.

Growing Service Network

Since both scale-free network and exponential network are growing network, we use the same algorithm to generate these kinds of service networks. The difference between the two is the way of choose which existing services are connected to. Scale-free network employs preferential attachment in choosing services to be connected, whereas exponential network chooses them randomly. The algorithm to generate scale-free service network is based on the scale-free web graph model, which is also a directed



(a)



(b)

Figure 3.7: Cascading failure in service networks with different degree of dependency $\langle dep \rangle$ (a) and different degree of alternative $\langle alt \rangle$ (b).

network model [Bollobás et al., 2003]. The work also formally proves that the network will eventually exhibit scale-free characteristic as the network grows.

The preferential attachment probability function to generate scale-free service network is defined as [Chen and Shi, 2004]

$$\Pi(k_i^{in}, \alpha) = \frac{k_i^{in} + \alpha}{\Sigma_j (k_j^{in} + \alpha)} \tag{3.7}$$

where k_i^{in} is the in-degree of service s_i, and j is the index for all services in the network. Function $\Pi(k_i^{in}, \alpha)$ calculates the likelihood of choose service s_i having k_i^{in} degree, which implies that services with higher in-degree have more likelihood of being chosen. In exponential networks, by contrast, service selection is random, i.e. in-degree is ignored. The probability function is different with the basic scale-free probability function in that it includes an initial attractiveness parameter α. The initial attractiveness parameter yields the attractiveness value for isolated and young nodes [Dorogovtsev et al., 2000].

The required parameters for the growing service network algorithm are as follows:

- n_0 is the number of isolated services at the beginning of the simulation t_0
- $\Delta t \geq 0$ is the duration of the network generation
- $c \in [0, 1]$ is the expected proportion of composite services in the network

- δ is the expected degree of dependency $\langle dep \rangle$ such that $\delta \approx \langle dep \rangle$
- λ is the expected degree of alternative $\langle alt \rangle$ such that $\lambda \approx \langle alt \rangle$
- α is the initial attractiveness parameter

The algorithm works as follows. At t_0, initialize the network with n_0 number of isolated services. And then, for each timestep $t = t_1$ until $t = t_0 + \Delta t$, the following is performed:

1. Add a service s_i to the network.
2. With probability c, generate a random number dep_i in the interval $[1, \delta \times 2 - 1]$.
3. If dep_i is generated in step (2), create dep_i dependency links from s_i. The set of dependency links belong to s_i is $L_i = \{l_{i,0}, ..., l_{i,dep_i-1}\}$.
4. For each dependency link $l_{i,j} \in L_i$ created in step (3), generate $alt_{i,j}$, a random number in the interval $[0, \lambda \times 2]$. Perform one of these steps:
   (a) If $alt_{i,j}$ is zero, choose an existing service and connect the dependency link $l_{i,j}$ from s_i to the chosen service.
   (b) If otherwise, choose $alt_{i,j} + 1$ number of existing services. Split dependency link $l_{i,j}$ into $alt_{i,j} + 1$ number of alternate links and connect these alternate links to the chosen services.

The services are chosen using preferential attachment for scale-free service network, or in random manner for exponential service network. The connections will only be made between services that are currently not connected and not to self. In other words, the network assumes no multiple links and no self links. The first step in the algorithm represents the initialization of

a service network with some number of atomic services. Next, the service network grows by adding one service at a time. The addition of a service to the network represents the deployment of a new service. The probability c in step (2) controls the number of composite services in the network, and thus determines whether the new service is a composite service or an atomic service. If one or more dependency links are created, the service becomes composite service, otherwise it becomes atomic service.

Figure 3.6 (left) shows the in-degree distribution of some scale-free service networks that generated using the algorithm. The scale-free service networks are generated with the parameters $n_0 = 10$, $\Delta t = 10,000$, $c = 0.4$, $\alpha = 1$, $\lambda = 0$, and different δ to obtain different degree of dependency $\langle dep \rangle$. The in-degree distribution of the generated networks follows power-law. The generated exponential service networks also show exponential in-degree distribution as expected (Fig. 3.6, center). It is also generated with the same parameters as the scale-free service networks but excluding the α. These results confirm that the algorithm generates scale-free and exponential networks properly.

Random Service Network

To generate random service network, we use the same algorithm with the one to generate exponential service network, where the connections are made randomly without preferential attachment. However, as a non-growing network, generating random networks does not require node addition. Therefore, the algorithm uses all the parameters that are also used to generate exponential service network, except that Δt is ignored. Since

the number of nodes is fixed, the number of initial nodes n_0 is also the expected size of the network, i.e. $n_0 = |S|$. Specifically, the algorithm works as follows:

1. Initialize the network with n_0 number of isolated services.
2. For each service $s_i \in S$, perform the steps (2), (3), and (4) as generating exponential service networks according to the algorithm in Section 3.4.1.

In Fig. 3.6 (right), the in-degree distribution of some generated random service networks is shown. The networks are generated with $n_0 = 10,010$, $c = 0.4$, $\lambda = 0$, $\alpha = 1$, and different δ to obtain different $\langle dep \rangle$. As expected, the in-degree exhibits Poisson distribution.

3.4.2 Real Service Networks

In addition to the artificial service networks, we also generate service networks based on The Language Grid services and ProgrammableWeb APIs. The generation of service networks from these real service networks has two purposes. First, to identify the typical topology of real service networks. Second, to confirm the validity of the algorithm to generate artificial service network. Simulations of cascading failure are also performed on these service networks.

As illustrated in Fig. 3.4, The Language Grid services are classified into service types according to the functionalities they provide. The Language Grid service network is generated by representing service instances as atomic service nodes, and composite service types as composite service nodes. The

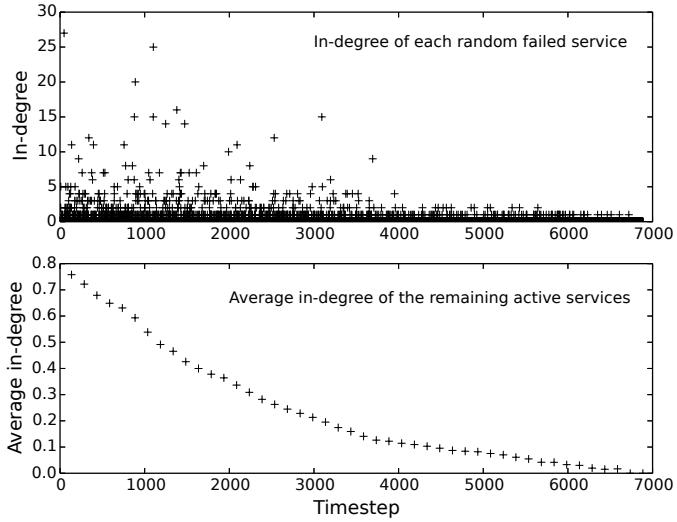


Figure 3.8: The in-degree of each random failed service (top) and the average in-degree of the remaining active services (bottom) during cascading-failure simulation.

links from composite service nodes to their component service nodes are created based on the service types they combined. If a combined service type is realized by only one service instance, a dependency link is created connecting the composite service node to the atomic service node. If there are more than one service instance that can provide the functionality for the service type, alternate links are created to connect the composite service node to these atomic service nodes. Nested composition is created when a composite service types combines one or more composite service types.

The generation of the ProgrammableWeb service network is more straightforward. The APIs and mashups are generated as atomic service nodes and composite service nodes, respectively. Dependency links are generated from these composite service nodes to their required components. As explained in Section 3.3.3, no alternate links is created due to the assumption

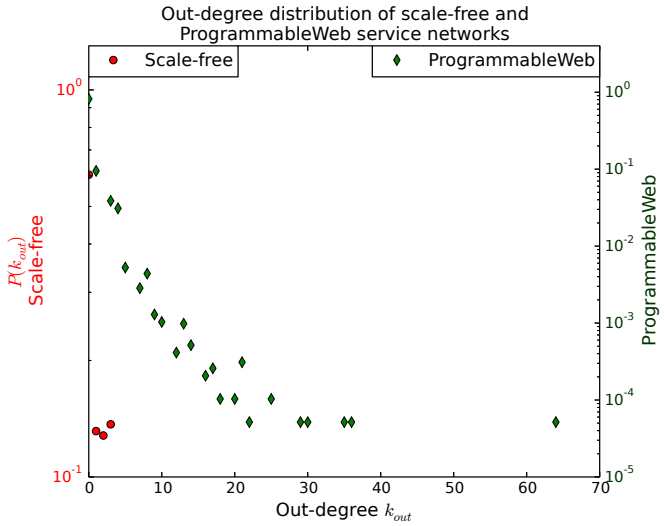


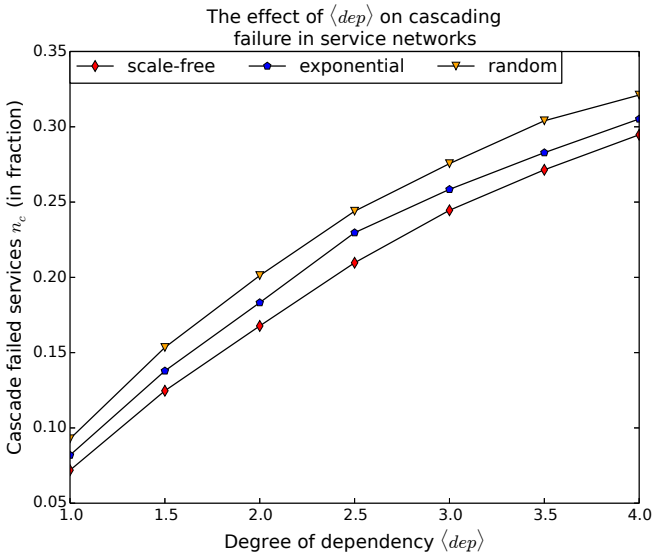
Figure 3.9: The out-degree distribution of scale-free and the ProgrammableWeb service networks.

that $\langle alt \rangle = 0$.

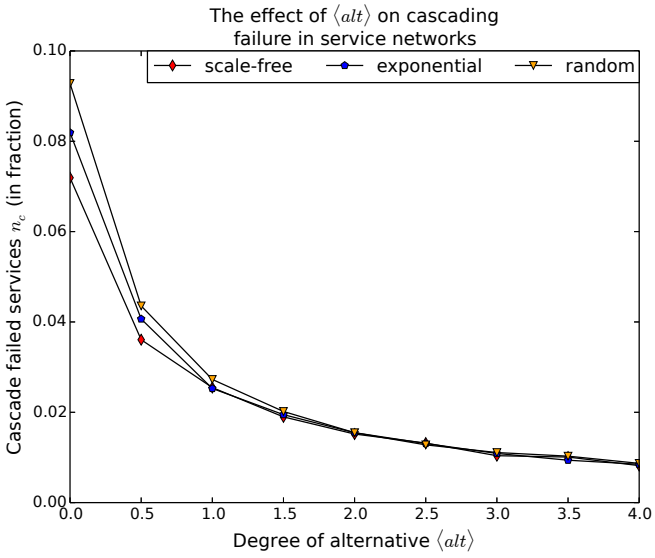
3.5 Cascading-Failure Simulation and Analysis

To simulate cascading failure, we generate service networks with different topology, $\langle dep \rangle$, and $\langle alt \rangle$ using the algorithm presented in Section 3.4. Five types of service networks are generated. The first three are scale-free, exponential, and random service networks which are the artificial service networks. The other two types are the real service networks generated from The Language Grid services and ProgrammableWeb APIs.

Growing service networks (scale-free and exponential service networks) are initialized with a small number of isolated services ($n_0 = 10$). The network



(a)



(b)

Figure 3.10: The effect of degree of dependency $\langle dep \rangle$ (a) and degree of alternative $\langle alt \rangle$ (b) on cascading failure in service networks.

grows by adding one service at a time for the duration $\Delta t = 10,000$. Random service networks are generated with $n_0 = 10,010$, which is also the expected size of the networks. The proportion of composite services is set at $c = 0.4$ according to the same property of the ProgrammableWeb service network. For scale-free networks, initial attractiveness parameter is set at $\alpha = 1$. These networks are also generated for different $\langle dep \rangle$ and $\langle alt \rangle$ by setting different value for δ and λ parameters, respectively. For the generation of real service networks, number of services and the connections between them are given.

Cascading-failure simulation starts after a service network is generated. At each timestep, random failure is performed to trigger cascading failure by randomly choosing a service to be deactivated. We call such a service as a random failed service. The failure cascades to all other services that strongly depend on the random failed service, and subsequently triggers a series of other failures. The services that experience failure because of another failure is called cascade failed services. For convenience, let n_r be the number of random failed services, and n_c be the number of cascade failed services. Random failure is repeated until the network breaks down completely, i.e. when $n_r + n_c = |S|$. We perform the cascading-failure simulations multiple times for each combination of $\langle dep \rangle$ and $\langle alt \rangle$ considering the stochastic nature of the algorithm.

3.5.1 The Effect of Network Topology on Cascading Failure

Figure 3.7a shows the fraction of cascade failed services over the fraction of random failed services. The degree of alternative is set at $\langle alt \rangle = 0$, while degree of dependency $\langle dep \rangle$ varies. In Fig. 3.7b, we set $\langle dep \rangle = 1$, which is the lowest possible value to have different $\langle alt \rangle$ (Fig. 3.7b). In these two figures, different network topologies are illustrated with different colors, while different degrees of interdependency use different markers.

The variation of tolerance between different network topologies is a result of their in-degree distribution. As illustrated in Fig. 3.3, both scale-free and exponential networks have very few nodes with high in-degree, while most of the nodes have low in-degree. The number of high in-degree nodes (hubs) in a scale-free network is less than in an exponential network of the same size (number of nodes) and the same number of links. As a result, hubs in scale-free networks are less likely to experience random failure. Similarly, the tolerance of exponential network is better than random network because high-degree nodes in random network are more frequent than in an exponential network of the same size and the same number of links.

3.5.2 In-degree Evolution during Network Failure

Despite the tolerance variation between different topologies, Fig. 3.7 shows that the increase of cascade failed services n_c is higher during the early stage of random failure (lower number of random failed nodes) than the later stage (higher number of random failed nodes). At first, this result may seem to

contradict the nature of scale-free and exponential networks, in which low-degree nodes are more likely to experience random failure than high-degree nodes [Albert et al., 2000]. However, as the services fail, the links connected to them are also removed from the network. Therefore, each random failure and the cascading failure that follows reduce the in-degree of the remaining active services. Consequently, this decreases the number of cascade failed services that inflicted by the next random failure. Figure 3.8 (top) shows that random failed services chosen at the early stage have generally higher in-degree than those at the later stage. The in-degree evolution during network failure is shown Fig. 3.8 (bottom), where the average in-degree of the active services in the network decreases over time.

3.5.3 The Effect of Degree of Dependency on Cascading Failure

In Fig. 3.7a, we have seen that cascade failed services are more frequent on service networks with higher $\langle dep \rangle$, because they have stronger inter-dependency between the services. Cascading failure in the real service networks shows some peculiarities. The Language Grid degree of dependency at $\langle dep \rangle = 2.81$ does not inflict much cascading failure since it has a rather high number of substitutable services at $\langle alt \rangle = 10.92$. However, the ProgrammableWeb service network, which is also a scale-free network and has $\langle dep \rangle = 2.11$, shows different result with the generated scale-free service networks at $\langle dep \rangle \approx 2$ (red triangle). This is a result of the ProgrammableWeb out-degree distribution, which in this case is also the distribution of dep_i. Figure 3.9 shows that, in ProgrammableWeb, the proportion

of the higher out-degree is much lower than that of the scale-free network.

We present Fig. 3.10a to analyze further the effect of $\langle dep \rangle$ on cascading failure. The figure shows the proportion of cascade failed services n_c over $\langle dep \rangle$ when 20% of the services in the network experience random failure. We set the degree of alternative at $\langle alt \rangle = 0$. The graphs show that the increase of n_c is somewhat linear to $\langle dep \rangle$. The linear relationship is a result of shallow depth of service composition in the network. The result suggests that improving the tolerance to cascading failure can be achieved by reducing the number of required component services from each composite service. However, the component services to be combined within a composite service are usually defined during the design of the composition. Therefore, reducing the number of required component services may not always be possible.

3.5.4 The Effect of Degree of Alternative on Cascading Failure

Figure 3.7b shows that the higher the $\langle alt \rangle$, the more substitutable services that lower n_c (e.g. at $\langle alt \rangle \approx 1$). Decreasing the interdependency between services improves the tolerance against cascading failure. As $\langle alt \rangle$ increases, the gap of n_c between different topologies decreases, and thus the effect of network topology on cascading failure becomes less significant. To analyze this further, we present Fig. 3.10b. The figure shows the proportion of cascade failed services n_c when 20% of the services in the network experience random failure and degree of dependency is set at $\langle dep \rangle = 1$, which is the lowest possible value to vary $\langle alt \rangle$. The graphs show that $\langle alt \rangle$ is in-

versely proportional to n_c. The result suggests that increasing $\langle alt \rangle$ could improve the network tolerance to cascading failure. However, this would be significant only when $\langle alt \rangle$ is currently low. This is shown in the graphs where, for example, n_c significantly drops in the interval $0 \leq \langle alt \rangle \leq 2$. The higher the $\langle alt \rangle$, the drop of n_c gets less significant, e.g. at $\langle alt \rangle > 2$.

In conclusion, adding more substitutable services to a service network potentially improves the network tolerance to cascading failure only when the degree of alternative is currently low. Providing many substitutable services is not necessary since it may be costly and not always significantly improve the tolerance of the service network.

3.6 Summary

The future Internet will be populated with a massive number of services interacting with each other. These interacting services are interdependent, and thus form a large-scale service network. Cascading failure in a service network occurs when the failure of one service propagates to the failure of its dependent services. In this chapter, we analyzed service network tolerance to cascading failure, which has not been widely studied.

The goal of this research is to investigate: (i) the effect of the dependency between services to cascading failure and (ii) cascading-failure tolerance in different network topology. We generated scale-free, exponential, and random service networks with different degree of interdependency between services, and performed random failure simulation to analyze cascading-failure tolerance. To support our analysis, we also included publicly ac-

cessible service networks generated from The Language Grid services and ProgrammableWeb API network.

The contribution of this chapter is some important findings as follows:

1. The effect of network topology on tolerance is more significant on lower *degree of alternative*, i.e. where the average number of substitutable services for each required component service is low.

2. The number of cascade failed services, i.e. the nodes experiencing cascading failure, is inversely proportional to the degree of alternative.

3. The number of cascade failed services is somewhat linear to the *degree of dependency*, i.e. the average number of component services.

4. Even though services with high in-degree is very less likely to experience random failure, the number of cascade failed services inflicted by each random failure decreases over time as a result of in-degree evolution during the network failure.

5. Scale-free topology has better tolerance to cascading failure, followed by exponential and random topology. This is contrast with the load-based cascading-failure tolerance in power networks, where random topology has better tolerance than scale-free [Crucitti et al., 2004].

These findings would be useful for designing service network and composite services to have better tolerance against cascading failure. We also suggest that having few substitutable services for each required service would be sufficient to ensure the tolerance. On the other hand, when the number of substitutable services is already high, adding more substitutable service may not significantly improve the tolerance.

Chapter 4

Role Modeling Method for Designing Functional Flexibility

4.1 Background

As it has been addressed in Chapter 3, dependencies between IoT services potentially cause cascading failure. IoT services experience cascading failure when failure of one or more services disrupt the other dependent services. The failure of one or more IoT service may occur due to the mobility of smart objects. Some smart objects may be mobile, such as mobile devices, wearable devices, or smart devices embedded to vehicles. Such smart objects may join (leave) to (from) different networks while they are moving across different locations. Consequently, connection with the other required smart objects may become lost. Failure may also occur due to the distributed nature of IoT services. IoT services are provided by different smart objects

and applications that are developed and maintained independently by different companies. When an IoT service is modified, the other dependent IoT services that require its functionality may not be aware of the modification. Consequently, these dependent IoT services may not be able to interact with the modified IoT service as before.

The above failure situations clearly show that IoT services should be designed with self-organization capability to respond against changes and failures. In this thesis, the self-organization capability is referred to as flexibility. To design flexibility in IoT services, the system is modeled as MAS where each service is represented as an agent. Two kinds of flexibility are introduced: *failover flexibility* and *functional flexibility*. The former has been addressed in Chapter 3, which is the capability of an IoT service to perform failover actions by switching among substitutable services upon internal or external failures. In this chapter, functional flexibility is defined in terms of MAS as agent capability to respond against external (in the environment) or internal (in the MAS) changes and failures that require the agent to adjust their behavior to perform different actions.

This chapter is motivated by the lack of suitable features from existing MAS engineering methodologies to design agent flexibility. To allow the designers to easily design agent flexibility, the modeling method should provide a feature to separate the design of agent behaviors and agent flexibility[1]. The former defines the actions that can be performed by the agents, whereas the latter defines which actions that can (or cannot) be performed in which situations. Most of existing MAS engineering methodologies tend to mix the two design activities. The separation provides convenience way for the

[1]Agent flexibility is also called agent behavior adaptation.

designers to design agent behavior adaptation and improves maintainability. To provide such feature, a high-level notion is required to represent agent behavior. This chapter proposes role-modeling method, which uses role notion to represent agent behavior such that agents can change their behavior at runtime by changing their roles. Role notion is a natural and widely used concept to represent agent behavior.

The discussion in this chapter is organized as follows. First, related work is addressed in Section 4.2. Next, the proposed modeling method is explained by addressing the metamodel and the modeling activities (Section 4.3). A case study is provided to demonstrate the applicability of the modeling method (Section 4.4). Finally, the chapter is concluded by addressing the contributions in Section 4.5.

4.2 Role Modeling in MAS Engineering Methodologies

The well-known MAS engineering methodologies are, among others, Gaia [Wooldridge et al., 2000], SODA [Omicini, 2001], ADELFE [Bernon et al., 2003], Prometheus [Padgham and Winikoff, 2003b], INGENIAS [Pavón and Gómez-Sanz, 2003], Tropos [Bresciani et al., 2004], PASSI [Cossentino et al., 2005], ASEME [Spanoudakis and Moraitis, 2011] and O-MaSE [DeLoach and García-Ojeda, 2010]. Table 4.1 compares the supported features to develop self-organizing MAS between these methodologies, which partially adopted from an extant work [Tran and Low, 2005]. *Autonomy* is the ability of the engineering methodology to design au-

Table 4.1: Comparison of existing MAS engineering methodologies

Methodology	Characteristic			
	Autonomy	Decentralization	Organizational Adaptation	Openness
Gaia	Y	Y	-	Y
SODA	Y	Y	-	-
ADELFE	Y	Y	Y*	Y
Prometheus	Y	Y	-	-
INGENIAS	Y	Y	-	-
Tropos	Y	Y	-	-
PASSI	Y	Y	-	-
ASEME	Y	Y	-	-
O-MaSE	Y	Y	Y**	Y

The table is partially adopted from an extant comparison of MAS engineering methodologies [Tran and Low, 2005]

'Y': the MAS engineering methodology possibly supports the corresponding characteristic

'-': the methodology does not support the characteristic or does not clearly specify any modeling methods to support the characteristic

*ADELFE requires MAS designers to model agents as cooperative agents and to define all non-cooperative situations (fails) that might occur, and thus not suitable when failure situations are hard to be predefined.

**Even though O-MaSE supports decentralization and adaptation, its structural adaptation mechanism is performed by calculating the best combination between agents, roles and goal set. This approach requires high computation resource and is somewhat centralized due to the use global knowledge.

tonomous agents. *Decentralization* criterion evaluates the capability of the methodology to design MAS that have no centralized control. Since these aspects are the core features of MAS, all of the aforementioned methodologies support them. However, self-organizing MAS should also have the capability to organize themselves at runtime to respond against changes (*organizational adaptation*), and to allow agents joining and leaving from the systems (*openness*). Most of the methodologies do not completely cover these two features, except few of them such as ADELFE [Bernon et al., 2003] and O-MaSE [DeLoach and García-Ojeda, 2010]. Despite the advantages of these two methodologies, they also do not specifically target self-organizing systems, and thus have limitations to design behavior adaptation in such systems.

4.2.1 Role Notion in MAS Engineering Methodologies

Despite a number of definitions of role in social study, the theory of role is believed to be originated from a theatrical metaphor of social life. A role is the expected characteristic behavior of an actor who plays a part based on a script. In social context, the concept of role consists of three components: (i) pattern and characteristic of social behaviors, (ii) parts or identities that are assumed by social participants and (iii) scripts or expectations for behaviors [Biddle, 1986]. Role and role modeling have also been introduced in object oriented software engineering [Kendall, 1999]. Role models identify and describe the interactions of objects with each other in terms of roles. A role represents a position characterized by a set of responsibilities to be assigned to objects [Kendall, 2001].

Role notion has been used to represent behavior and functionalities of systems in different domains, such as sensor networks [Kochhal et al., 2003], distributed and embedded systems [Ren et al., 2007, del Val et al., 2014], smart device systems [Kawsar et al., 2008] and service-oriented architecture [Rouvoy et al., 2009]. In MAS, a role usually represents a set of functionalities, a position of duty or an aggregation of behaviors to be played by agents. Most of the well-known methodologies use role notion except Prometheus and ADELFE. The former prefers to use another term since role notion is defined differently between different methodologies [Padgham and Winikoff, 2003b]. In the latter, role notion is intentionally excluded to allow the designers to focus on designing cooperative agents [Bernon et al., 2005]. The concept of role is usually used together with the other related notions, such as group and agent class. The interplay between agent, group and role notions has been proposed in AGR [Ferber and Gutknecht, 1998] and has influenced the existing MAS engineering methodologies to date.

4.2.2 Role Modeling Process

Role modeling activities in each methodology are conducted based on the engineering phases of the methodology. Most of the methodologies perform role modeling during analysis and design phases, while some of the rest involve requirement phase. In spite of these differences, role modeling process can be generalized into four activities: (i) *identification*, (ii) *elaboration*, (iii) *interaction design* and (iv) *assignment*. The first activity identifies the roles that should exist to represent the behaviors of the agents. MAS designers identify roles from scratch, or derived them from the results of

Table 4.2: The role modeling process

Phase	Activity	Artefacts
Analysis	Identifying roles	List of roles
	Elaborating the roles in terms of capabilities	Role capabilities
Design	Designing relations between the roles	Role-relationship model (RRM)
	Designing interactions between the roles	Role-interaction model (RIM)
	Assigning roles to agent classes	Role-enactment model (REM)
	Designing behavior adaptation	Role-transition model (RTM)

the previous modeling activities. The next activity, *elaboration*, defines the roles in more detailed in terms of their properties and some necessary description. The interaction between the agents, between agents and the other entities, and with their environment, are designed in *interaction design* activity. In the final activity, the roles are assigned to agent classes to define which agents will play which roles at runtime. The core activity to design behavior adaptation is performed in this final step that will be explained further in the following. Gaia, SODA and PASSI perform similar role modeling process, while the others perform in different order or combine several activities.

4.2.3 Behavior Adaptation Design

Behavior adaptation using roles can be realized by allowing the agents to change roles at runtime. In some of the methodologies, such as O-MaSE and ASEME, an agent can play multiple roles concurrently or sequentially change from one role to another. Another methodology, Gaia, has also addressed this idea [Zambonelli et al., 2003], but the feature has not been fully

supported. The other methodologies, including Tropos, INGENIAS and PASSI, also allow an agent to be associated with multiple roles. However, SODA does not clearly specify this feature.

Even though most of the methodologies support behavior adaptation by changing roles, only some that clearly address how MAS reorganize themselves at runtime. First, in ADELFE, reorganization is made possible by designing MAS as a collection of cooperative agents which requires the designers to define all possible non-cooperative situations (exceptions). This approach may not always be suitable for self-organizing systems that expected to be situated in dynamic and uncertain environments. Self-organizing systems may have many possible failure situations, some of which may not be predicted at design time. Second, the organizational adaptation in O-MaSE, based on OMACS framework [DeLoach, 2009], is performed by calculating the best combination of roles, agents and the current goal set. This approach is somewhat centralized and requires the agents to use global knowledge, and thus it is also not suitable for self-organizing systems that tend to be decentralized.

The existing modeling method that addresses the dynamic of agent behavior is behavior graph [Weyns et al., 2005]. Dynamic behavior of an agent is modeled as a transition graph, in which agents perform transition from one role to another. A role consists of actions to be performed by the agents. Despite its straightforward way to model dynamic behavior of agents, the modeling method limits each agent to play only one role at a time.

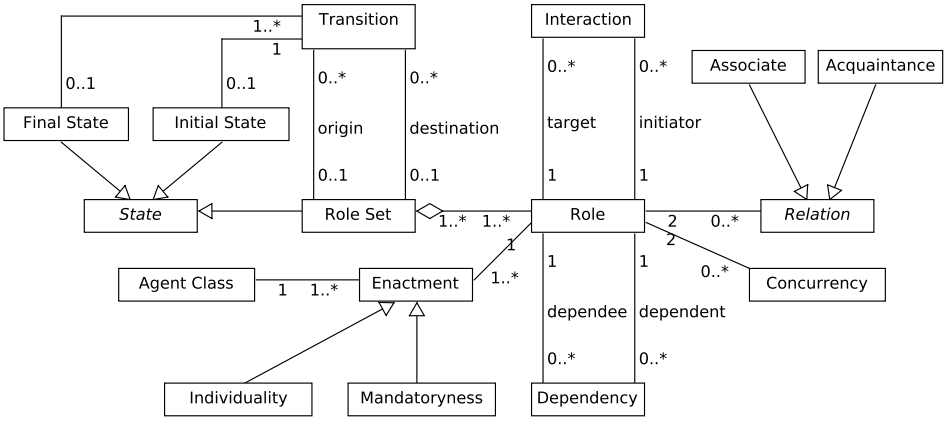


Figure 4.1: The metamodel of the role-modeling method

4.3 Designing Agent Behavior

The proposed role-modeling method consists of a series of modeling activities, most of which produce modeling artifacts as listed in Table 4.2. The notions used in the modeling process and their relation are illustrated in Fig. 4.1. Throughout the modeling process, designers identify roles, model association and interaction between the roles, and design how the roles are played by the agents at runtime. The modeling process starts with identifying and elaborating roles to define the required roles that represent agents behavior. Next, association between roles are defined in terms of relation, interaction and dependency between them. The relationship and interaction between the roles are designed in *role-relationship model* (RRM) and *role-interaction model* (RIM), respectively. Designers are also required to assign roles to agent classes to define which types of agents to play which roles using *role-enactment model* (REM). Finally, the agents behavior adaptation are designed as transition between role sets in *role-transition model* (RTM).

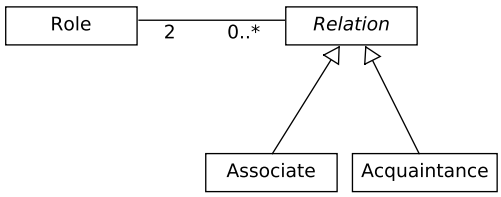


Figure 4.2: The metamodel of role-relationship model

These steps will be detailed further in the following sections.

4.3.1 Identifying and Elaborating Roles

In this first modeling activity, necessary roles are identified and defined. A role represents a set of behaviors to be played by the agents at runtime. Each role can be elaborated in terms of its capabilities. In implementation level, capabilities are realized as functions, methods, or services, that are provided or can be performed by the agents. In the later modeling activity (Section 4.3.3), roles will be assigned to agent classes to define the behavior of each agent class.

Roles can be defined in two ways: *top-down* and *bottom-up* approach. Both approaches assume that agent classes have been defined in advance. In top-down approach, the designers define one or more roles that will be assigned to agent classes, and then define the capabilities of each roles. In bottom-up approach, the capabilities are firstly defined, and then roles are defined to group these capabilities. Some examples of the output of this activity, as well as the other activities, are provided in the case study (Section 4.4).

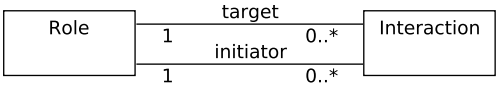


Figure 4.3: The metamodel of role-interaction model

4.3.2 Designing Relationship and Interaction Between Roles

The relationship between the roles are identified in RRM (Fig. 4.2). Each relation involves two roles, while a role may participate in one or more relations or does not participate in any relation at all. There are two kinds of relation: *associate* and *acquaintance* relations. An *associate* relation indicates that the two participating roles may be played concurrently by an agent. On the other hand, the absence of such relation between two roles prevents them to be played concurrently by an agent. An existence of an *acquaintance* relation between two roles means that the roles may be able to interact with each other.

Following the design of RRM, the possible interactions between the roles are modeled in RIM (Fig. 4.3). RIM describes the interaction that may occur between agents based on the roles they play. The interacting roles are only those that involve in an *acquaintance* relation, while the interactions themselves are derived from the capabilities of the roles. As shown in Fig. 4.3, an interaction involves two roles: the *initiator* role and the *target* role. The initiator role is the one that initiates the interaction by performing action to the target role. The target role can response the initiator by performing other actions.

4.3.3 Assigning Roles to Agent Classes

Roles need to be assigned to the agents to enable the agents playing them at runtime. However, assigning the roles directly to the agents requires the designers to identify the agents at design time. This is not suitable for self-organizing MAS that requires the system to be open. In an open MAS, the number of participating agents can change at runtime because agents can join and leave from the system. Therefore, in this activity, the roles are assigned to agent classes instead of directly to the agents.

Agent classes are the types of agents; agents of the same class have the same capabilities because they enact the same roles. However, they may play different roles at runtime, and thus show different behavior. The way agents play their roles at runtime is designed in RTM (Section 4.3.4). The configuration of roles in each agent class is also defined in this modeling activity, such as defining which roles should be played at all times and which roles can only be played concurrently with the other roles. Different agent classes can enact the same roles with different configurations. This is distinct from the other existing methodologies, such as O-MaSE [DeLoach and García-Ojeda, 2010], where the agent classes are solely identified by the assigned roles.

The metamodel of REM is illustrated in Fig. 4.4. There are three kinds of association: *enactment*, *dependency* and *concurrency*. When a role is assigned to an agent class, the association between them is called enactment. Enactment is specialized into *individuality* and *mandatoryness*. Individuality indicates that the role should not be played concurrently with any other roles. Mandatoryness requires the role to be always played by the instances.

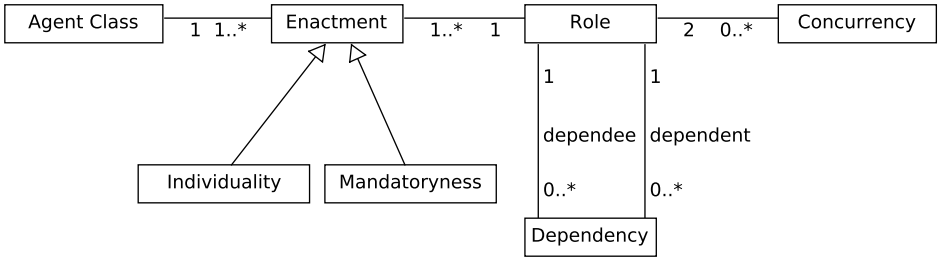


Figure 4.4: The metamodel of role-enactment model

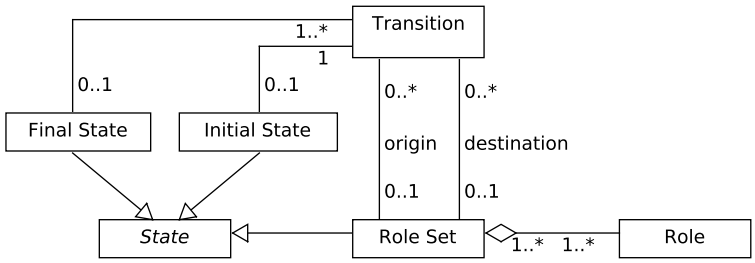


Figure 4.5: The metamodel of role-transition model

While *enactment* associates agents and roles, *dependency* and *concurrency* define association between two enacted roles. A dependency association involves a dependent role and a dependee role. It defines that the dependent role can only be played when the dependee role is also being played by the instances. A concurrency association between two roles indicates that the two roles can be played simultaneously.

4.3.4 Designing Behavior Adaptation

Behavior adaptation for each agent class is defined using RTM (Fig. 4.5). In this final modeling activity, behavior adaptation is designed as transitions between role sets. The model describes how agent behavior changes by

Table 4.3: Potential issues in playing multiple roles

Issues	Related Modeling	Description
Role conflict	RRM	Analyze potential conflict between the capabilities of associate roles Avoid the conflict by redesigning the RRM to omit the associate relation between the roles
Role overload	REM	Analyze potential overload in each concurrent roles Create more agent class to reduce the concurrency and the number of roles in each class
Role transition oscillation	RTM	Analyze the possibility of oscillation for each transition cycle Modify the RTM or split the agent class to reduce transition cycle
Common concurrency issues	RRM REM	Analyze possible concurrency issues for each associate roles (RRM) and concurrent roles (REM)

making transition from one state to another. A state represents a behavior of an instance identified by one or more roles. The designers define each role set based on the REM of the agent class. For example, a role which is defined as *mandatory* in the REM must be included in every role set in the corresponding RTM. The roles that can coexist in the same role set are also determined by their relations in the REM; only roles associated by *concurrency* or *dependency* association that can be a member of the same role set. The modeling of role transition in RTM is different with that of behavior graph [Weyns et al., 2005], since it allows multiple roles to be played by an agent at a time.

4.3.5 Identifying Potential Issues

The modeling activities are also useful to assist MAS designers to analyze potential issues that may occur because of playing multiple roles concurrently. Table 4.3 lists the issues and how the related modeling activities support the designers to identify and handle them. During the design of RRM, MAS designers should analyze the potential conflict between the capabilities of the associated roles, especially when two roles are associated as acquaintances. When potential conflict is detected, e.g. two capabilities having functionalities with opposite effect, the designers should redesign the RRM or handle the conflict in the later engineering phase. In designing REM, role overload may occur when an agent does not have enough resources to play multiple roles concurrently. In this case, the designers should consider to redesign the REM, or create more agent classes to reduce the number of roles in the agent class. Role transition oscillation is an unexpected behavior that occurs when an agent continuously making transition between two roles. This kind of issue may occur when an RTM has one or more transition cycles. To solve this issue, the designers can split the agent class or redesign the RTM. Finally, common concurrency issues, such as deadlock, livelock, starvation and interference, should also be anticipated by the designers as creating RRM and REM.

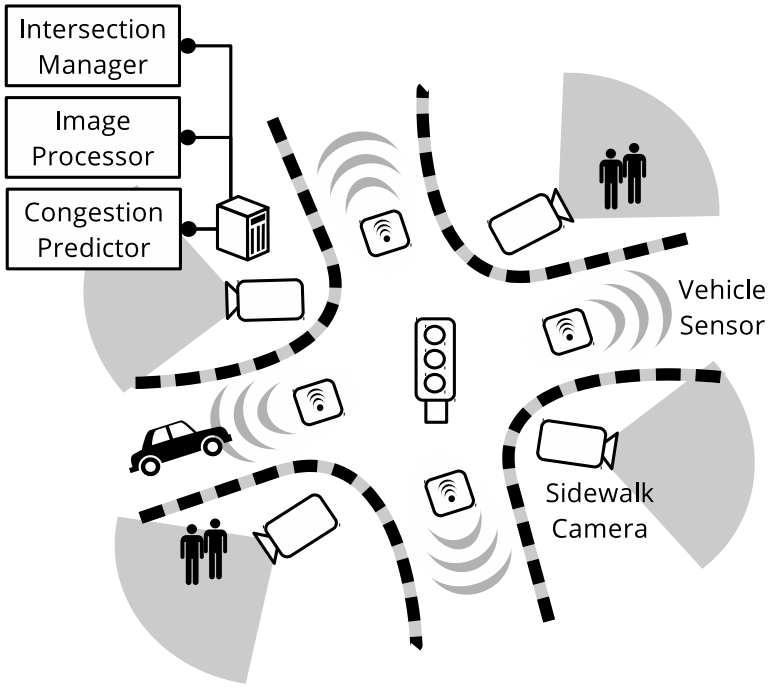


Figure 4.6: The IoT-based traffic intersection system in normal operation.

4.4 Case Study: Flexibility in a Traffic Intersection System

To demonstrate the proposed role modeling in designing behavior adaptation, the traffic intersection system (Fig. 4.6) is used as the case study. The traffic intersection system is modeled as self-organizing MAS, in which the components, i.e. sensors, devices, and applications, are represented as agents. In this case study, both failover flexibility and functional flexibility are demonstrated in a scenario when a system component experiences failure.

Cascading failure in the traffic intersection system has been addressed in Section 3.2. In this scenario, suppose West Vehicle Sensor experiences failure so that incoming (outgoing) vehicles from (to) west direction cannot be detected. Without the data of vehicles from the failed sensor, Congestion Predictor may not be able to predict congestion accurately. To solve this problem, Congestion Predictor should get the number of vehicles from another source, that is Image Processor. Normally, Image Processor processes the images provided by sidewalk cameras to count the number of pedestrian and performs other analyses. Additionally, Image Processor also includes the functionality to count the number of vehicles from a given image. To do so, Image Processor can instruct a nearby sidewalk camera, such as Southwest Sidewalk Camera, to switch its direction for recording traffic situation on the west side of the intersection (see Fig. 4.7).

The scenario demonstrates how some objects in an IoT system perform behavior adaptation by implementing flexibility to change the way they op-

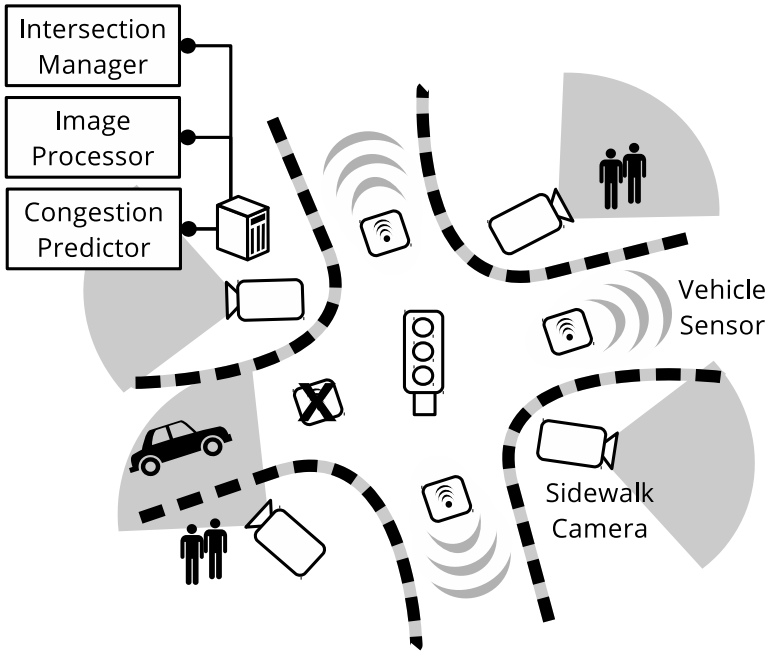


Figure 4.7: The IoT-based traffic intersection system with a vehicle sensor failure.

erate upon some failure and changes in the system. Failover flexibility is performed by the Congestion Predictor that change the way it retrieves and calculates the number of vehicles by switching from West Vehicle Sensor to Image Processor. Functional flexibility is performed by the Image Processor and Southwest Sidewalk Cameras that change the way they operate to provide vehicle number data. The difference between failover and functional flexibility is mainly the event that triggers the behavior adaptation. The former is triggered by failure, such as vehicle sensor failure, and usually requires the agent, e.g. smart device, to switch its interaction from the failed agent to a substitutable agent. This type of failure has been addressed in Chapter 3. The latter is triggered by other kinds of events, such as a request from another agent to change the way it operates.

4.4.1 Identifying and Elaborating Roles

Before identifying and elaborating the roles, agent classes should be identified. Identifying agent classes is out of the scope of this research. However, existing MAS engineering methodologies provide the ways to do so. Moreover, agent classes, that represent different types of agents, can be easily derived from the problem domain. Identifying agent classes in the traffic intersection system is straightforward. The agent classes are: Congestion Predictor, Image Processor, Sidewalk Camera, Vehicle Sensor, Intersection Manager, and Traffic Lights.

To model the behavior adaptation, the required roles and their capabilities should be defined. At runtime, agents perform behavior adaptation by changing from a set of roles to another, and thus different roles represent dif-

ferent behavior. For example, Congestion Predictor class requires two roles: Sensor-Based Congestion Predictor and Image-Based Congestion Predictor. The former represent the behavior of Congestion Predictor when all vehicle sensors work normally. The latter is required when one or more vehicle sensors fail, and then Congestion Predictor requests Image Processor to provide the number of vehicles. Consequently, Image Processor also requires two roles that correspond to the two situations. Pedestrian Image Processor role is played in normal situation, while Vehicle Image Processor role is played when the image processor is required to count the number of vehicle from traffic images. Similarly, Sidewalk Camera also requires two roles two handle the two situations, Pedestrian Monitor role and Vehicle Monitor role. The remaining agent classes only require one role. The roles and their description are listed in Table 4.4.

After the roles are identified, the next step is to elaborate the roles by defining their capabilities. If the designers choose to identify and elaborate roles using bottom-up approach, these steps will be performed in reversed order. Capabilities are the actions that can be performed or the functionalities that are provided by the agents. Capabilities can be realized in different ways, depends on the implementation language. For example, in object-oriented programming, capabilities are implemented as methods. Capabilities of the roles in the traffic intersection system and their implementation are addressed in Chapter 5.

Table 4.4: Roles in the traffic intersection system

Roles	Description
Traffic Signals Controller	Change the traffic lights according to the assigned timing
Traffic Manager	Set the traffic signals timing for the traffic lights
Sensor-Based Congestion Predictor	Predict congestion based on the number of vehicles that given by vehicle sensors
Image-Based Congestion Predictor	Predict congestion based on the number of vehicles that given by the image processor
Vehicle Image Processor	Count the number of vehicles from the images that given by sidewalk cameras
Pedestrian Image Processor	Count the number of pedestrians from the images that given by sidewalk cameras
Vehicle Monitor	Record traffic situation and send the images to the image processor
Pedestrian Monitor	Record pedestrians and sidewalks
Vehicle Detector	Detect vehicles that passes the road on which the vehicle sensor is installed

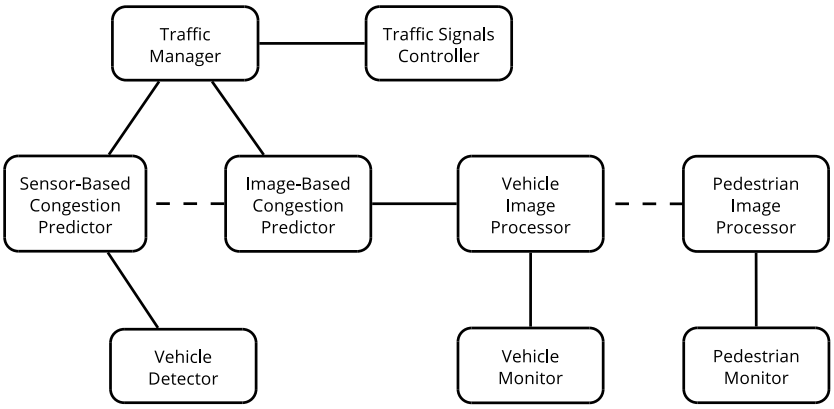


Figure 4.8: The role-relationship model of the traffic intersection system. Acquaintance roles are connected by a solid line, while associate roles are connected by a dashed line.

4.4.2 Designing Relationship and Interaction Between Roles

The relationship of the defined roles is illustrated in the RRM (Fig. 4.8). Roles are represented as rounded rectangles, while acquaintance and associate relations are represented as solid lines and dashed lines, respectively. For example, Traffic Signals Controller role and Traffic Manager role are acquaintance since the timing of the Traffic Lights should be defined appropriately by Traffic Manager. Sensor-Based Congestion Predictor and Image-Based Congestion Predictor are associate roles since they may be played concurrently by Congestion Predictor agent. Likewise, Vehicle Image Processor and Pedestrian Image Processor roles are also associate since Image Processor may play these two concurrently.

Having defined the RRM, the interaction between the roles are defined among the *acquaintance* roles (Fig. 4.9). An interaction is represented as

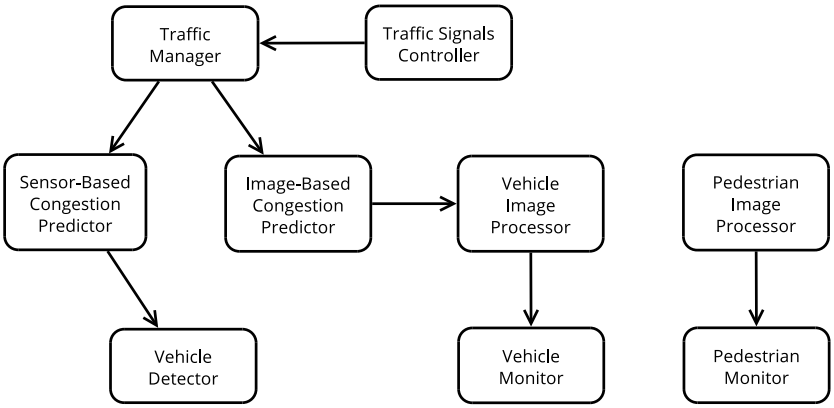


Figure 4.9: The role-interaction model of the traffic intersection system. Interactions are represented by arrows originated from the initiator roles.

an arrow from the initiator role to the responder role. For example, Traffic Signals Controller performs action to Traffic Manager, which is when Traffic Signals Controller requests the signal timing to control the Traffic Lights. Interaction between roles also reflects the dependency between the agent. Note that the arrows in Fig. 4.9 somewhat corresponds to the dependency links in the service network representation (Fig. 3.2).

4.4.3 Assigning Roles to Agent Classes

Figure 4.10 shows the REM of agent classes in the traffic intersection systems. The REM in this case is straightforward, in which the roles are assigned to the agent classes of which instance will play the roles at runtime. In more complex cases, designers can define different types of enactment and role association in REM that have been addressed in Section 4.3.3. The notations that are used in REM are illustrated in Fig. 4.11.

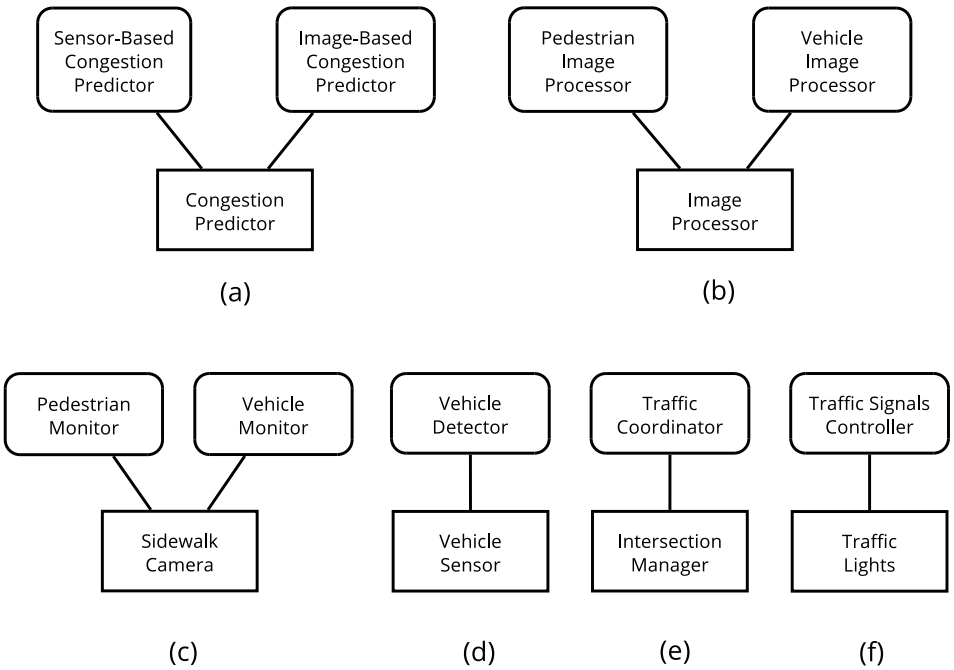


Figure 4.10: The role-enactment model of the agent classes in the traffic intersection system. Sharp-edge rectangles represent agent classes, while rounded rectangles represent roles. The role that is enacted by an agent class is connected with a line between them.

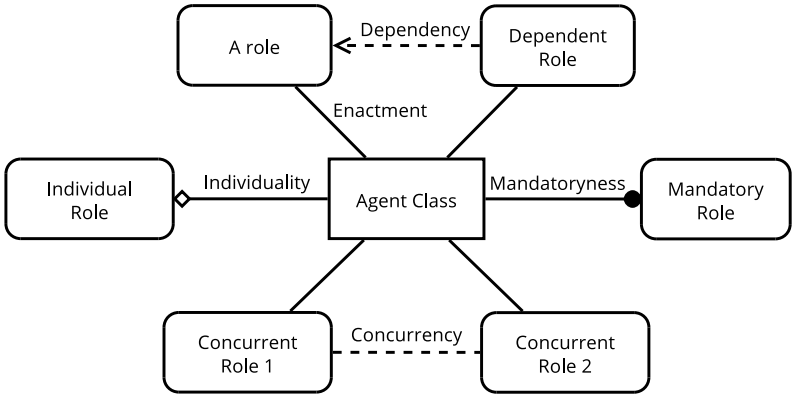


Figure 4.11: Different types of enactment and role association in role-enactment model.

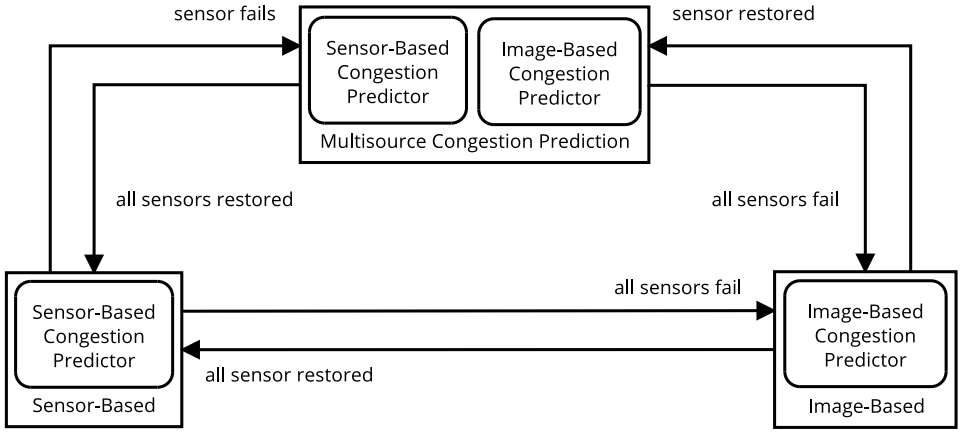


Figure 4.12: The role-transition model of Congestion Predictor agent class.

4.4.4 Designing Behavior Adaptation

The RTM of Congestion Predictor agent class is illustrated in Fig. 4.12. The RTM consists of three states, each of which is illustrated with sharp rectangles containing one or more roles. Sensor-Based state is the state when all the vehicle sensors are available, and thus the agent plays the role Sensor-Based Congestion Predictor. Transition from one state to another is indicated by an arrow. Sensor-Based state has two possible transition. When a sensor fails, it transitions to Multisource Congestion Prediction state that combines Sensor-Based Congestion Predictor role and Image-Based Congestion Predictor role. This means that the agent will play both roles concurrently. From this state, the agent returns to Sensor-Based state when all sensors are restored, or transitions to Image-Based state when all sensors fail. For completeness, the RTM of Image Processor and Sidewalk Camera agent classes are illustrated in Fig. 4.13 and Fig. 4.14, respectively.

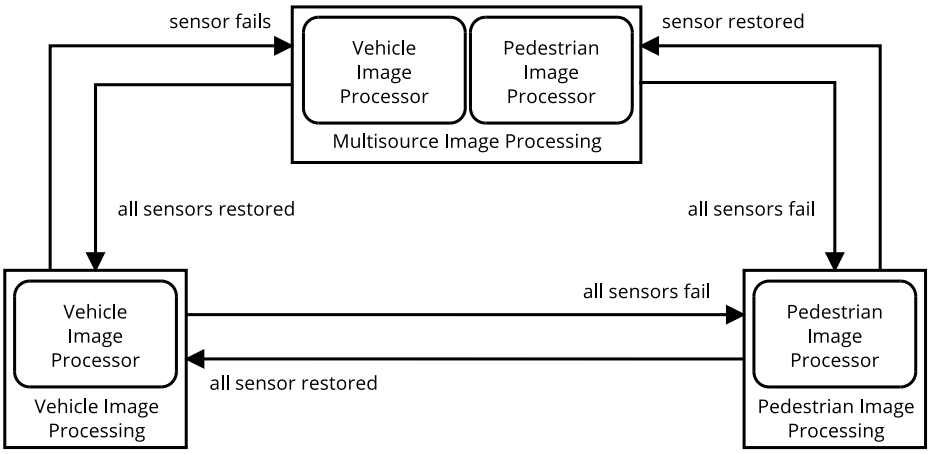


Figure 4.13: The role-transition model of Image Processor agent class.

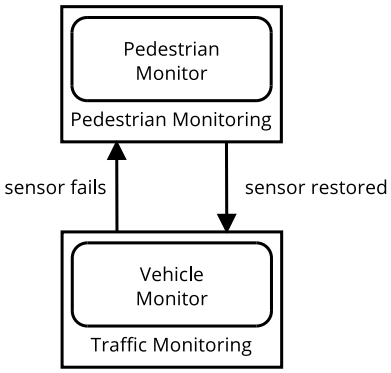


Figure 4.14: The role-transition model of Sidewalk Camera agent class.

4.5 Summary

To prevent cascading failure and to maintain the availability of the whole system, IoT services should be embedded with flexibility, which is the capability to respond upon changes and failure. This chapter proposes role-modeling method to design IoT-services flexibility as agents flexibility (agent behavior adaptation) in MAS. Designing agent behavior adaptation is not the same as designing agent flexibility, and thus the two design activities should be separated. Designing agent behavior deals with defining the actions that can be performed by the agents. By contrast, designing agent flexibility deals with defining which actions that can (or cannot) be performed on which situations.

To separate between designing agent behavior and designing agent flexibility, a high-level agent behavior notion is required. The proposed modeling method adopts role notion to represent agent behavior such that agents can change their behavior by changing roles. The use of role notion in MAS modeling method has several advantages.

1. Even though role notion has not been used to design agent behavior adaptation, it has been used in the existing MAS engineering methodologies to represent agent behavior. Therefore, MAS designers may have familiar with role notion and can easily grasp the way to design behavior adaptation using roles.

2. Role notion provides a concept to aggregate multiple agent capabilities to perform actions into a unit of agent behavior. This allows designers to define which actions that can be performed in which situations by organizing them into roles.

3. Role notion also allows agent behaviors to be easily integrated into transition diagram as in role-transition model (RTM). In RTM, each state is defined by a collection of one or more roles.

The main contribution of role-modeling method is its support to separate the design of agent behavior and agent flexibility. The separation allows designers to easily design agent flexibility and also improves maintainability. In addition, role-modeling method also assists designers to analyze the potential issues that may occur because of playing multiple roles. The applicability of the proposed role-modeling method is demonstrated using the traffic intersection system case study.

Chapter 5

Role-Based Language for Implementing Flexible Behavior in IoT Applications

5.1 Background

This chapter proposes a role-based AOP language to implement functional and failover flexibility of IoT services from the models created by role-modeling method. In Chapter 4, role-modeling method is proposed to provide suitable method for designing flexible IoT services as flexible (adaptive) agents in MAS. After creating the models of flexible agents, designers should implement the models in a simulated environment before they are realized in the actual environment. This work is motivated by the lack of support from existing AOP languages to implement the models of flexible

IoT services.

The use of agent-oriented programming[1] (AOP) would be convenience to implement MAS based on the models of flexible IoT services since agent-oriented properties, such as agent classes, goals, and actions, can be mapped and implemented directly in the program. However, most of existing AOP languages do not provide required features to easily implement agent flexibility. First, they do not provide a high-level notion to implement agent behavior which is required for directly implementing the models created by role-modeling method. Second, most of AOP languages mix the implementation of agent behavior and agent behavior adaptation (agent flexibility). This is also not suitable with the models created in role-modeling that separate the two modeling activities. Third, the language should allow designers to define possible change and failure events and the corresponding plans to respond to these events. However, this is also not supported by Jason [Bordini and Hübner, 2006], the most well-known AOP, which only chooses and executes one plan on each event.

The language is defined by extending AgentSpeak(L) [Rao, 1996], which originally follows BDI architecture in that agent mental states are defined in terms of goals, beliefs, and plans. The extension parts are converted to AgentSpeak(L) so that Jason interpreter can interpret them. The proposed language allows these mental states to be defined in roles such that agents can adopt and release roles at runtime, according to role-transition models. The language also has better support on modularity and plan execution since it is capable of handling plans having the same triggering events that are defined either in the same or in different files.

[1] Other researchers use the term "agent programming languages" (APL)

5.2 Modeling IoT Applications as Multi-Agent Systems

MAS modeling methods and implementation technology can be utilized in developing IoT services since both MAS and IoT services share common characteristics, which are autonomous, proactive, reactive, and social. IoT services can be considered as MAS because their components, i.e. smart devices, also exhibit autonomous characteristic. Smart devices also have proactive and social properties since they are embedded with computing and communication capabilities. The reactiveness of IoT-services components is the result of their capability to sense the environment, as in sensor devices, and to respond upon perceptions. These common characteristics motivate our approach to model and implement adaptive behavior of smart devices in IoT services using MAS modeling method and AOP.

The capability of smart devices to change the way they operate at runtime upon changes and failure can be modeled in MAS using a high-level agent behavior concept such as role. Role notion has also been used to represent behavior and functionalities of systems in IoT-related fields, such as in distributed and embedded systems [Ren et al., 2007] and smart device systems [Kawsar et al., 2008]. Even though most of MAS engineering methodologies allow representing agent behavior using role notion, they do not provide suitable feature to design agent behavior adaptation and dynamic structure reorganization at runtime [Tran and Low, 2005]. One of the MAS engineering methodologies that try to support such feature is O-MaSE [DeLoach and García-Ojeda, 2010], which specifically addresses agent behavior adaptation and dynamic reorganization of MAS. However, its approach

to perform agent behavior adaptation and MAS reorganization is rather centralized since it requires to calculate the best combination of roles, agents, and the current goal sets [Lhaksmana et al., 2013].

Due to the lack of support from MAS engineering methodologies to design agent adaptive behavior at runtime, in previous work (Chapter 4) we have proposed a modeling method based on role concept [Lhaksmana et al., 2013]. The role-modeling method allows designers to conveniently model how agent change their behavior at runtime. A role represents a unit of agent behavior, which is defined in terms of agent capability to perform actions. We adopt role-based modeling method to model flexible IoT services that are capable to change the way they operate upon changes and failure. The advantage of role-based modeling method is that it allows MAS designers to define high-level agent behavior in terms of roles, and how the agents change their behavior by making transition from a set of roles to another. In addition, it also separates the design of agent behavior and agent behavior adaptation. The former defines the actions that can be performed by the agents, whereas the latter defines which actions that can be performed on which situations. The separation is the important feature in role-modeling method to improve maintainability in design and implementation level.

In this chapter, we propose an AOP language that supports the implementation of the role-modeling method to implement IoT-services flexibility as adaptive agents in MAS. In the next section, we address some of the notable AOP languages and their support for implementing agent behavior adaptation using role notion.

5.3 Agent-Oriented Programming

Although most of the MAS applications have been built using regular (not agent-oriented) programming languages, research communities actively utilize and extend AOPs [Bordini et al., 2006]. According to a survey [Müller and Fischer, 2014], Java has been used by more than half of MAS development. They also identified some agent platforms that have been used frequently, such as Jade, WADE, Jack, Co-Jack, C-BDI, etc. Even though the use of AOPs for implementing MAS applications is limited, they are useful for developing MAS simulations, which are often required due to the complexity of MAS. In this section, some AOPs and platforms are discussed briefly as an overview of the latest status in this field.

One of the first effort in AOP is AGENT-0, which proposes an AOP paradigm as a specialization of object-oriented programming (OOP) language [Shoham, 1993]. An application that built using OOP consists of multiple modules (classes), each of which is instantiated as one or more objects at runtime. In AOP, the object-like artifacts are called agents, which is also capable to run independently and to communicate with other agents. The distinction of agent concept against object in OOP is its mental state properties which are influenced by belief-desire-intention (BDI) architecture. AGENT-0 defines its mental state properties as a simplified version of BDI, such as beliefs, capabilities, and decisions. These mental states are somewhat a simplified version of belief-desire-intention (BDI) model for intelligent agents.

AgentSpeak(L) is a BDI-based AOP language that was proposed following the aforementioned AGENT-0 [Rao, 1996]. The language is well-known

in research community, especially after Jason [Bordini and Hübner, 2006], the interpreter for its extended version was published. The first version of Jason was released in 2004 and it is still actively maintained until today. The reception of Jason by MAS community is shown by the existence of some MAS development tools, extensions, and projects related to Jason that have been proposed by other researchers. Jason provides BDI-based agent-oriented paradigm, is actively maintained, and is widely recognized by MAS community.

Most of the proposed AOP languages try to support the implementation of intelligent agents and MAS. Along with this direction, some other related effort tries to support interagent communication, such as KQML [Finin et al., 1994] and AgenTalk [Kuwabara et al., 1995]. The former proposes a language and protocol for knowledge sharing between agents, while the latter is a programming language to describe protocols among multiple agents based on extended finite state automata. The use of finite state automata is also proposed in Q language to describe behaviors of agents for multiagent simulation [Ishida, 2002].

In addition to AOP languages, the effort to support MAS development has also produced frameworks and some development tools. MOISE is an AOP framework that extends AgentSpeak(L) based on Jason features [Hübner et al., 2007]. MOISE was proposed to support MAS organizational modeling such that agents are capable of perceiving organizational changes. MOISE sequential activities of defining groups, roles, mission, etc., are aimed to properly model the organizational specification of MAS. However, it is not practical to design flexible agent behavior based on roles due to the following. MOISE, and also Jason, do not provide a feature to clearly sep-

arate the implementation of agent behaviors and agent behavior adaptation. In these languages, agent behaviors are mainly implemented as goals and plans. To respond against changes and failures, developers should define if-else structure within the plans or define different plans for each change and failure situation. This is not suitable for implementing the model of IoT-services flexibility, since separation between agent behaviors and behavior adaptation is the important feature in role-modeling method to improve maintainability. The limitations of the existing AOP languages and frameworks motivate our research to propose an AOP language to support the implementation of agent behavior adaptation in MAS.

5.4 Extending AgentSpeak(L) and Jason Interpreter

We propose an AOP language for implementing smart device adaptive capability in IoT services by extending AgentSpeak(L) and modifying Jason to implement the extension. AgentSpeak(L) is used since both the language and Jason interpreter have been widely recognized in MAS research community. It is also a BDI-based AOP language that provides straightforward way to implement agent mental states. The proposed language supports the implementation of agent behavior adaptation models of MAS that are created by role-modeling method [Lhaksmana et al., 2013].

5.4.1 Agent Mental States in Jason

As a BDI-based language, Jason agents are characterized by mental states, such as belief, goal, and plan. Agent's belief captures the information about the environment where the MAS is situated. A goal expresses the state that should be achieved by the agent. The structure of a plan consists of triggering event and context, which also refers as *head*, and its *body* that comprises some actions to be executed or goals to be achieved. When an event occurs, Jason triggers the appropriate plan whose head satisfies the event.

To date, the current version of Jason (1.4.2) has not supported role notion. Each agent defines its behavior in terms of mental states. Even though two or more agents share the same mental states, i.e. the same beliefs, goals, or plans, these mental states must be implemented by each agent. At runtime, an agent may send messages to another agent to share its beliefs and goals. However, it is not possible to share the plans of one agent to another. With these limitations, it is not practical to implement role-based flexibility of MAS in Jason.

5.4.2 Adopting Role Modeling in Jason

To implement the proposed AgentSpeak(L) extension, role notion and its related concepts from role modeling [Lhaksmana et al., 2013] are adopted into Jason. As illustrated in Fig. 5.1, the adoption of role modeling concepts into Jason is achieved by adding some new classes indicated in gray color. Role class represents a unit of agent behavior in terms of beliefs, goals, and plans,

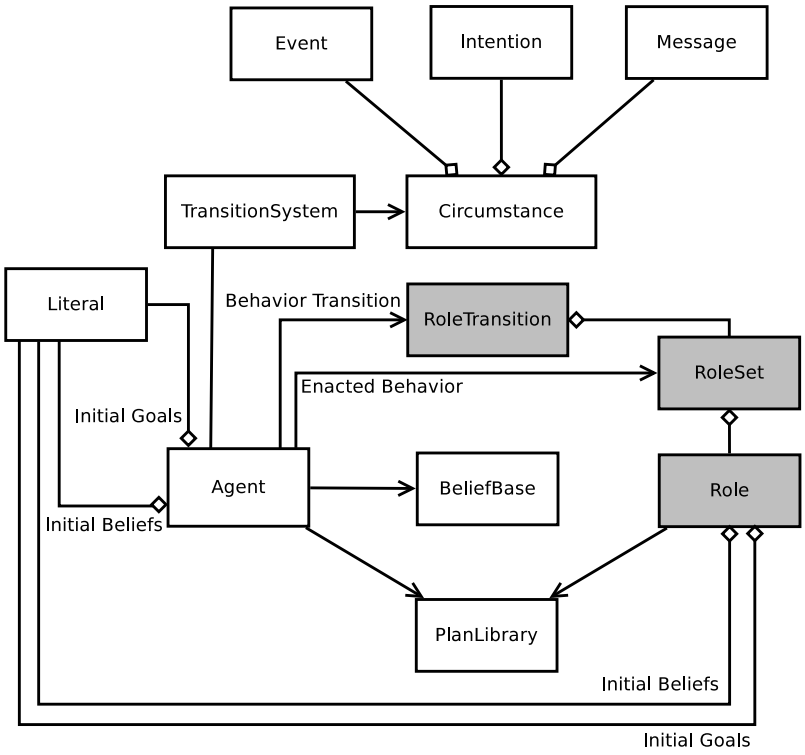


Figure 5.1: Diagram of some core classes of the extended Jason. The newly added classes are colored gray.

as well as initial beliefs and goals. Since role modeling allows an agent to play one or more roles at a time, the combination of roles is implemented in RoleSet class. The current combination of roles that is being played by the agent is indicated by the Enacted Behavior association. RoleTransition class contains the possible role sets that can be played by the agents, and the transition between these role sets. The other classes that are colored white are the existing Jason classes. Only some of the most important and relevant Jason classes are included since the figure is intended to show the extension. The most central class in Jason is the Agent class. This class was modified

to support the extension and to utilize the extended classes.

5.4.3 Extending Jason Grammar

Listing 5.1: The extended Jason grammar

```
agent         → enactment init_roles init_beliefs init_goals plans
enactment     → "enact" role ("," role )* "."
init_roles    → ( role_adoption "." )*
role_adoption → ".adopt(" role ")"
role_release  → ".release(" role ")"
curr_role     → ".play(" role ")"
role          → literal
```

Jason grammar is extended to introduce role and its related notions for role modeling implementation. The grammar of the extended Jason is described in Listing 5.1. In the extended grammar, a Jason agent may define `enactment` that specifies the roles to be enacted by the agent. The term `init_roles` defines the roles that will be played by the agent when it is started. The terms that formatted in italic, i.e. `init_beliefs`, `init_goals`, `plans`, and `literal`, are part of original Jason syntax that are included in the above for completeness. Agent play roles by executing role adoption using `.adopt` internal action. To release a role that is being played, it execute role release using `.release` internal action. Another internal action, `.play`, is provided to check whether or not a role is currently being played. The extended grammar above will give a pseudocode as illustrated in Listing 5.2. In line 2, the enacted roles are defined. The roles that will be played when the agent is instantiated are shown in line 5. Finally, the plan to perform a transition by releasing and adopting roles is illustrated in line 14.

Listing 5.2: A pseudocode of an agent implementation in the extended Jason

```
// enacted roles
enact role1,role2,role3.

// initial roles
.adopt(role1,role2).

// initial beliefs
...

// initial goals
...

// transition plans
+triggering_event :
  condition, role1, role2 <-
  .release(role1);
  .adopt(role3).
...
```

To implement the proposed grammar, we extend Jason architecture as illustrated in Fig. 5.2. The extended module is indicated in gray, i.e. the Extended AgentSpeak(L) Preprocessor. This extension responsible on transforming the code that is written in the proposed language into Jason's AgentSpeak(L) language. The result of the transformation is passed to the AgentSpeak(L) parser. The extension is mainly implemented in J-Edit programming environment, which is the standard IDE that included with Jason installation.

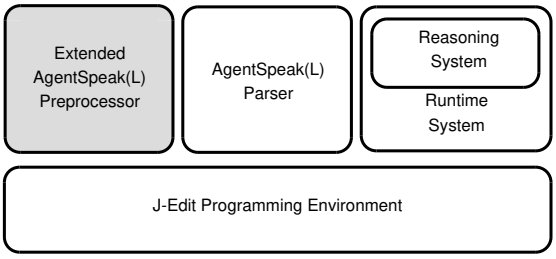


Figure 5.2: Architecture of the extended Jason interpreter.

5.5 Case Study

5.5.1 Description

This section provides a case study, a traffic intersection system (Fig. 4.7), to provide a proof of concept for the applicability of the proposed AOP language for implementing adaptive behavior of IoT services in MAS. To avoid unnecessary details, the case study has been made simple compared to the actual traffic intersection systems, which usually include more components and provide more features. The traffic intersection system is an IoT-based system that consists of some smart devices, sensors, and applications that capable to interact with each other. The purpose of the traffic intersection system is to reduce the possibility of congestion and the risk of accident by controlling the traffic lights according to the number of vehicles in the intersection. The traffic intersection system is modeled as MAS, in which the components, i.e. sensors, devices, and applications, are represented as agents.

In this case study, agent behavior adaptation is demonstrated in a scenario when an agent, i.e. a system component, experiences failure. Suppose the

vehicle sensor that installed on the west direction experiences failure so that incoming (outgoing) vehicles from (to) this direction cannot be detected. In Fig. 4.7, the failed vehicle sensor is indicated with a cross sign. Without the vehicle data from the failed sensor, Congestion Predictor system may not be able to predict congestion accurately. Consequently, Intersection Manager system will not be able to set the traffic lights timing appropriately. To solve this problem, Congestion Predictor should get the number of vehicles from another source, that is from Image Processor system. Normally, Image Processor processes the images that provided by sidewalk cameras to count the number of pedestrian and performs other analyses. In addition, Image Processor also includes the functionality to count the number of vehicles from a given image. To do so, Image Processor should instruct a nearby sidewalk camera, such as the southwest Sidewalk Camera (bottom left sidewalk camera in Fig. 4.7), to switch its direction for recording traffic situation on the west side of the intersection.

Failover Flexibility

The scenario demonstrates how some agents in the traffic intersection system perform behavior adaptation by changing the way they operate upon some failure and changes in the system. The Congestion Predictor demonstrates *failover flexibility* to respond upon the West Vehicle Sensor failure. It changes the way it retrieves and calculates the number of vehicles by switching from the West Vehicle Sensor to the Image Processor. In Fig. 3.2, it is shown that West Vehicle Sensor and Image Processor are substitutable services. Further discussion about substitutable services and service network model [Lhaksmana et al., 2015] are provided in Chapter 3.

Listing 5.3: Failure handling in Jason

```
...

// a plan that will be executed when goal1 is added
+!goal1 <-
  ...

// failure handling that will be executed
// when the above plan experiences failure
-!goal1 <-
  ...
  // actions for failure handling

...
```

Functional Flexibility

Image Processor and the Southwest Sidewalk Camera also change the way they operate to provide vehicle number data. In this case, Southwest Sidewalk Camera changes its behavior from capturing the image of pedestrian to the image of the vehicles. Likewise, Image Processor also adjusts the way it processes the images from processing the image of pedestrian to processing the image of vehicles. The flexible behaviors that are demonstrated by Southwest Sidewalk Camera and Image Processor are *functional flexibility* since they do not involve component substitutions.

Failure Handling

Another important feature of the extended Jason is its support to implement failure handling at role-level. Jason supports failure handling at agent-level

by specifying plan with goal deletion triggering event [Bordini et al., 2007]. As illustrated in Listing 5.3, the syntax for failure handling follows Jason failure handling except that the failure handling plans can be defined at both agent-level and role-level, i.e. within agent code and role code. In the extended Jason, this allows failure handling to be generalized at role-level, which consequently improves code quality. Moreover, it also allows one to override failure handling by redefined the plan at the agent-level.

5.5.2 The Implemented Jason Agents

Each type of the traffic intersection system component is represented as a Jason agent. Therefore, the implemented Jason agents are *congestion_predictor*, *image_processor*, *sidewalk_camera*, *intersection_manager*, and *traffic_lights*. In addition, *simulator* agent is required to control the simulation, such as to simulate sensor failures and to restore the failed sensors. The goals of each agent and its plans are addressed as follows.

1. Simulator agent
   The simulation of the traffic intersection system is controlled by the *simulator* agent. It starts the simulation and instructs *vehicle_sensor* agents to simulate failure and restoration, which consequently triggers *congestion_predictor* agent to change the way it calculates the number of vehicles and predicts congestion.

2. Vehicle Sensor agent
   As illustrated in Fig. 4.7, the traffic intersection system has four vehicle sensors, which located in each direction of the intersection. Jason allows an agent to be realized as multiple instances. Thus, these four

vehicle sensors are implemented as four instances of *vehicle_sensor* agent. The objective of a *vehicle_sensor* agent is to provide number of vehicle to *congestion_predictor* agent. It also defines some plans to simulate failure and restoration, which will be executed upon receiving instruction from *simulator* agent.

3. Congestion Predictor agent

   The *congestion_predictor* agent requires vehicle detection data from *vehicle_sensor* agent to predict congestion, hence the association between them. Upon receiving data from *vehicle_sensor*, it executes a plan to calculate the current number of vehicles and then calculate the likelihood of congestion. When one or more *vehicle_sensor* agents fail, it requests *image_processor* agent to provide vehicle detection data which will be handled another plan.

4. Image Processor agent

   The *image_processor* agent processes the images that provided by *sidewalk_camera* agents. Normally, *sidewalk_camera* agents provide images of sidewalk and pedestrians so that *image_processor* can calculate the number of pedestrians in the intersection or send the image to the traffic management center for further processing. When one or more *vehicle_sensor* agents fail, having received request from *congestion_predictor*, *image_processor* requests one or more *sidewalk_camera* agents to provide traffic images instead of sidewalk images.

5. Sidewalk Camera agent

   Similar to the *vehicle_sensor* agents, four instances are defined to represent four sidewalk cameras as illustrated in Fig. 4.7. The *side-*

walk_camera agents record the sidewalks around the intersection and also record the vehicles when necessary.

6. Intersection Manager and Traffic Lights agents
   The *intersection_manager* agent calculates the right timing for the *traffic_lights* agent based on the traffic situation and the likelihood of congestion that are provided by *congestion_predictor*.

5.5.3 Implementation in Jason

Due to the limited space, only the implementation of *congestion_predictor* agent is discussed in this section. However, it is representative to demonstrate the realization of behavior adaptation in Jason and its extension. The pseudocode of *congestion_predictor* agent for the implementation in the original Jason is provided in Listing 5.4. To avoid unnecessary details, the body of the plans is only described in comments.

Listing 5.4: The pseudocode of *congestion_predictor* agent in the original Jason

```
1  // initial goal
2  !predict_congestion.
3
4  // plan for pursuing predict_congestion goal
5  +!predict_congestion <-
6    // get the list of number of vehicles
7    // calculate the probability of congestion
8
9  // on receiving sensor_data belief from vehicle_sensor S
10 // that provides the number of vehicle N
11 +sensor_data(S, N) <-
12   // get vehnb_fsen belief having the number of vehicle Nv from sensor S
13   // update the belief on number of vehicle
```

```
14    // calculate the likelihood of congestion:  !predict_congestion
15
16  // on receiving imgproc_data belief from image_processor
17  // that provides the number of vehicle N from direction D
18  +imgproc_data(D, N) <-
19    // update the belief on number of vehicle
20    // calculate the likelihood of congestion:  !predict_congestion
21
22  // on failure of one or more sensors
23  +sensor_failed(S) <-
24    // update the list of active sensors
25    // set require_imgproc flag to true
26
27  // when a failed sensor is restored
28  +sensor_restored(S) <-
29    // update the list of active sensors
30    // if all sensors are active
31    // then set require_imgproc flag to false
32
33  // update the list of active sensors
34  ...
```

The agent *congestion_predictor* has `!predict_congestion` initial goal (line 2). The plan to achieve this goal is defined in line 5, that calculates the total number of vehicles and the likelihood of congestion in the intersection. The plans for belief addition of `+sensor_data(S, N)` (line 11) and `+imgproc_data(D, N)` (line 18) are executed when the agent receives number of vehicles data from *vehicle_sensor* and *image_processor* agents, respectively. These plans update the number of vehicles in the agent's belief base, and then triggers the achievement of `!predict_congestion` goal. The last two plans in line 23 and line 28 are executed when the agent receives notifications (beliefs) that a *vehicle_sensor* fails (`+sensor_failed(S)`) and recovers from failure

(`+sensor_restored(S)`), respectively. The former updates the list of active sensors, and then adds the belief `+require_imgproc(true)` to indicate that the agent requires number of vehicles data from *image_processor*. Similarly, the latter updates the list of active sensors and updates the belief if all sensors have returned to active.

Listing 5.4 shows that the core functionalities of *congestion_predictor* agents are defined in the first three plans, which are triggered by belief addition events `+!predict_congestion`, `+sensor_data`, and `+imgproc_data`. The other plans are defined to handle *vehicle_sensor* failure and restoration. These show that in Jason, both the core functionalities of the agents and the way the agents response upon changes and failures are not distinguished. For simple program as above, programmers are still able to distinguish easily agent behavior (core functionalities) and behavior adaptation (flexibility). However, for larger and more complex MAS that involves more agents and interaction between them, mixing the two may cause the program becomes difficult to be maintained.

5.5.4 Implementation in the Extended Jason

To allow Jason programmers to separate agent's core functionalities and agent behavior adaptation, Jason is extended by adopting role modeling [Lhaksmana et al., 2013]. In role modeling, agent's core functionality is called agent behavior, which is represented using role notion. Agent flexibility is modeled as transitions from a set of roles to another. The use of role to represent agent behavior also introduces some advantages. The same behavior that is exhibit by different agents can be implemented in a role so

that developers do not need to implement the same behavior multiple times.

Jason is extended such that agent's goals, initial beliefs, and plans can be grouped into roles. Agents play a combination of roles and change roles at runtime to adopt different behaviors. The roles that are enacted by an agent can be defined in the agent code itself or in their own files. The latter allows programmers to organize agents and roles into modules, which consequently improve the maintainability of the program.

To demonstrate the separation of agent behavior and behavior adaptation, the agent *congestion_predictor* enacts two roles: *sensor-based_pred* and *image-based_pred*. The former will be played when one or more *vehicle_sensor* agents provide vehicle number data. The latter will be played when one or more *vehicle_sensor* agents fail so that one or more *image_processor* agents provide the number of vehicle data. The implementation of *congestion_predictor* agent in the extended Jason is provided in Listing 5.5, while Listing 5.6 and Listing 5.7 are the code of *sensor-based_pred* and *image-based_pred* roles, respectively.

Listing 5.5: The pseudocode of *congestion_predictor* agent in the extended Jason

```
1  // enacted roles
2  enact sensor-based_pred, image-based_pred.
3
4  // initial role
5  .adopt(sensor-based_pred).
6
7  // role-transition plans
8
9  // on failure of one or more sensors
10 +sensor_failed(S) <-
11   // update the list of active sensors
```

```
12    // adopt role image-based_pred
13    if (not .play(image-based_pred)) {
14      .adopt(image-based_pred);
15    }.
16
17  // when a failed sensor is restored
18  +sensor_restored(S) <-
19    // if all sensors are active
20    // then release role image-based_pred
21    ?active_sensors(L);
22    .length(L, X);
23    if (X == 4 ) {
24      // release role image-based_pred
25      .release(image-based_pred);
26    };
27
28    // update the list of active sensors
29  ...
```

The main objective of adopting role into Jason is to separate the implementation between agent behavior and behavior adaptation. Agent behavior is implemented in roles, while behavior adaptation is implemented in agents. For *congestion_predictor* agent, the core agent behaviors are goals and plans to predict congestion and how the agent handle the required data. Therefore, `!predict_congestion` goal and its plan, as well as the plans to handle `+sensor_data` and `+imgproc_data` beliefs, are removed from *congestion_predictor* agent (Listing 5.5) to be implemented in their corresponding roles (Listing 5.6 and 5.7). Basically, the agent only requires initial roles and plans to perform role transition, i.e. releasing and adopting one or more roles to change its behavior. In Listing 5.5, role transitions are implemented in the plans that handle `+sensor_failed` and `+sensor_restored` belief.

Listing 5.6: The pseudocode of *sensor-based_pred* role

```
// initial goal
!predict_congestion.

// plan for pursuing predict_congestion goal
+!predict_congestion <-
  // get Lsensor the list of number of vehicles
  // calculate the probability of congestion:
  !calculate_probability(Lsensor).

// on receiving sensor_data belief from vehicle_sensor S
// that provides the number of vehicle N
+sensor_data(S, N) <-
  // update the belief on number of vehicle
  // calculate the likelihood of congestion:
  !predict_congestion.

...
```

Listing 5.7: The pseudocode of *image-based_pred* role

```
// initial goal
!predict_congestion.

// plan for pursuing predict_congestion goal
+!predict_congestion <-
  // get Limgproc the list of number of vehicles
  // calculate the probability of congestion:
  !calculate_probability(Limgproc).

// on receiving imgproc_data belief from image_processor
// that provides the number of vehicle N from direction D
+imgproc_data(D, N) <-
  // update the belief on number of vehicle
  // calculate the likelihood of congestion:
  !predict_congestion.

...
```

Both *sensor-based_pred* and *image-based_pred* implement the initial goal of `!predict_congestion` and its plan. However, the actions that defined in the plan body are only those that are relevant to the role. For example, the body of the plan in line 5 Listing 5.6 consists of retrieving the number of vehicle according to the *vehicle_sensor* agents, and then calculates the probability of congestion according to the number. Likewise, the body of the same plan in Listing 5.7 only concerns on the number of vehicle that is provided by the *image_processor* agent.

5.6 Evaluation

5.6.1 Features of the Extended Jason

The comparison between the features of the proposed language and the original Jason is listed in in Table 5.1. In the proposed language, the implementation of the enacted roles can be defined in the agent itself or in separate files. Original Jason provides a directive to preprocess source code, such as to include source code from another file using include directive, i.e. { `include(<filename>)` }. However, the include directive does not perform any additional processing other than copying the source code. When both the agent code and the included code contain the plans with the same event, only one of them will be executed even though the context of each plan is satisfied. This is due to Jason paradigm where the execution of an appropriate plan for a given context is believed will satisfy the current goal. Definition of multiple plans for the same event should be supported in implementing flexible IoT services since designers are expected to design

Table 5.1: New features in the extended Jason compared to the original Jason

Feature	Extended Jason	Original Jason
Preprocessing	Combine plans that have the same triggering event. Adding context to plans to specify at which role the plans will be executed. Adding code to agent file source code before it is parsed in Jason.	Simply copy codes from another file to the point where the include directive is specified
Initial goal execution	Executed every time the role is adopted (on `.adopt`)	Executed only once when the agent starts
Plan definition	Allow to have plans with the same triggering event and context in different roles	A plan will override other plans that have the same event and context, i.e. the other plans will be useless
Plan execution	Plans are executed according to which roles are being played by the agent	No distinction between the plans that defined in the included file and in the agent itself
Multiple plan execution	For each event, multiple plans can be executed when the specified conditions are satisfied	Only at most one plan is chosen to be executed

different responses for different changes and failures conditions. The initial goal execution between the proposed language and the original Jason is also different. In the proposed language, the initial goals that are defined in the roles will be executed every time the roles is adopted by the agent. On the other hand, initial goals that are included through include directive is only executed once when the agent is instantiated. Role adoption also decides which plans that are active and can be executed by the agent. This is not possible in the original Jason; all plans are always executable either those defined in the included files or in the agent script itself. Role adoption is the important feature to implement behavior adaptation since it decides which plans that can be executed on which conditions.

5.6.2 Maintainability

Readability is one of the criteria to evaluate maintainability of a program. One of the most straightforward readability measure that is relevant to evaluate our proposed language is the number of non-commented lines of code, NCLOC for short, which has been used to evaluate multiagent programming [Lillis et al., 2013]. We implemented the complete code of *congestion_predictor* agent, of which pseudocode is in Listing 5.4, in 32 NCLOC. In the extended Jason, the same agent is implemented in three files, which are the agent code itself (20 NCLOC), and two roles (*image-based_pred* and *sensor-based_pred*) each of which is 9 NCLOC. The pseudocode versions of these implementations are Listing 5.5, Listing 5.6, and Listing 5.7, respectively. Even though the total NCLOC in the extended Jason is higher, i.e. 38 NCLOC, the average of NCLOC for each file is $38/3 \approx 13$ NCLOC.

The lower number of NCLOC average and the modularization of code have been widely belief could improve readability and maintainability.

5.7 Summary

A programming language is proposed to implement failover and functional flexibility of IoT services as adaptive agents in MAS based on the models created from role-modeling method. The use of AOP would be convenience to implement flexible IoT services as agents since agent-oriented properties, such as agent classes, goals, and actions, can be mapped and implemented directly in the program. However, most of existing AOP languages do not provide a high-level notion to implement agents roles. In addition, AOP languages also do not provide a feature to clearly separate agent behavior and behavior adaptation (agent flexibility). The separation of agent behavior and agent flexibility is the important feature in role-modeling method to improve maintainability of the program.

Against this background, a language is proposed to allow IoT-services flexible capability to be implemented as high-level agent behavior using role notion. The language is defined by extending AgentSpeak(L), which originally follows BDI architecture in that agent mental states are defined in terms of goals, beliefs, and plans. The extension parts are converted to AgentSpeak(L) so that Jason interpreter can interpret them. The proposed language allows these mental states to be defined in each role such that agents can adopt and release roles at runtime according to role-transition models. The language also has better support on modularity since it is ca-

pable of handling plans having the same triggering events that are defined either in the same or in different files. The current Jason (1.4.2) does not handle plans with the same triggering event; only one plan that can be executed, whereas the other plans become useless. An implementation of a case study demonstrates that the modular implementation of roles does not significantly increase the number of non-commented lines of code. This shows that the modularity can be increased while maintaining readability.

Chapter 6

Conclusion and Discussion

This thesis tackles reliability issue in IoT, which is one of the challenges in realizing its potential. The strength of IoT is a result of the existence of billions smart objects, i.e. devices and sensors with computing and communication capability that are integrated to the Internet. These massive number of smart devices are capable of interacting with each other, with the existing publicly-accessible services and applications in the Internet, and with the environment where they are situated.

The reliability issue of IoT arises from its dynamic characteristic and unpredictable environment. First, the dynamic characteristic of IoT is caused by the mobility of smart objects and the distributed nature of IoT applications. The number of mobile smart objects in IoT, such as mobile devices, wearable devices, and other mobile M2M devices, is predicted to increase significantly so that it will be dominant in the future IoT [Cis, 2015]. The mobility of such smart objects allows them to continuously join (leave) to

(from) different networks while they are roaming through different locations, and thus dynamically connect and disconnect with the other reachable smart objects.

The distributed nature of smart objects is also a factor that may cause reliability issue. Different smart objects may be developed and maintained independently by different companies. Modification in one smart object may not be noticed by other smart objects that interact with it. Consequently, a smart object may not be able to get the expected response from the other smart object after it is modified. Another cause of reliability issue is the unpredictable environment where IoT applications are situated. Smart objects may interact directly with the environment using actuators and sensors attached to them. The environment where these smart objects are situated is subject to change. Moreover, some sensors are deployed in hostile environment, such as sensors for atmosphere monitoring, tsunami detection, etc. In such environment, smart objects should have the capability to respond upon unexpected changes and failure.

In addition to the reliability issue above, a particular type of failure is also investigated in this thesis. From network point of view, cascading failure can be defined as a type of failure that propagates from one node (link) to some other nodes (links). This type of failure has been studied in power network and infrastructure domain since major cascading failures involving large area and infrastructure network occurred in the past. However, cascading failure in a network in which the nodes are interdependent to each other has different characteristic and has not yet studied.

6.1 Contributions

This thesis proposes solutions for the reliability issue in service-based IoT applications, or IoT services for brevity, by embedding flexibility. Here, flexibility is the capability of systems to change their behavior at runtime to respond upon changes and failure. To this end, IoT services are modeled as self-organizing MAS where agents represent services. Thus, the interaction and dependency between agents represent interaction and dependency between services. Flexibility is defined at the agent level as failover flexibility and functional flexibility. The former is the agent capability to switch among substitutable agents upon internal and external failures, while the latter is the agent capability to change its behavior to enable different actions upon internal and external changes. The proposed solutions give the following contributions:

1. Provide an analysis on IoT service network tolerance and how failover flexibility could significantly improve the tolerance.

   IoT service network tolerance to cascading failure is determined by the degree of interdependency between service and the network topology. In this work, cascading failure is simulated on service networks with different degree of interdependency and different network topology. Among some important findings, the simulation and analysis results suggest service network to have failover flexibility to reduce the risk of cascading failure. The results also suggest that the tolerance to cascading failure can be improved significantly by increasing the average number of substitutable services when the network currently has rather low number of substitutable services; the effect is not sig-

nificant when otherwise. In addition, this work also finds scale-free topology shows better tolerance, followed by exponential and random topology. This is also an important finding since related work in power grids shows random topology to have better tolerance against different kind of cascading failure.

2. Introduce role-modeling method for designing functional flexibility of IoT services.

   Abstracting IoT services as self-organizing MAS allows them to be designed using MAS modeling methods. However, existing MAS engineering methodology do not provide convenience modeling method to design agent flexibility in self-organizing MAS. To model agent flexibility, role notion is used to represent agent behavior. Role notion provides a high-level representation of agent behavior such that agents, i.e. IoT services, can change their behavior at runtime by switching from a set of roles to another to respond upon changes. Role modeling improves maintainability and provides a convenience way to design agent flexibility by separating the design of agent behavior and agent behavior adaptation (flexibility).

3. Provide an AOP extension to implement functional and failover flexibility of MAS for simulating IoT systems.

   The flexibility of IoT services should be implemented in a simulated environment before it is implemented in the actual settings. To this end, a programming language is required to implement the flexibility based on the models created by role-modeling method. To easily implement the models, the programming language should support role notion and the separation of agent behavior and agent behav-

ior adaptation. A programming language is proposed by extending AgentSpeak(L) programming language that is implemented on Jason interpreter. Compared to the original programming language, the use of role improves maintainability by supporting better modularity. In addition, it also supports multiple-plan execution that is required for implementing flexible IoT services.

6.2 Future Direction

This thesis deals with reliability issue in IoT services and proposes its solutions, which consists of fault-tolerance analysis, a modeling method, and a programming language. The proposed solutions open the possibility to pursue future directions as follows:

1. Extending role-modeling method to provide an end-to-end self-organizing MAS engineering methodology.
   Currently, role-modeling method only covers design phase and focuses on designing agent behavior. Other system development phases, such as analysis and testing should be considered to provide a complete end-to-end MAS engineering methodology for developing IoT services. Moreover, testing a self-organizing MAS requires simulation due to their emergent and dynamic characteristics. The more complex the self-organizing MAS and their environment, the more complex the simulation to be developed. Therefore, simulation analysis and design could be particular problems which separated but can be derived from analysis and design of the actual systems themselves.

2. Integrating role-modeling method with an existing MAS engineering methodology.

   Another possible future direction is to integrate role-modeling method with an existing MAS engineering methodology. In this direction, an existing engineering methodology is extended to allow designers to use role-modeling method with the methodology. Alternatively, role-modeling method should be extended and adjusted such that it becomes compatible with an existing MAS engineering methodology.

3. Considering other characteristics of self-organizing systems in MAS development.

   The core characteristics of self-organizing systems, such as autonomous, decentralized, dynamic, and openness, have been considered in this thesis and supported in role-modeling method. However, more complex self-organizing systems may have more complex features. For example, some self-organization mechanisms include the capability to learn form its environment and change their behavior as a result of this learning process. In some extent, the ability to change agent behavior has been supported by role-modeling method by allowing the designers to specify some possible roles. More complex self-organizing systems may require the adaptation of agent behavior that has not been expected by the designers. For example, a robot may decide to add a new sensor or a new algorithm to perform an action. Such advance capability may be realized in implementation level. However, engineering methodology should support the designers to properly design the capabilities.

Publications

Journal

Kemas Muslim Lhaksmana, Yohei Murakami, and Toru Ishida. Analysis of large-scale service network tolerance to cascading failure. In *IEEE Internet of Things Journal*. vol.PP, no.99, pp.1-13. 2016.

Conferences

Kemas Muslim Lhaksmana, Yohei Murakami, and Toru Ishida. Cascading failure tolerance in large-scale service networks. In *Services Computing (SCC), IEEE International Conference on*. pp. 1-8. 2015.

Kemas Muslim Lhaksmana, Yohei Murakami, and Toru Ishida. Role modeling for adaptive multiagent systems engineering. In *Web Intelligence (WI) and Intelligent Agent Technologies (IAT), IEEE/WIC/ACM International Joint Conferences on*. pp. 287-292. 2013.

Other Publications

Istikmal, Yuliant Sibaroni, **Kemas Muslim Lhaksmana**, Ratna Mayasari, Tody Ariefianto Wibowo, Ridha Muldina, and Tengku Ahmad Riza. RI-FASKES geographic information system based on web. In *Electrical Power, Electronics, Communications, Controls & Informatics International Seminar (EECCiS)*. pp. E10-1–E10-6. 2012.

Kemas Muslim Lhaksmana. Multiagent systems research and applications in distributed systems. In *National Computation Conference (SNAKOM)*. pp. 183-186. 2012. (In Indonesian)

Izzatul Ummah, Fitriyani, **Kemas Muslim Lhaksmana**, and Tri Brotoharsono. A design of parallel algorithm for routing based on ant colony optimization. In *National Computation Conference (SNAKOM)*. pp. 90-93. 2012. (In Indonesian)

Kemas Muslim Lhaksmana. Decision table to constraints generation in rules-based application. In *National Conference on ICT-M (KNIP)*. pp. 307-311. 2011.

Bibliography

[Cis, 2015] (2015). Cisco visual networking index: Global mobile data traffic forecast update, 20142019. Technical report, Cisco.

[GSM, 2015] (2015). The mobile economy 2015. Technical report, GSMA.

[Abdallah and Lesser, 2004] Abdallah, S. and Lesser, V. (2004). Organization-based cooperative coalition formation. In *Intelligent Agent Technology, 2004.(IAT 2004). Proceedings. IEEE/WIC/ACM International Conference on*, pages 162–168. IEEE.

[Abdallah and Lesser, 2007] Abdallah, S. and Lesser, V. (2007). Multiagent reinforcement learning and self-organization in a network of agents. In *Proceedings of the 6th international joint conference on Autonomous agents and multiagent systems*, AAMAS '07, pages 39:1–39:8, New York, NY, USA. ACM.

[Abdelwahab et al., 2014] Abdelwahab, S., Hamdaoui, B., Guizani, M., and Rayes, A. (2014). Enabling smart cloud services through remote sensing: An Internet of everything enabler. *Internet of Things Journal, IEEE*, 1(3):276–288.

[Albert, 2005] Albert, R. (2005). Scale-free networks in cell biology. *Journal of cell science*, 118(21):4947–4957.

[Albert et al., 2000] Albert, R., Jeong, H., and Barabási, A.-L. (2000). Error and attack tolerance of complex networks. *nature*, 406(6794):378–382.

[Aliu et al., 2013] Aliu, O., Imran, A., Imran, M., and Evans, B. (2013). A survey of self organisation in future cellular networks. *Communications Surveys Tutorials, IEEE*, 15(1):336–361.

[Armbrust et al., 2009] Armbrust, M., Fox, O., Griffith, R., Joseph, A. D., Katz, Y., Konwinski, A., Lee, G., Patterson, D., Rabkin, A., Stoica, I., and Zaharia, M. (2009). Above the clouds: A Berkeley view of cloud computing. Technical report, University of California at Berkeley.

[Ash and Newth, 2007] Ash, J. and Newth, D. (2007). Optimizing complex networks for resilience against cascading failure. *Physica A: Statistical Mechanics and its Applications*, 380:673–683.

[Ashby, 1962] Ashby, W. (1962). Principles of the self-organizing system. *Principles of Self-organization: Transactions of the University of Illinois Symposium*, pages 255–278.

[Ashby, 1947] Ashby, W. R. (1947). Principles of the self-organizing dynamic system. *The Journal of General Psychology*, 37(2):125–128.

[Ashton, 2009] Ashton, K. (2009). That Internet of Things thing. *RFiD Journal*, 22(7):97–114.

[Atzori et al., 2010] Atzori, L., Iera, A., and Morabito, G. (2010). The Internet of Things: a survey. *Computer networks*, 54(15):2787–2805.

[Atzori et al., 2012] Atzori, L., Iera, A., Morabito, G., and Nitti, M. (2012). The Social Internet of Things (SIoT) when social networks meet the Internet of Things: Concept, architecture and network characterization. *Computer Networks*, 56(16):3594 – 3608.

[Barabási and Albert, 1999] Barabási, A.-L. and Albert, R. (1999). Emergence of scaling in random networks. *science*, 286(5439):509–512.

[Barabási et al., 1999] Barabási, A.-L., Albert, R., and Jeong, H. (1999). Mean-field theory for scale-free random networks. *Physica A: Statistical Mechanics and its Applications*, 272(1):173–187.

[Barabási et al., 2000] Barabási, A.-L., Albert, R., and Jeong, H. (2000). Scale-free characteristics of random networks: the topology of the world-wide web. *Physica A: Statistical Mechanics and its Applications*, 281(1):69–77.

[Barabási et al., 2002] Barabási, A.-L., Jeong, H., Néda, Z., Ravasz, E., Schubert, A., and Vicsek, T. (2002). Evolution of the social network of scientific collaborations. *Physica A: Statistical Mechanics and its Applications*, 311(3):590–614.

[Bernon et al., 2005] Bernon, C., Camps, V., Gleizes, M. P., and Picard, G. (2005). Engineering adaptive multi-agent systems: The ADELFE methodology. In Sellers, B. H. and Giorgini, P., editors, *Agent-Oriented Methodologies*, pages 172–202. Idea Group Pub, NY, USA.

[Bernon et al., 2003] Bernon, C., Gleizes, M.-P., Peyruqueou, S., and Picard, G. (2003). ADELFE: a methodology for adaptive multi-agent systems engineering. In *Proceedings of the 3rd international conference on*

Engineering societies in the agents world III, ESAW'02, pages 156–169, Berlin, Heidelberg. Springer-Verlag.

[Biddle, 1986] Biddle, B. J. (1986). Recent development in role theory. *Annual review of sociology*, pages 67–92.

[Bollobás et al., 2003] Bollobás, B., Borgs, C., Chayes, J., and Riordan, O. (2003). Directed scale-free graphs. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '03, pages 132–139, Philadelphia, PA, USA. Society for Industrial and Applied Mathematics.

[Bordini and Hübner, 2006] Bordini, R. and Hübner, J. (2006). BDI agent programming in AgentSpeak using Jason. In Toni, F. and Torroni, P., editors, *Computational Logic in Multi-Agent Systems*, volume 3900 of *Lecture Notes in Computer Science*, pages 143–164. Springer Berlin Heidelberg.

[Bordini et al., 2006] Bordini, R. H., Braubach, L., Dastani, M., Seghrouchni, A. E. F., Gomez-Sanz, J. J., Leite, J., O'Hare, G., Pokahr, A., and Ricci, A. (2006). A survey of programming languages and platforms for multi-agent systems. *Informatica*, 30(1).

[Bordini et al., 2007] Bordini, R. H., Hübner, J. F., and Wooldridge, M. (2007). *Programming Multi-Agent Systems in AgentSpeak Using Jason (Wiley Series in Agent Technology)*. John Wiley & Sons.

[Bresciani et al., 2004] Bresciani, P., Perini, A., Giorgini, P., Giunchiglia, F., and Mylopoulos, J. (2004). Tropos: an agent-oriented software de-

velopment methodology. *Autonomous Agents and Multi-Agent Systems*, 8(3):203–236.

[Capera et al., 2003] Capera, D., George, J.-P., Gleizes, M.-P., and Glize, P. (2003). The AMAS theory for complex problem solving based on self-organizing cooperative agents. In *Enabling Technologies: Infrastructure for Collaborative Enterprises, 2003. WET ICE 2003. Proceedings. Twelfth IEEE International Workshops on*, WETICE '03, pages 383–388, Washington, DC, USA. IEEE Computer Society.

[Catarinucci et al., 2015] Catarinucci, L., de Donno, D., Mainetti, L., Palano, L., Patrono, L., Stefanizzi, M., and Tarricone, L. (2015). An IoT-aware architecture for smart healthcare systems. *Internet of Things Journal, IEEE*, 2(6):515–526.

[Chen and Shi, 2004] Chen, Q. and Shi, D. (2004). The modeling of scale-free networks. *Physica A: Statistical Mechanics and its Applications*, 335(1):240–248.

[Cheng et al., 2009] Cheng, B., Lemos, R., Giese, H., Inverardi, P., Magee, J., Andersson, J., Becker, B., Bencomo, N., Brun, Y., Cukic, B., Marzo Serugendo, G., Dustdar, S., Finkelstein, A., Gacek, C., Geihs, K., Grassi, V., Karsai, G., Kienle, H., Kramer, J., Litoiu, M., Malek, S., Mirandola, R., Mller, H., Park, S., Shaw, M., Tichy, M., Tivoli, M., Weyns, D., and Whittle, J. (2009). Software engineering for self-adaptive systems: A research roadmap. In Cheng, B., Lemos, R., Giese, H., Inverardi, P., and Magee, J., editors, *Software Engineering for Self-Adaptive Systems*, volume 5525 of *Lecture Notes in Computer Science*, pages 1–26. Springer, Berlin, Heidelberg.

[Cirani et al., 2014] Cirani, S., Davoli, L., Ferrari, G., Leone, R., Medagliani, P., Picone, M., and Veltri, L. (2014). A scalable and self-configuring architecture for service discovery in the Internet of Things. *Internet of Things Journal, IEEE*, 1(5):508–521.

[Corkill and Lesser, 1983] Corkill, D. D. and Lesser, V. R. (1983). The use of meta-level control for coordination in a distributed problem solving network. In *Proceedings of the Eighth international joint conference on Artificial intelligence*, volume 2 of *IJCAI'83*, pages 748–756, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.

[Cossentino, 2003] Cossentino, M. (2003). *MAS meta-model in PASSI*. Istituto di Calcolo e Reti ad Alte Prestazioni, Palermo, Italy.

[Cossentino, 2005] Cossentino, M. (2005). From requirements to code with the PASSI methodology. In Sellers, B. H. and Giorgini, P., editors, *Agent-Oriented Methodologies*, pages 79–106. Idea Group Pub, NY, USA.

[Cossentino et al., 2005] Cossentino, M., Gaglio, S., Sabatucci, L., and Seidita, V. (2005). The PASSI and agile PASSI MAS meta-models compared with a unifying proposal. In *Proceedings of the 4th international Central and Eastern European conference on Multi-Agent Systems and Applications*, CEEMAS'05, pages 183–192, Berlin, Heidelberg. Springer-Verlag.

[Crucitti et al., 2004] Crucitti, P., Latora, V., and Marchiori, M. (2004). Model for cascading failures in complex networks. *Physical Review E*, 69(4):045104.

[Dalpiaz et al., 2008] Dalpiaz, F., Ali, R., Asnar, Y., Bryl, V., and Giorgini,

P. (2008). Applying Tropos to socio-technical system design and runtime configuration. In *Proceeding of the Italian workshop Dagli OGGETTI agli AGENTI*, WOA'08.

[Dalpiaz et al., 2009] Dalpiaz, F., Giorgini, P., and Mylopoulos, J. (2009). An architecture for requirements-driven self-reconfiguration. In *Proceedings of the 21st International Conference on Advanced Information Systems Engineering*, CAiSE '09, pages 246–260, Berlin, Heidelberg. Springer-Verlag.

[de Lemos et al., 2013] de Lemos, R., Giese, H., Mller, H., Shaw, M., Andersson, J., Litoiu, M., Schmerl, B., Tamura, G., Villegas, N., Vogel, T., Weyns, D., Baresi, L., Becker, B., Bencomo, N., Brun, Y., Cukic, B., Desmarais, R., Dustdar, S., Engels, G., Geihs, K., Gschka, K., Gorla, A., Grassi, V., Inverardi, P., Karsai, G., Kramer, J., Lopes, A., Magee, J., Malek, S., Mankovskii, S., Mirandola, R., Mylopoulos, J., Nierstrasz, O., Pezzè, M., Prehofer, C., Schäfer, W., Schlichting, R., Smith, D., Sousa, J., Tahvildari, L., Wong, K., and Wuttke, J. (2013). Software engineering for self-adaptive systems: A second research roadmap. In de Lemos, R., Giese, H., Mller, H., and Shaw, M., editors, *Software Engineering for Self-Adaptive Systems II*, volume 7475 of *Lecture Notes in Computer Science*, pages 1–32. Springer Berlin Heidelberg.

[De Souza et al., 2008] De Souza, L. M. S., Spiess, P., Guinard, D., KÖhler, M., Karnouskos, S., and Savio, D. (2008). Socrades: A web service based shop floor integration infrastructure. In Floerkemeier, C., Langheinrich, M., Fleisch, E., Mattern, F., and Sarma, S., editors, *The Internet of Things*, volume 4952 of *Lecture Notes in Computer Science*, pages 50–67. Springer Berlin Heidelberg.

[del Val et al., 2014] del Val, E., Rebollo, M., and Botti, V. (2014). Combination of self-organization mechanisms to enhance service discovery in open systems. *Information Sciences*, 279:138–162.

[DeLoach, 2009] DeLoach, S. A. (2009). OMACS a framework for adaptive, complex systems. In Dignum, V., editor, *Handbook of Research on Multi-Agent Systems: Semantics and Dynamics of Organizational Models*, pages 76–98. IGI, Hershey, PA, USA.

[DeLoach and García-Ojeda, 2010] DeLoach, S. A. and García-Ojeda, J. C. (2010). O-MASE: a customisable approach to designing and building complex, adaptive multi-agent systems. *Int. J. Agent-Oriented Softw. Eng.*, 4(3):244–280.

[DeLoach et al., 2001] DeLoach, S. A., Wood, M. F., and Sparkman, C. H. (2001). Multiagent systems engineering. *International Journal of Software Engineering and Knowledge Engineering*, 11(03):231–258.

[Deneubourg et al., 1990] Deneubourg, J.-L., Aron, S., Goss, S., and Pasteels, J. (1990). The self-organizing exploratory pattern of the argentine ant. *Journal of Insect Behavior*, 3:159–168.

[Di Marzo Serugendo et al., 2004] Di Marzo Serugendo, G., Foukia, N., Hassas, S., Karageorgos, A., Mostfaoui, S., Rana, O., Ulieru, M., Valckenaers, P., and Van Aart, C. (2004). Self-organisation: Paradigms and applications. In Di Marzo Serugendo, G., Karageorgos, A., Rana, O., and Zambonelli, F., editors, *Engineering Self-Organising Systems*, volume 2977 of *Lecture Notes in Computer Science*, pages 1–19. Springer Berlin Heidelberg.

[Di Marzo Serugendo et al., 2005] Di Marzo Serugendo, G., Gleizes, M.-P., and Karageorgos, A. (2005). Self-organisation in multi-agent system. Technical report, Institut de Recherche en Informatique de Toulouse.

[Ding et al., 2013] Ding, Y., Jin, Y., Ren, L., and Hao, K. (2013). An intelligent self-organization scheme for the internet of things. *Computational Intelligence Magazine, IEEE*, 8(3):41–53.

[Dorogovtsev and Mendes, 2002] Dorogovtsev, S. N. and Mendes, J. F. (2002). Evolution of networks. *Advances in physics*, 51(4):1079–1187.

[Dorogovtsev et al., 2000] Dorogovtsev, S. N., Mendes, J. F. F., and Samukhin, A. N. (2000). Structure of growing networks with preferential linking. *Physical Review Letters*, 85(21):4633.

[Ebel et al., 2002] Ebel, H., Mielsch, L.-I., and Bornholdt, S. (2002). Scale-free topology of e-mail networks. *arXiv preprint cond-mat/0201476*.

[Erdős and Rényi, 1960] Erdős, P. and Rényi, A. (1960). On the evolution of random graphs. *Magyar Tud. Akad. Mat. Kutató Int. Közl*, 5:17–61.

[Evans, 2011] Evans, D. (2011). The Internet of Things: How the next evolution of the internet is changing everything. *CISCO white paper*.

[Faloutsos et al., 1999] Faloutsos, M., Faloutsos, P., and Faloutsos, C. (1999). On power-law relationships of the internet topology. *SIGCOMM Comput. Commun. Rev.*, 29(4):251–262.

[Feng et al., 2014] Feng, Z., Lan, B., Zhang, Z., and Chen, S. (2014). A study of semantic web services network. *The Computer Journal*.

[Ferber and Gutknecht, 1998] Ferber, J. and Gutknecht, O. (1998). A meta-model for the analysis and design of organizations in multi-agent systems. In *Proceedings of the 3rd International Conference on Multi Agent Systems*, ICMAS '98, pages 128–, Washington, DC, USA. IEEE Computer Society.

[Ferber et al., 2004] Ferber, J., Gutknecht, O., and Michel, F. (2004). From agents to organizations: An organizational view of multi-agent systems. In Giorgini, P., Mller, J., and Odell, J., editors, *Agent-Oriented Software Engineering IV*, volume 2935 of *Lecture Notes in Computer Science*, pages 214–230. Springer-Verlag, Berlin, Heidelberg.

[Finin et al., 1994] Finin, T., Fritzson, R., McKay, D., and McEntire, R. (1994). KQML as an agent communication language. In *Proceedings of the Third International Conference on Information and Knowledge Management*, CIKM '94, pages 456–463, New York, NY, USA. ACM.

[Ford and Fulkerson, 1956] Ford, L. and Fulkerson, D. (1956). Maximal flow through a network. *Can. J. Math.*, 8:399–404.

[García-Magariño et al., 2009a] García-Magariño, I., Fuentes-Fernández, R., and Gómez-Sanz, J. J. (2009a). Ingenias development process assisted with chains of transformations. In Cabestany, J., Sandoval, F., Prieto, A., and Corchado, J. M., editors, *Bio-Inspired Systems: Computational and Ambient Intelligence*, volume 5517 of *Lecture Notes in Computer Science*, pages 514–521. Springer-Verlag, Berlin, Heidelberg.

[García-Magariño et al., 2009b] García-Magariño, I., Gómez-Sanz, J., and Fuentes-Fernández, R. (2009b). INGENIAS development assisted with model transformation by-example: A practical case. In Demazeau, Y.,

Pavón, J., Corchado, J. M., and Bajo, J., editors, *7th International Conference on Practical Applications of Agents and Multi-Agent Systems (PAAMS 2009)*, volume 55 of *Advances in Intelligent and Soft Computing*, pages 40–49. Springer-Verlag, Berlin, Heidelberg.

[García-Magariño et al., 2011] García-Magariño, I., Gómez-Sanz, J. J., and Fuentes-Fernández, R. (2011). Model transformations for improving multi-agent system development in INGENIAS. In Gleizes, M.-P. and Gómez-Sanz, J. J., editors, *Agent-Oriented Software Engineering X*, volume 6038 of *Lecture Notes in Computer Science*, pages 51–65. Springer-Verlag, Berlin, Heidelberg.

[Gaston and desJardins, 2005] Gaston, M. E. and desJardins, M. (2005). Agent-organized networks for dynamic team formation. In *Proceedings of the fourth international joint conference on Autonomous agents and multiagent systems*, AAMAS '05, pages 230–237, New York, NY, USA. ACM.

[Geyik et al., 2013] Geyik, S., Szymanski, B., and Zerfos, P. (2013). Robust dynamic service composition in sensor networks. *Services Computing, IEEE Transactions on*, 6(4):560–572.

[Geyik et al., 2010] Geyik, S., Szymanski, B., Zerfos, P., and Verma, D. (2010). Dynamic composition of services in sensor networks. In *Services Computing (SCC), 2010 IEEE International Conference on*, pages 242–249.

[Ghosh et al., 2007] Ghosh, D., Sharman, R., Rao, H. R., and Upadhyaya, S. (2007). Self-healing systems survey and synthesis. *Decision Support*

Systems, 42(4):2164 – 2185. Decision Support Systems in Emerging Economies.

[Giorgini et al., 2008] Giorgini, P., Mylopoulos, J., Penserini, L., Perini, A., and Susi, A. (2008). Tropos at the age of eight: On-going research at fbk, unitn and ut. In Castro, J., Franch, X., Perini, A., and Yu, E., editors, *Proceedings of the Third International i* Workshop (iStar'08)*, volume 322, pages 83–89, Recife, Brazil. Morgan Kaufmann Publishers Inc.

[Gómez-Sanz et al., 2010] Gómez-Sanz, J. J., Fernández, C. R., and Arroyo, J. (2010). Model driven development and simulations with the INGENIAS agent framework. *Simulation Modelling Practice and Theory*, 18(10):1468–1482. Simulation-based Design and Evaluation of Multi-Agent Systems.

[Gordon, 1996] Gordon, D. M. (1996). The organization of work in social insect colonies. *Nature*, 380(6570):121–124.

[Guinard et al., 2010a] Guinard, D., Trifa, V., Karnouskos, S., Spiess, P., and Savio, D. (2010a). Interacting with the SOA-based Internet of Things: Discovery, query, selection, and on-demand provisioning of web services. *Services Computing, IEEE Transactions on*, 3(3):223–235.

[Guinard et al., 2010b] Guinard, D., Trifa, V., and Wilde, E. (2010b). A resource oriented architecture for the Web of Things. In *Internet of Things (IOT), 2010*, pages 1–8.

[Gupta et al., 2008] Gupta, S., Ray, A., Sarkar, S., and Yasar, M. (2008). Fault detection and isolation in aircraft gas turbine engines. part 1: Un-

derlying concept. *Proceedings of the Institution of Mechanical Engineers, Part G: Journal of Aerospace Engineering*, 222(3):307–318.

[Haller, 2010] Haller, S. (2010). The things in the Internet of Things. In *Internet of Things Conference*.

[Han et al., 2015] Han, S., Lee, G. M., Crespi, N., Luong, N. V., Heo, K., Brut, M., and Gatellier, P. (2015). DPWSim: A devices profile for web services (dpws) simulator. *Internet of Things Journal, IEEE*, 2(3):221–229.

[Hassas et al., 2006] Hassas, S., Di Marzo-Serugendo, G., Karageorgos, A., and Castelfranchi, C. (2006). On self-organising mechanisms from social, business and economic domains. *Informatica*, 30(1):63–71.

[Hoogendoorn, 2007] Hoogendoorn, M. (2007). Adaptation of organizational models for multi-agent systems based on max flow networks. In *Proceedings of the 20th international joint conference on Artifical intelligence*, IJCAI'07, pages 1321–1326, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.

[Horling et al., 2001] Horling, B., Benyo, B., and Lesser, V. (2001). Using self-diagnosis to adapt organizational structures. In *Proceedings of the fifth international conference on Autonomous agents*, AGENTS '01, pages 529–536, New York, NY, USA. ACM.

[Horling et al., 1999] Horling, B., Lesser, V., Vincent, R., Wagner, T., Raja, A., Zhang, S., Decker, K., and Garvey, A. (1999). *The TAEMS white paper*. Multi-Agent Systems Lab University of Massachusetts, Amherst, MA, USA.

[Huang et al., 2012] Huang, K., Fan, Y., and Tan, W. (2012). An empirical study of ProgrammableWeb: A network analysis on a service-mashup system. In *Web Services (ICWS), 2012 IEEE 19th International Conference on*, pages 552–559.

[Huang et al., 2013] Huang, K., Yao, J., Fan, Y., Tan, W., Nepal, S., Ni, Y., and Chen, S. (2013). Mirror, mirror, on the web, which is the most reputable service of them all? In *Service-Oriented Computing*, pages 343–357. Springer.

[Hübner et al., 2007] Hübner, J. F., Sichman, J. S., and Boissier, O. (2007). Developing organised multiagent systems using the MOISE+ model: programming issues at the system and agent levels. *International Journal of Agent-Oriented Software Engineering*, 1(3-4):370–395.

[Im et al., 2013] Im, J., Kim, S., and Kim, D. (2013). IoT Mashup as a Service: Cloud-based mashup service for the Internet of Things. In *Services Computing (SCC), 2013 IEEE International Conference on*, pages 462–469.

[Ishida, 2002] Ishida, T. (2002). Q: A scenario description language for interactive agents. *Computer*, 35(11):42–47.

[Ishida et al., 1992] Ishida, T., Gasser, L., and Yokoo, M. (1992). Organization self-design of distributed production systems. *IEEE Trans. on Knowl. and Data Eng.*, 4(2):123–134.

[Ishida et al., 2011] Ishida, T., Murakami, Y., and Lin, D. (2011). The Language Grid: Service-oriented approach to sharing language resources.

In Ishida, T., editor, *The Language Grid*, Cognitive Technologies, pages 3–17. Springer Berlin Heidelberg.

[Issarny et al., 2011] Issarny, V., Georgantas, N., Hachem, S., Zarras, A., Vassiliadist, P., Autili, M., Gerosa, M., and Hamida, A. (2011). Service-oriented middleware for the future internet: state of the art and research directions. *Journal of Internet Services and Applications*, 2(1):23–45.

[Jennings et al., 1998] Jennings, N. R., Sycara, K., and Wooldridge, M. (1998). A roadmap of agent research and development. *Autonomous agents and multi-agent systems*, 1(1):7–38.

[Kamboj and Decker, 2006] Kamboj, S. and Decker, K. S. (2006). Organizational self-design in semi-dynamic environments. In *Proceedings of the fifth international joint conference on Autonomous agents and multiagent systems*, AAMAS '06, pages 335–337, New York, NY, USA. ACM.

[Kawsar et al., 2008] Kawsar, F., Fujinami, K., and Nakajima, T. (2008). Prottoy middleware platform for smart object systems. *International Journal of Smart Home*, 2(3):1–18.

[Kendall, 1999] Kendall, E. (1999). Role modelling for agent system analysis, design, and implementation. In *Agent Systems and Applications, 1999 and Third International Symposium on Mobile Agents. Proceedings. First International Symposium on*, pages 204–218.

[Kendall, 2001] Kendall, E. (2001). Agent software engineering with role modelling. In Ciancarini, P. and Wooldridge, M., editors, *Agent-Oriented Software Engineering*, volume 1957 of *Lecture Notes in Computer Science*, pages 163–169. Springer, Berlin, Heidelberg.

[Kendall, 1998] Kendall, E. A. (1998). Agent roles and role models: New abstractions for intelligent agent system analysis and design. In *Proceeedings of the International Workshop on Intelligent Agents in Information and Process Management*, pages 204–218.

[Kephart and Chess, 2003] Kephart, J. O. and Chess, D. M. (2003). The vision of autonomic computing. *Computer*, 36(1):41–50.

[Kil et al., 2009] Kil, H., Oh, S.-C., Elmacioglu, E., Nam, W., and Lee, D. (2009). Graph theoretic topological analysis of web service networks. *World Wide Web*, 12(3):321–343.

[Kinney et al., 2005] Kinney, R., Crucitti, P., Albert, R., and Latora, V. (2005). Modeling cascading failures in the north american power grid. *The European Physical Journal B-Condensed Matter and Complex Systems*, 46(1):101–107.

[Kochhal et al., 2003] Kochhal, M., Schwiebert, L., and Gupta, S. (2003). Role-based hierarchical self organization for wireless ad hoc sensor networks. In *Proceedings of the 2nd ACM International Conference on Wireless Sensor Networks and Applications*, WSNA '03, pages 98–107, New York, NY, USA. ACM.

[Kostakos et al., 2013] Kostakos, V., Ojala, T., and Juntunen, T. (2013). Traffic in the smart city: Exploring city-wide sensing for traffic control center augmentation. *IEEE Internet Computing*, 17(6):22–29.

[Kota et al., 2012] Kota, R., Gibbins, N., and Jennings, N. R. (2012). Decentralized approaches for self-adaptation in agent organizations. *ACM Trans. Auton. Adapt. Syst.*, 7(1):1:1–1:28.

[Kraus et al., 2003] Kraus, S., Shehory, O., and Taase, G. (2003). Coalition formation with uncertain heterogeneous information. In *Proceedings of the second international joint conference on Autonomous agents and multiagent systems*, pages 1–8. ACM.

[Kuwabara et al., 1995] Kuwabara, K., Ishida, T., and Osato, N. (1995). AgenTalk: Coordination protocol description for multiagent systems. In *ICMAS*, pages 455–461.

[LeHong and Fenn, 2013] LeHong, H. and Fenn, J. (2013). Hype cycle for emerging technologies, 2013. *Gartner*.

[Lesser and Corkill, 1980] Lesser, V. R. and Corkill, D. D. (1980). Cooperative distributed problem solving and organizational self-design. In Davis, R., editor, *Report on the Workshop on Distributed AI*, page 11. MIT AI Laboratory, Boston, MA, USA.

[Lhaksmana et al., 2013] Lhaksmana, K., Murakami, Y., and Ishida, T. (2013). Role modeling for adaptive multiagent systems engineering. In *Web Intelligence (WI) and Intelligent Agent Technologies (IAT), 2013 IEEE/WIC/ACM International Joint Conferences on*, volume 2, pages 287–292.

[Lhaksmana et al., 2015] Lhaksmana, K. M., Murakami, Y., and Ishida, T. (2015). Cascading failure tolerance in large-scale service networks. In *Services Computing (SCC), 2015 IEEE International Conference on*, pages 1–8.

[Lillis et al., 2013] Lillis, D., Collier, R. W., and Jordan, H. R. (2013). Evaluation of a conversation management toolkit for multi agent program-

ming. In Dastani, M., Hübner, J. F., and Logan, B., editors, *Programming Multi-Agent Systems: 10th International Workshop, ProMAS 2012, Valencia, Spain, June 5, 2012, Revised Selected Papers*, pages 90–107. Springer Berlin Heidelberg, Berlin, Heidelberg.

[Little, 2002] Little, R. G. (2002). Controlling cascading failure: Understanding the vulnerabilities of interconnected infrastructures. *Journal of Urban Technology*, 9(1):109–123.

[Lizcano et al., 2008] Lizcano, D., Jiménez, M., Soriano, J., Cantera, J. M., Reyes, M., Hierro, J. J., Garijo, F., and Tsouroulas, N. (2008). Leveraging the upcoming Internet of Services through an open user-service front-end framework. In *Towards a Service-Based Internet*, pages 147–158. Springer.

[Middleton et al., 2014] Middleton, P., Trully, J., Brant, K. F., Goodness, E., Gupta, A., Hines, J. F., Koslowski, T., McIntyre, A., and Tratz-Ryan, B. (2014). Forecast: Internet of Things, endpoints and associated services, worldwide, 2014. *Gartner*.

[Miorandi et al., 2012] Miorandi, D., Sicari, S., Pellegrini, F. D., and Chlamtac, I. (2012). Internet of Things: Vision, applications and research challenges. *Ad Hoc Networks*, 10(7):1497–1516.

[Moreno-Vozmediano et al., 2013] Moreno-Vozmediano, R., Montero, R., and Llorente, I. (2013). Key challenges in cloud computing: Enabling the future Internet of Services. *Internet Computing, IEEE*, 17(4):18–25.

[Müller and Fischer, 2014] Müller, J. and Fischer, K. (2014). Application impact of multi-agent systems and technologies: A survey. In Shehory,

O. and Sturm, A., editors, *Agent-Oriented Software Engineering*, pages 27–53. Springer Berlin Heidelberg.

[Murakami et al., 2012] Murakami, Y., Tanaka, M., Lin, D., and Ishida, T. (2012). Service grid federation architecture for heterogeneous domains. In *Services Computing (SCC), 2012 IEEE Ninth International Conference on*, pages 539–546.

[Nichols, 1990] Nichols, D. (1990). Coke machine history. *C. Everhart," Interesting uses of networking*.

[Oh et al., 2008] Oh, S.-C., Lee, D., and Kumara, S. R. (2008). Effective web service composition in diverse and large-scale service networks. *Services Computing, IEEE Transactions on*, 1(1):15–32.

[Omicini, 2001] Omicini, A. (2001). SODA: Societies and infrastructures in the analysis and design of agent-based systems. In Ciancarini, P. and Wooldridge, M., editors, *Agent-oriented software engineering*, AOSE'00, pages 185–193, Berlin, Heidelberg. Springer, Springer-Verlag.

[Ortiz et al., 2014] Ortiz, A., Hussein, D., Park, S., Han, S., and Crespi, N. (2014). The cluster between internet of things and social networks: Review and research challenges. *Internet of Things Journal, IEEE*, 1(3):206–215.

[Padgham and Winikoff, 2003a] Padgham, L. and Winikoff, M. (2003a). Prometheus: a methodology for developing intelligent agents. In Giunchiglia, F., Odell, J., and Wei, G., editors, *Agent-Oriented Software*

Engineering III, volume 2585 of *Lecture Notes in Computer Science*, pages 174–185. Springer, Berlin, Heidelberg.

[Padgham and Winikoff, 2003b] Padgham, L. and Winikoff, M. (2003b). Prometheus: a methodology for developing intelligent agents. In Giunchiglia, F., Odell, J., and Wei, G., editors, *Agent-Oriented Software Engineering III*, volume 2585 of *Lecture Notes in Computer Science*, pages 174–185. Springer Berlin Heidelberg.

[Papazoglou et al., 2008] Papazoglou, M. P., Traverso, P., Dustdar, S., and Leymann, F. (2008). Service-oriented computing: a research roadmap. *International Journal of Cooperative Information Systems*, 17(02):223–255.

[Paraschos et al., 2012] Paraschos, A., Spanoudakis, N. I., and Lagoudakis, M. G. (2012). Model-driven behavior specification for robotic teams. In *Proceedings of the 11th International Conference on Autonomous Agents and Multiagent Systems - Volume 1*, AAMAS '12, pages 171–178, Richland, SC. International Foundation for Autonomous Agents and Multiagent Systems.

[Parunak and Brueckner, 2011] Parunak, H. and Brueckner, S. (2011). Software engineering for self-organizing systems. In Weyns, D. and Müller, J. P., editors, *Proceedings of the 12th International Workshop on Agent-Oriented Software Engineering*, AOSE'11, pages 1–21.

[Pavón and Gómez-Sanz, 2003] Pavón, J. and Gómez-Sanz, J. (2003). Agent oriented software engineering with INGENIAS. In *Proceedings of the 3rd Central and Eastern European conference on Multi-agent*

systems, CEEMAS'03, pages 394–403, Berlin, Heidelberg. Springer-Verlag.

[Pavón et al., 2005] Pavón, J., Gómez-Sanz, J. J., and Fuentes, R. (2005). The INGENIAS methodology and tools. In Hendersen-Sellers, B. and Giorgini, P., editors, *Agent-oriented Methodologies*, pages 236–276. IGI, Hershey, PA, USA.

[Qureshi and Perini, 2008] Qureshi, N. A. and Perini, A. (2008). Towards seamless adaptation: An agent-oriented approach. In *Proceedings of the 2008 Second IEEE International Conference on Self-Adaptive and Self-Organizing Systems*, SASO '08, pages 471–472, Washington, DC, USA. IEEE Computer Society.

[Rao, 1996] Rao, A. (1996). AgentSpeak(L): BDI agents speak out in a logical computable language. In Van de Velde, W. and Perram, J., editors, *Agents Breaking Away*, volume 1038 of *Lecture Notes in Computer Science*, pages 42–55. Springer Berlin Heidelberg.

[Rao and Georgeff, 1998] Rao, A. S. and Georgeff, M. P. (1998). Modeling rational agents with a BDI-architecture. In Huhns, M. N. and Singh, M. P., editors, *Readings in agents*, pages 317–328. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[Raychoudhury et al., 2013] Raychoudhury, V., Cao, J., Kumar, M., and Zhang, D. (2013). Middleware for pervasive computing: A survey. *Pervasive and Mobile Computing*, 9(2):177 – 200. Special Section: Mobile Interactions with the Real World.

[Ren et al., 2007] Ren, S., Yu, Y., Chen, N., Tsai, J. J.-P., and Kwiat, K.

(2007). The role of roles in supporting reconfigurability and fault localizations for open distributed and embedded systems. *ACM Trans. Auton. Adapt. Syst.*, 2(3).

[Rinaldi et al., 2001] Rinaldi, S., Peerenboom, J., and Kelly, T. (2001). Identifying, understanding, and analyzing critical infrastructure interdependencies. *Control Systems, IEEE*, 21(6):11–25.

[Rosato et al., 2008] Rosato, V., Issacharoff, L., Tiriticco, F., Meloni, S., Porcellinis, S., and Setola, R. (2008). Modelling interdependent infrastructures using interacting dynamical models. *International Journal of Critical Infrastructures*, 4(1):63–79.

[Rouvoy et al., 2009] Rouvoy, R., Barone, P., Ding, Y., Eliassen, F., Hallsteinsen, S., Lorenzo, J., Mamelli, A., and Scholz, U. (2009). Music: Middleware support for self-adaptation in ubiquitous and service-oriented environments. In *Software engineering for self-adaptive systems*, pages 164–182. Springer.

[Sanchez et al., 2014] Sanchez, L., Muoz, L., Galache, J. A., Sotres, P., Santana, J. R., Gutierrez, V., Ramdhany, R., Gluhak, A., Krco, S., Theodoridis, E., and Pfisterer, D. (2014). SmartSantander: IoT experimentation over a smart city testbed. *Computer Networks*, 61:217 – 238. Special issue on Future Internet Testbeds Part I.

[Schroth and Janner, 2007] Schroth, C. and Janner, T. (2007). Web 2.0 and SOA: Converging concepts enabling the Internet of Services. *IT Professional*, 9(3):36–41.

[Serugendo et al., 2006] Serugendo, G. D. M., Gleizes, M. P., and Kara-

georgos, A. (2006). Self-organisation and emergence in mas: An overview. *Informatica (Slovenia)*, 30(1):45–54.

[Shehory et al., 1998] Shehory, O., Sycara, K., Chalasani, P., and Jha, S. (1998). Agent cloning: an approach to agent mobility and resource allocation. *Communications Magazine, IEEE*, 36(7):58, 63–67.

[Shoham, 1993] Shoham, Y. (1993). Agent-oriented programming. *Artificial Intelligence*, 60(1):51 – 92.

[Smart et al., 2008] Smart, A. G., Amaral, L. A., and Ottino, J. M. (2008). Cascading failure and robustness in metabolic networks. *Proceedings of the National Academy of Sciences*, 105(36):13223–13228.

[Song et al., 2010] Song, Z., Cárdenas, A., and Masuoka, R. (2010). Semantic middleware for the Internet of Things. In *Internet of Things (IOT), 2010*, pages 1–8.

[Spanoudakis and Moraitis, 2008] Spanoudakis, N. and Moraitis, P. (2008). The agent modeling language (amola). In Dochev, D., Pistore, M., and Traverso, P., editors, *Artificial Intelligence: Methodology, Systems, and Applications*, volume 5253 of *Lecture Notes in Computer Science*, pages 32–44. Springer, Berlin, Heidelberg.

[Spanoudakis and Moraitis, 2011] Spanoudakis, N. and Moraitis, P. (2011). Using ASEME methodology for model-driven agent systems development. In *Proceedings of the 11th international conference on Agent-oriented software engineering*, AOSE'10, pages 106–127, Berlin, Heidelberg. Springer-Verlag.

[Stafford-Fraser, 2001] Stafford-Fraser, Q. (2001). On site: The life and times of the first web cam. *Commun. ACM*, 44(7):25–26.

[Stankovic, 2014] Stankovic, J. (2014). Research directions for the internet of things. *Internet of Things Journal, IEEE*, 1(1):3–9.

[Stavropoulos et al., 2013] Stavropoulos, T., Vrakas, D., and Vlahavas, I. (2013). A survey of service composition in ambient intelligence environments. *Artificial Intelligence Review*, 40(3):247–270.

[Stecca et al., 2015] Stecca, M., Moiso, C., Fornasa, M., Baglietto, P., and Maresca, M. (2015). A platform for smart object virtualization and composition. *Internet of Things Journal, IEEE*, 2(6):604–613.

[Steine et al., 2015] Steine, M., Geilen, M., and Basten, T. (2015). A distributed reconfiguration approach for quality-of-service provisioning in dynamic heterogeneous wireless sensor networks. *ACM Trans. Sen. Netw.*, 11(2):34:1–34:41.

[Tanenbaum and Steen, 2006] Tanenbaum, A. S. and Steen, M. v. (2006). *Distributed Systems: Principles and Paradigms (2nd Edition)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

[Teixeira et al., 2011] Teixeira, T., Hachem, S., Issarny, V., and Georgantas, N. (2011). Service oriented middleware for the internet of things: A perspective. In Abramowicz, W., Llorente, I., Surridge, M., Zisman, A., and Vayssire, J., editors, *Towards a Service-Based Internet*, volume 6994 of *Lecture Notes in Computer Science*, pages 220–229. Springer Berlin Heidelberg.

[Tran and Low, 2005] Tran, Q.-N. N. and Low, G. C. (2005). Comparison of ten agent-oriented methodologies. In Sellers, B. H. and Giorgini, P., editors, *Agent-Oriented Methodologies*, pages 341–367. Idea Group Pub, NY, USA.

[Vasseur and Dunkels, 2010] Vasseur, J.-P. and Dunkels, A. (2010). *Interconnecting Smart Objects with IP: The Next Internet*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[Villaverde et al., 2014] Villaverde, B., De Paz Alberola, R., Jara, A., Fedor, S., Das, S., and Pesch, D. (2014). Service discovery protocols for constrained machine-to-machine communications. *Communications Surveys Tutorials, IEEE*, 16(1):41–60.

[Weiser, 1991] Weiser, M. (1991). The computer for the 21st century. *Scientific american*, 265(3):94–104.

[Weyns et al., 2012] Weyns, D., Iftikhar, M., Malek, S., and Andersson, J. (2012). Claims and supporting evidence for self-adaptive systems: A literature study. In *Software Engineering for Adaptive and Self-Managing Systems (SEAMS), 2012 ICSE Workshop on*, pages 89 –98.

[Weyns et al., 2005] Weyns, D., Schelfthout, K., Holvoet, T., and Glorieux, O. (2005). Towards adaptive role selection for behavior-based agents. In Kudenko, D., Kazakov, D., and Alonso, E., editors, *Adaptive Agents and Multi-Agent Systems II*, volume 3394 of *Lecture Notes in Computer Science*, pages 295–312. Springer Berlin Heidelberg.

[Wooldridge et al., 2000] Wooldridge, M., Jennings, N. R., and Kinny, D. (2000). The Gaia methodology for agent-oriented analysis and design.

Autonomous Agents and Multi-Agent Systems, 3(3):285–312.

[Yu, 1997] Yu, E. (1997). Towards modelling and reasoning support for early-phase requirements engineering. In *Requirements Engineering, 1997., Proceedings of the Third IEEE International Symposium on*, pages 226–235.

[Yu and Mylopoulos, 1994] Yu, E. S. K. and Mylopoulos, J. (1994). Understanding ”why” in software process modelling, analysis, and design. In *Proceedings of the 16th international conference on Software engineering*, ICSE '94, pages 159–168, Los Alamitos, CA, USA. IEEE Computer Society Press.

[Zakon, 1997] Zakon, R. H. (1997). Hobbes' internet timeline.

[Zambonelli et al., 2003] Zambonelli, F., Jennings, N. R., and Wooldridge, M. (2003). Developing multiagent systems: The Gaia methodology. *ACM Trans. Softw. Eng. Methodol.*, 12(3):317–370.

[Zanella et al., 2014] Zanella, A., Bui, N., Castellani, A., Vangelista, L., and Zorzi, M. (2014). Internet of things for smart cities. *Internet of Things Journal, IEEE*, 1(1):22–32.

[Zeng et al., 2011] Zeng, D., Guo, S., and Cheng, Z. (2011). The Web of Things: A survey (invited paper). *Journal of Communications*, 6(6).

[Zorzi et al., 2010] Zorzi, M., Gluhak, A., Lange, S., and Bassi, A. (2010). From today's INTRAnet of things to a future INTERnet of things: a wireless- and mobility-related view. *Wireless Communications, IEEE*, 17(6):44–51.