2013-09-13

Verification of Multi-Agent Systems Using AUML Methodology

Mireslami, Seyedehmehrnaz

UNIVERSITY OF CALGARY

Verification of Multi-Agent Systems Using AUML Methodology

by

Seyedehmehrnaz Mireslami

A THESIS

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE

DEGREE OF MASTER OF SCIENCE

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

CALGARY, ALBERTA

SEPTEMBER, 2013

Abstract

Verification of Multi-Agent Systems (MAS) is vital since it results in reducing design costs. Agent UML (AUML) is a methodology for MAS design that is an extension of Unified Modeling Language (UML). Although UML is used for object-oriented designs, AUML can handle the interactions among agents to deal with agent-based designs. In this thesis, AUML is employed for designing MASs and a set of conversion rules is proposed to convert AUML notations into UML diagrams to be used for MAS verification. Emergent behaviour is a critical problem in MASs that leads to unexpected behaviours due to the assumptions of behaviour model synthesis, i.e. overgeneralization. The main contributions of this thesis are: 1) Designing multi-agent systems using AUML methodology and preparing scenarios for verification. 2) Developing a component-level approach for verifying multi-agent systems preventing overgeneralization. 3) Proposing a system-level algorithm to obtain comprehensive system behaviour analysis.

Acknowledgements

I would like to thank all people who helped me in production of this thesis. First, I would like to express my thanks to my supervisor, Dr. Far, for his comments and suggestions in making this thesis. In addition, I appreciate the valuable comments from the committee members. I would like to thank my family which is the most important factor of my success. I wish to thank my mom and dad who always guide me into the way of success in my life. To my lovely and supportive husband, Amin, thanks for making a happy environment for me during the writing of this thesis. Finally, thanks to my sister, Anahita, who has always been a good listener.

Table of Contents

List of Tables

List of Figures

List of Terms

Acronyms:

AIP	:	Agent Interaction Protocol
AOSE	:	Agent Oriented Software Engineering
AUML	:	Agent Unified Modeling Language
BDI	:	Beliefs, Desires and Intentions
CA	:	Communication Act
CEBD	:	Component-level Emergent Behaviour Detection
FIPA	:	Foundation for Intelligent Physical Agents
FSM	:	Finite State Machine
hMSC	:	high-Level Message Sequence Chart
ITU	:	International Telecommunication Union
LTS	:	Labeled Transition System
MAS	:	Multi-Agent System
MaSE	:	Multi-agent Software Engineering
MDTM	:	Model-based Detection and Testing of Multi-agent systems
MSC	:	Message Sequence Chart
MSG	:	Message Sequence Graph
OMG	:	Object Management Group
P2P	:	Peer-to-Peer
SD	:	Sequence Diagram
SEBD	:	System-level Emergent Behaviour Detection
SMS	:	Short Message Service
SV	:	System-level Verification
UML	:	Unified Modeling Language

Sub-scripts:

i	:	Index of agents, messages, labeled transitions systems and words
j	:	Index of agents, messages, labeled transitions systems and words
k	:	Index of agents, messages, labeled transitions systems and words
l	:	Index of agents

Scalars:

A_i	:	Concurrent automata of agent i
C_i	:	Component i
D_i	:	Domain theory for agent i
e	:	An event

LTS$_i$	:	Labeled transition system i
m_i	:	Message i
p	:	A path of messages
q_0	:	Initial state
q_f	:	Final state
q_i	:	State i
S_i	:	Scenario i
w_j	:	A word representing an order of messages
δ_i	:	A transition relation

Sets:

A_i^m	:	Finite state machine for agent i in scenario m
F_i	:	A set of accepting states
L	:	System language
M	:	Set of message sequence diagrams
Q_i	:	A set of states of agent i
S^m	:	A finite set of states
Σ^m	:	The alphabet that is the set of all messages
σ^m	:	A set of transition relations

Functions:

$i!l(c)$	:	A message with content c sent from agent i to agent l
$i?l(c)$	:	A message with content c received by agent i from agent l
$m\|_i[j]$	:	j^{th} message of agent i in scenario m
$q_i^{S_1,S_2}$	:	The i^{th} blended state of state machines of scenarios S_1 and S_2
$v_i\|(\cdot)$	:	Value of a state in state machine of agent i

Definitions:

Agent	:	A component in a multi-agent software system
Behaviour model synthesis	:	Going from scenarios to state machines to analyse system behaviours
Component-level verification	:	Verification by analysing behaviours of each agent individually
Distributed systems	:	A software system that contains several computing resource components
Emergent behaviour	:	Unexpected agent behaviour
GPS Rec.	:	A sensor that collects location data
Identical states	:	Similar states in different state machines

Implied scenario	:	Partial behaviour that is not expected by the scenario specifications
Message	:	Interaction among agents
Multi-agent system	:	A software system that contains several computing resource agents that act independently
Overgeneralization	:	Unnecessary actions due to assumptions of behaviour model synthesis
Runtime behaviour	:	The actual behaviour of a multi-agent system after implementation
Scenario	:	Representation of partial behaviours of a multi-agent system
System-level verification	:	Verification by analysing all system agents' behaviours simultaneously
Traffic Rec.	:	A sensor that collects traffic data
Weather Rec.	:	A sensor that collects weather data

Chapter 1

Introduction

With the increasing size of the problems encountered in industrial applications, demands for Multi-Agent Systems (MASs) have been increased. This enables the engineers to benefit from a set of computing resource components instead of a single one. In multi-agent systems, several agents communicate to each other to perform the desired system actions and behaviours. In agent-based software systems, agents have individual control, know their conditions and can deal with themselves and other agents [1].

Today, MASs have become very popular in industrial applications. Existing faults in the design of industrial MASs can result in costly failures. Detecting and removing these faults in early design stages of MASs is more effective than detecting them during and after implementation stages. In order to reduce the design costs, verification of multi-agent systems is performed. In the verification procedure, MAS behaviours are verified against the unwanted runtime behaviours. Therefore, in this thesis, the main focus is developing a comprehensive method for verification of multi-agent systems.

The rest of this chapter is organized as follows: In Section 1.1, the problem definition and motivations of doing this research are presented. In Section 1.2, the research objectives of this thesis are given, which is followed by research methodology discussed in Section 1.3. Contributions of this work are clarified in the Section 1.4. Finally, in Section 1.5, the organization of this thesis is explained.

1.1 Motivation

Due to the popularity of Multi-Agent Systems (MASs), designing MASs has become a main task of software engineering. Since in most applications the reliability of MASs is a major concern, verification of MAS behaviours is a critical step of design in order to detect unexpected failures. Most of the existing system verification methodologies are defined using the widely accepted Unified Modeling Language (UML) [2]. Considering that UML is proposed for object-oriented software system design, fundamental differences between objects and agents make UML not sufficient for designing the interactions between agents in MAS design. Therefore, this shortcoming should be addressed in MAS verification tools.

Existing unexpected system behaviours in MAS make the system unreliable and unpredictable. To verify MASs against these unexpected behaviours, several techniques have been developed [3–5]. In [5], a component-level approach is proposed to verify the behaviour of MASs by creating state machines for each agent (component) of MAS. In this approach, there are some assumptions for behaviour model synthesis that results in time consuming unnecessary actions. Therefore, this approach should be enhanced to address this issue.

On the other hand, the existing component-level verification techniques analyse the behaviours of each agent individually but cannot consider the behaviour of the whole system at the same time. Even though the unexpected behaviours of individual agents can be caught by component-level techniques, lack of analysis of system-level behaviours may result in neglecting a portion of unexpected behaviours that occur due to agent interactions. Therefore, developing a system-level approach for MAS verification is necessary to provide a thorough behaviour analysis.

1.2 Research Objectives

The main focus of this research is providing a comprehensive method for verification of multi-agent systems. The milestones of this work are as follows:

- Designing Multi-Agent Systems (MASs) using the extended Agent UML [6] in order to represent the interactions among system agents properly.
- Proposing a set of conversion rules to convert AUML sequence diagrams to UML sequence diagrams to be used for verification.
- Verifying MASs by proposing a component-level approach to analyse system requirements and catch unexpected system behaviours.
- Preventing unnecessary actions in component-level verification, which is called over-generalization, to save time and memory.
- Providing a comprehensive behaviour model using labeled transition systems.
- Developing a system-level approach for MAS verification using behaviour model synthesis.
- Detecting unexpected behaviour in early stages of MAS design at system level.

1.3 Methodology

One of the common problems in designing software systems is defining software requirements and characteristics in order to reach software systems' goals. Several factors such as stakeholders and size of the systems can have significant effects on this problem. In order to have a complete software system design, scenarios are used to clearly describe requirements of the systems. Therefore, the behaviour of the software systems can be described using several scenarios. Scenarios are widely used for large-scale software systems with several components such as Multi-Agent Systems (MASs).

Scenarios are usually produced by UML. UML is designed for object-oriented software engineering. Considering that there are not many common features between objects and agents, UML cannot handle the interactions among the agents. In this thesis, in order to overcome this problem, AUML formalism [6], i.e. an agent-based extension of widely accepted UML, is employed to define the agent interactions in MAS. However, most of the existing verification techniques, e.g. [3, 5], are developed based on UML. In this thesis, once the scenarios are produced using AUML formalism, a set of conversion rules are proposed to convert AUML sequence diagrams to UML sequence diagrams in order to prepare scenarios for MAS verification procedure.

A component-level verification method is proposed in [5] to verify multi-agent systems. The behaviours of the system are analysed by verifying individual agent behaviours. This task is performed by synthesising state machines from scenarios. In this method, state machines are produced for each agent of different scenarios. Then, identical states of different state machines are identified and merged in order to detect the unwanted runtime behaviours for each agent of the MAS. One of the shortcomings of this method is overgenralization that happens due to the assumptions in behaviour model synthesis. Considering that not all the identical states are causes of unwanted behaviours, merging all the identical states of state machines leads to overgeneralization. Therefore, in order to deal with this problem, a set of criteria is presented in this thesis to determine which identical states result in unwanted behaviours and should be merged.

On the other hand, the component-level verification methods can only model the behaviours of each agent at a time. Therefore, the behaviours of the whole system that are defined by interactions between agents may be neglected. In this thesis, a system-level MAS verification algorithm is proposed to model the whole system simultaneously and catch all the unwanted system behaviours. Labled Transition Systems (LTSs) are utilized instead of state machines to produce a system-level behaviour model. Finally, the resultant LTSs are verified to catch unexpected behaviours of the MAS. All the proposed methods of this thesis are validated by presenting a case study of a real-time fleet management system. This joint case study project has been negotiated with the Encom Wireless [7] and City of Calgary [8] to design, implement and evaluate a multi-agent simulation system for Commercial Vehicle Enforcement. In this thesis, a small scale prototype of this project is focused as a proof of concept which will be explained in Section 4.3.

1.4 Contributions

The main focus of this research is creating a comprehensive framework for verification of multi-agent systems using behaviour model synthesis. A brief summary of this thesis is shown in Figure 1.1. As shown in this figure, the specific area of this thesis are marked by red rectangles. Several tools such as Message Sequence Charts (MSC) and Sequence Diagrams (SD) exist to represent scenario specifications. Also, Component-level Emergent Behaviour Detection (CEBD), System-level Emergent Behaviour Detection (SEBD) and Model-based Detection and Testing of Multi-agent systems (MDTM) are the possible outputs of behaviour modeling based on the requests.

In this thesis, the Sequence Diagrams (SDs) are considered as the inputs to behaviour modeling because of their abilities to visualize system specifications in a time order. The component-level and system-level approaches are performed for analysing MAS requirements. The component-level approach is proposed to analyse the behaviours of the system while preventing overgeneralization. Furthermore, in order to have a comprehen-

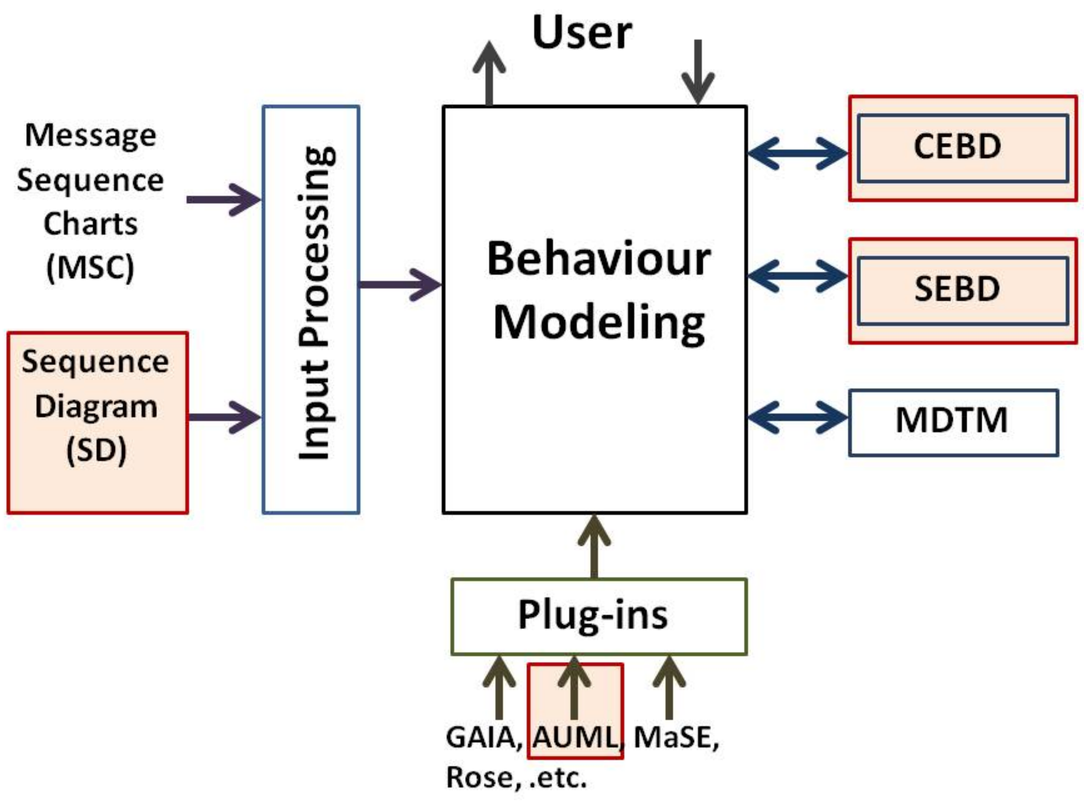

Figure 1.1: Demonstration of thesis outline

sive method for analysing system behaviours, a system-level MAS verification algorithm is proposed. Therefore, the unexpected behaviours that are caught by Component-level Emergent Behaviour Detection (CEBD) and System-level Emergent Behaviour Detection (SEBD) are considered as outputs of this work.

The main contributions of this thesis are listed as follows:

- Designing Multi-Agent Systems (MASs) using AUML formalism.

- Proposal of a set of conversion rules to convert AUML notations to UML sequence diagrams to be used in verification procedure.

- Developing a component-level MAS verification approach that prevents overgeneralization.
- Applying labeled transition systems for synthesizing a comprehensive behaviour model.
- Proposal of a system-level algorithm for verification of MASs.
- Validating the proposed methods by presenting a case study of real-time fleet management system.

Publications Related to This Thesis:

- **S. Mireslami**, M. Moshirpour, B. H. Far, "Detecting Emergent Behavior in Distributed Systems Caused by Overgeneralization", In Proceedings of the International Conference on Software Engineering and Knowledge Engineering (SEKE 2012), San Francisco Bay, USA, July 1-3, 2012, pp 70-73.
- **S. Mireslami**, B. H. Far, "Automated Verification of AUML Based Multi-Agent System Design", In Proceedings of the IEEE Canadian Conference on Electrical and Computer Engineering (CCECE 2013), Regina, Saskatchewan, Canada, May 5-8, 2013, pp 1-4.
- **S. Mireslami**, B. H. Far, "A System-Level Approach for Model-Based Verification of Distributed Software Systems", In press to be published in Proceedings of IEEE International Conference on Systems, Man, and Cybernetics (SMC 2013), Manchester, UK, October 13-16, 2013, 6 Pages.

Other Publication:

- M. Moshirpour, **S. Mireslami**, R. Alhajj, B. H. Far, "Automated Ontology Construction from Scenario Based Software Requirements Using Clustering Techniques", In Proceedings of the IEEE International Conference on Information Reuse and Integration (IRI 2012), Las Vegas, USA, August 2012, pp 541-547.

1.5 Thesis Outline

In this thesis, several methods are proposed for synthesis of behaviour model for multi-agent systems in order to catch unexpected system behaviours. In Chapter 2, the background information for Agent UML is presented. The related works of Multi-Agent Systems (MAS) verification are discussed in Chapter 3. In Chapter 4, to design the interactions among agents, AUML formalism is employed and a set of conversion rules is proposed to produce UML sequence diagrams from AUML notations. Chapter 5 presents a proposed component-level technique for MAS verification by synthesis of behaviour models from scenarios in order to detect emergent behaviours while preventing overgeneralization. In Chapter 6, to obtain a system-level behaviour model for system analysis, an approach using labeled transition systems is proposed. Finally, conclusions and future work are presented in Chapter 7.

Chapter 2

Background: Agent UML

2.1 Introduction

In this chapter, the preliminaries and background information related to the design of Multi-Agent Systems (MAS) are provided. In addition, the background for scenario-based specifications, distributed systems, multi-agent systems, Unified Modeling Language (UML) and Agent Unified Modeling Language (AUML) are presented.

The rest of this chapter is organized as follows: In Section 2.2, scenario-based specifications are introduced along with sequence charts. Distributed systems are introduced in Section 2.3. The background for multi-agent systems is described in Section 2.4. Furthermore, this chapter provides a background for unified modeing language and agent unified modeling language in Sections 2.5 and 2.6, respectively, for designing object-oriented and agent-based designs. Finally, the chapter is summarized in Section 2.7.

2.2 Scenario-Based Specifications

Using scenario-based specifications is a popular approach to represent software system requirements. In order to show the objectives of a system, interactions among the users (e.g. programmers, designers, and engineers) are defined by scenarios. Scenarios give software designers details of interactions among the components to enable a comprehensive system implementation [9].

Scenarios are usually represented by an array of messages that show the behaviours of the system. The sequence charts that are defined in the Unified Modeling Language (UML) are known and approved representations to demonstrate scenarios [10]. Message

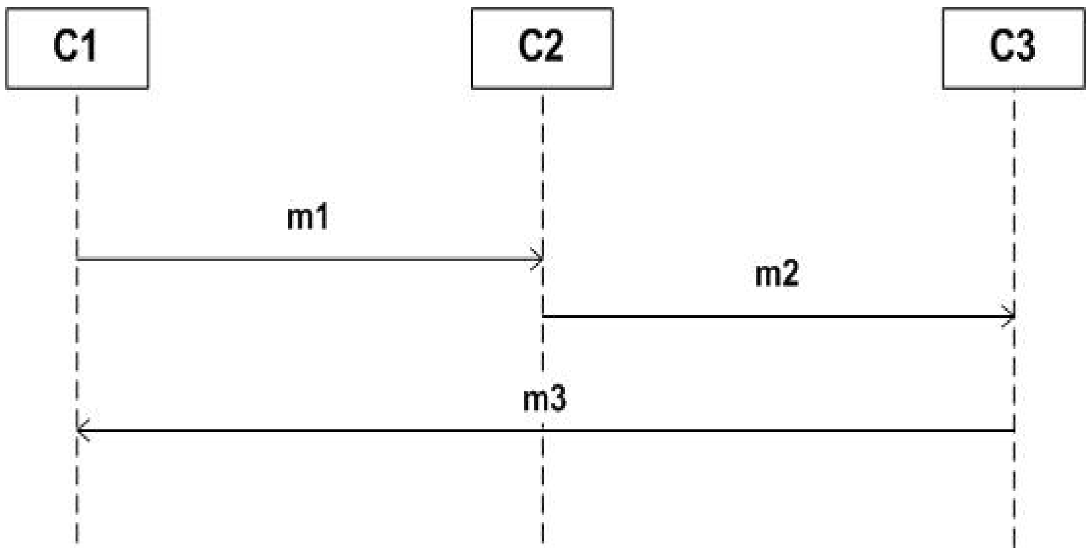

Figure 2.1: An example sequence chart

Sequence Charts (MSCs) and Sequence Diagrams (SDs) are popular types of sequence charts which are commonly utilized for representing the software systems [11]. International Telecommunication Union (ITU) has already standardized MSCs [12] while SDs are standardized by Object Management Group (OMG) [2].

Each scenario includes several system components that are major processes of the software system. In sequence charts, these system components are shown by rectangles with vertical lines. The interactions among the system components are represented by passing messages between them. In sequence charts, arrows are used to indicate messages. The messages are organized in a top down order meaning that a message that is represented at the top of the sequence chart is sent before all the messages that come after [10].

In Figure 2.1, a simplified example of sequence chart is shown. As shown in this figure, the system represented in this example includes three components that are shown by C_1, C_2 and C_3. These components interact with each other by passing messages m_1, m_2 and m_3. Since the order of messages is top down, first, message m_1 is sent which is followed by m_2 and m_3. It should be noted that these details are valid for both MSCs and SDs.

2.3 Distributed Systems

Distributed systems are composed of interacting components that are parts of a single network. For a software system to attain its objectives, these components communicate with each other. Designing distributed systems is a complicated task [5], since the system components take actions independently while they may be affected by the other components' actions.

As in most applications there is no central controller in distributed systems, components should have the ability to coordinate their activities. However, system components of a distributed system execute in concurrent processes. Therefore, component independency and concurrency are the characteristics of these systems [5, 13]. Operating systems and ATM are two examples of distributed systems with centralized controllers and Peer-to-Peer (P2P) systems are examples of distributed systems without centralized controller [13].

2.4 Multi-Agent Systems

Multi-Agent Systems (MAS) have become very popular today. Industrial demands for agent-based software engineering have significantly increased in the past years. This demand increase is due to the increasing trend of the complexity of the problems encountered in the industry [14].

MASs include several interacting agents in a network. Agents are autonomous and interactive software entities that act both alone and with other agents. In addition, these agents know their conditions and the intended effects of their actions, hence, take responsibility for their needs [1].

Considering that agents are autonomous, MASs are recognized as self-organized systems that can handle complex problems, i.e. problems that cannot be addressed using an individual agent. Therefore, MASs are employed for designing large-scale distributed software systems. Typical applications of MASs are in design and modeling of social systems and online trading [6].

In order to enable the development of MASs, Agent Oriented Software Engineering (AOSE) techniques are developed. AOSE techniques are new methods for representing, analysing and designing a software system. Multi-agent Software Engineering (MaSE) [1] and Agent Unified Modeling Language (AUML) [15] are two well-known methodologies among others used for designing multi-agent systems that will be described in the following sections.

2.5 Unified Modeling Language

In order to represent and model software systems in a standard way, Unified Modeling Language (UML) has been defined in 1990s [16]. Designing large-scale software systems is a challenging task. Since all the details of a software system should be considered in the design stage in order to meet the requirements, it is very important to perform a detailed system modeling. Modeling a software system gives a manageable abstraction of the actual system [17].

UML is a language for modeling software systems which is defined to provide a comprehensive system model. Major parts of UML are sets of semantic notations that make UML an effective language for modeling object-oriented software systems. Considering that cooperation of component modeling, data modeling and object modeling creates UML, it can be utilized for system development and implementation [16]. International organization for standardization has accepted UML as a standard language for modeling industrial software systems [17].

UML includes several types of diagrams such as Use Case, Activity, Class, Object, Sequence, Communication, Timing, Interaction Overview, Composite Structure, Component, Package, State Machine and Deployment diagrams [16]. Deciding which diagrams to use is the task of software developers based on the type of system applications. In order to find the most appropriate UML diagrams to model a software system, these diagrams are classified into five categories: logical view, process view, development view, physical view and use case view [16]. These categories are described briefly in the following paragraph.

The diagrams in the logical view category are used to visualize the software systems to help achieve a complete development. Sequence diagrams are examples of logical view category. The diagrams that are in the process view category are used to visualize the processes happening within the system. Activity diagrams are examples of process view category. In order to show communication between system components, the diagrams in the development view category are utilized. Component and package diagrams are subsets of this category. To design software system, the diagrams in the physical view category are used. These diagrams connect the system abstraction to the deployed system. This set contains deployment diagrams. Use case view category helps the development process represent the system behaviour and actions. This category includes overview and use case diagrams. In the following subsections, the most popular UML diagrams are described briefly.

2.5.1 Sequence Diagrams

Interaction diagrams are often used to model software systems by representing the interactions among different components of the systems. Since it is very beneficial to represent the order of interactions in a simple way, Sequence Diagrams (SD) are the most popular types of interaction diagrams. The interactions among different components of a software system can be expressed using sequence diagrams [17].

Sequence diagrams represent the interactions among objects in the order of time. Therefore, in these diagrams, only the time when an interaction (i.e. event) starts is represented and the duration of interactions and events is not considered. Sequence diagrams also include all the system components also known as system participants.

In sequence diagrams, system components are represented using rectangles with corresponding lifelines. Lifelines are shown using vertical lines. In these diagrams, the time increases in a top-down order. Events in the sequence diagrams are considered as the points when an interaction may occur. The interactions among system components are represented by messages and are shown using horizontal arrows. The direction of a message is from the component that wants to send that message to the component that will receive it. Furthermore, action bars are shown by vertical rectangles in sequence diagrams. These bars are used to represent the period of time when a component is active, i.e. sends or receives messages [16].

In Figure 2.2, an example sequence diagram is shown. In this diagram, there exist four system components which are named ”User”, ”Server”, ”Selector” and ”Interface”. These system components interact with each other by passing messages such as ”Send Location of User”.

In this thesis, sequence diagrams are used to represent system specifications. Considering that in this research a visualized software system model is needed to show different components of the system and interactions among them, sequence diagrams are very beneficial. In addition, sequence diagrams provide a time order of the events in a software system and enable the system designers to perform timing analysis for the system.

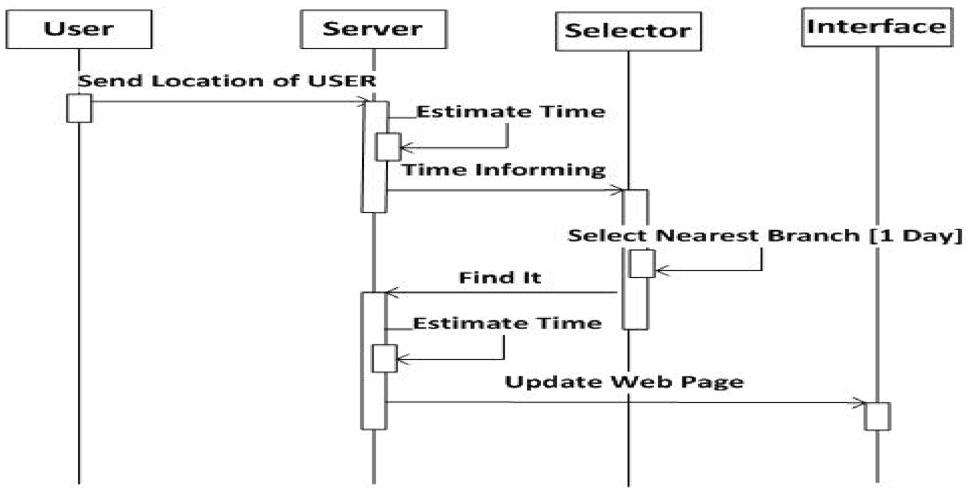

Figure 2.2: An example sequence diagram

2.5.2 Communication Diagrams

Communication diagram expresses the communications among system components. It indicates which components should be connected to enable the necessary interactions in the system. Since sequence diagrams and communication diagrams are similar, choosing the appropriate diagram is usually based on the required information and is the decision of system designer. In the applications where the order of passing messages should be considered, sequence diagrams are utilized. On the other hand, the communication diagrams can be used if it is desired to to represent the connections among different parts of the system [16].

Communication diagrams include all the system components, communication messages and communication links. System components are shown by rectangles that contain the name of the components and classes. In communication diagrams, arrows are used to represent communication messages. The direction of messages is from sender to the receiver. To connect two components, a communication link is used which is shown using a single line [16].

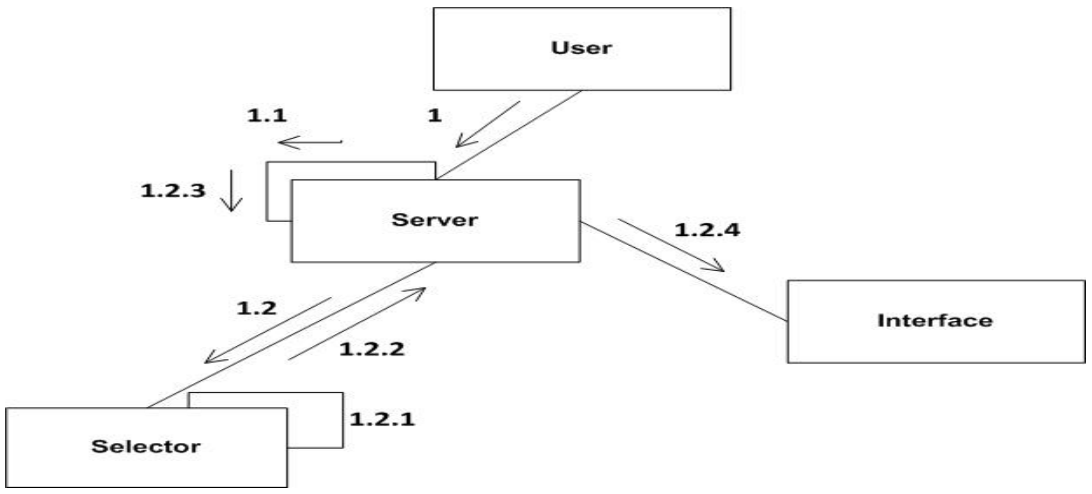

Figure 2.3: An example communication diagram

Since components can be placed anywhere in the communication diagrams, these diagrams are usually known as free-form diagrams [17]. Therefore, in order to show the order of messages, a label is assigned to each single message. The procedure of labeling messages starts by assigning number 1 to the first message and continues incrementally until all messages are labeled. A message is considered as a nested message when it can be only activated by another message. In fact, activation of a nested message depends on occurrence of another message. The label of a nested message includes both the number of the activating message and the number of the nested message which are separated by a dot. Foe example, if nested message with number 5 is activated by a message with number 1, its label will be ”1.5”.

Figure 2.3 represents an example of communication diagram. This diagram includes four system components that are ”User”, ”Server”, ”Selector” and ”Interface”. The interactions among these participants are shown using the links (black lines). In addition, the order of messages are represented using their numbers while they are shown using arrows. As an example, the message labled as ”1.2” that corresponds to message ”Time informing” in sequence diagram of Figure 2.2 is a nested message that is activated by the message with label ”1”.

2.5.3 Timing Diagrams

Timing diagrams are utilized when time representation is needed for modeling software systems. These diagrams are considered as appropriate UML diagrams, when it is desired to put time restriction for system modeling. For example, a timing diagram is used when a designer aims to indicate that a message is required to take place in a certain period of time. Timing diagrams are employed to represent the behaviours of objects in a requested time interval. This time information cannot be provided using the other UML diagrams. All the information about the duration of events and their send and receive times is provided using timing diagrams [16].

A timing diagram includes all system components, states, times and messages. In the left side of the timing diagram, the names of the components appear vertically. There are several states in timing diagrams where system components take part. The related states corresponding to each component are placed close to it. The time order is represented in an increasing order from left to the right of a timing diagram. In different time intervals, system components can be in different states. In addition, in timing diagrams, messages are used to represent when a system component changes its state. These messages are shown by arrows that connect the current state of each component to the next state [17].

In Figure 2.4, an example of a timing diagram is represented. As it can be seen, this timing diagram includes two system components. There exist four states for the component ”Participant 1” and three states for the component ”Participant 2”. In this diagram, two messages are used to show the interactions among the system components. For example, ”Message 1” connects the State 2 of ”Participant 1” to the State 2 of ”Participant 2”. This message is sent by ”Participant 1” at time ”1t” and received at time ”2t” by participant 2.

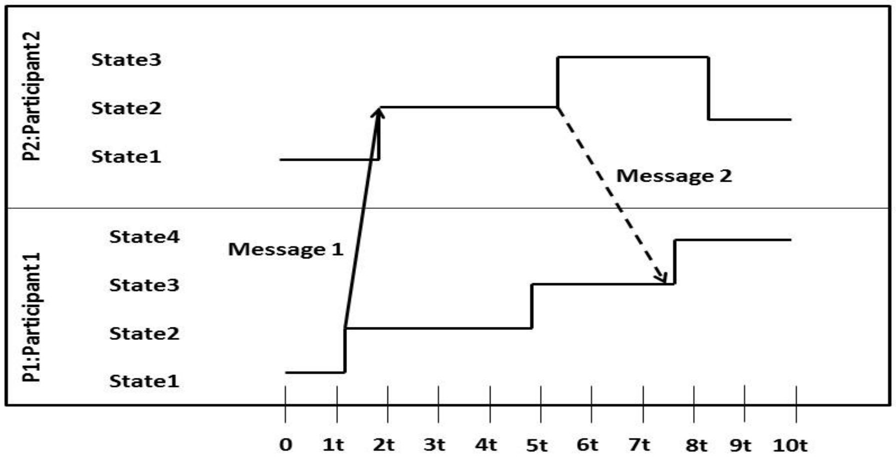

Figure 2.4: An example timing diagram

2.5.4 Interaction Overview Diagrams

In order to implement a software system, several interactions need to happen. To have a high-level view of these interactions for modeling a system, interaction overview diagrams are utilized. Interaction overview diagrams may include all the diagrams that are explained in previous sections such as sequence diagrams and timing diagrams. Based on the required information for modeling a software system, different UML diagrams are combined and represented in a single interaction overview diagram [17].

In interaction overview diagrams, the names of system components appear in the lifeline section. The interaction overview diagram starts with an initial node shown using a filled circle and ends with a final node which is shown by a circle with a filled inner circle [16].

An example interaction overview diagram is shown in Figure 2.5. This diagram includes two other UML diagrams: a sequence diagram and a communication diagram. There exist four system components in this diagram which are named ”User”, ”Server”,

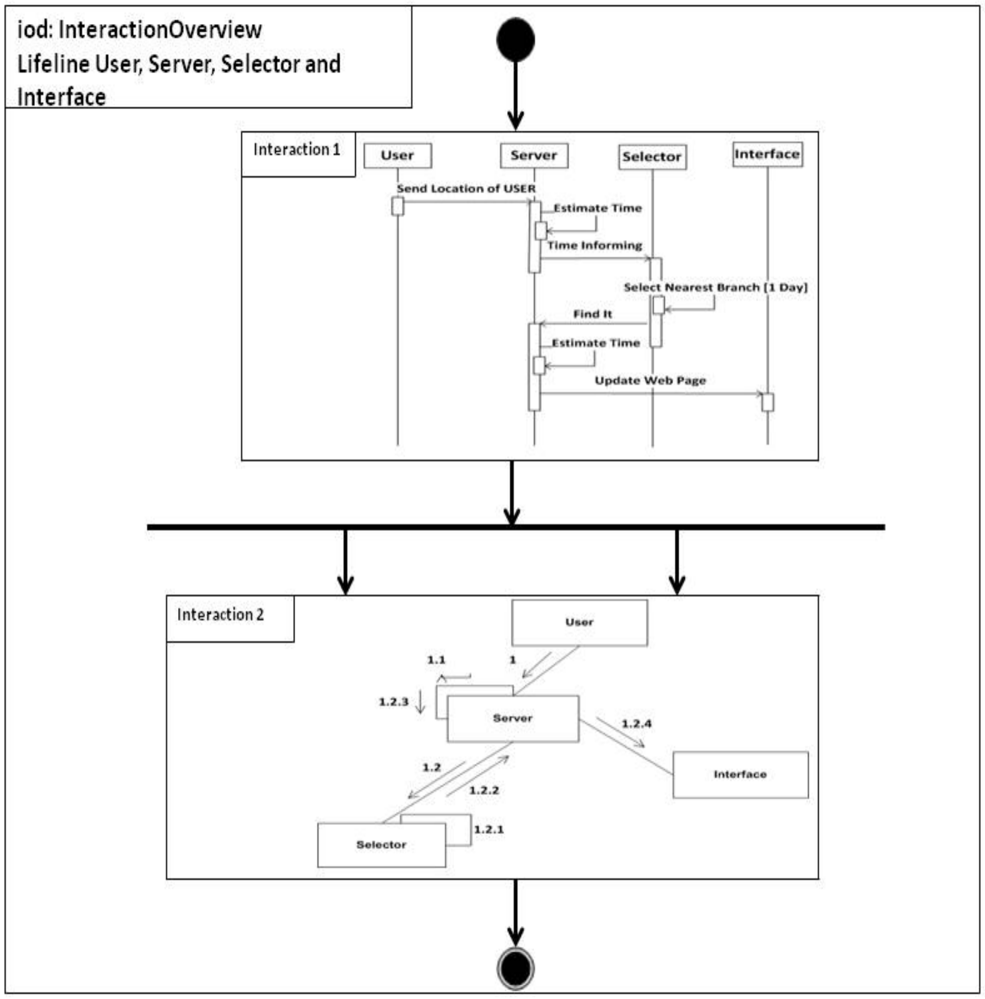

Figure 2.5: An example interaction overview diagram

"Selector" and "Interface". These components are expressed in the lifeline located at the top left hand side of the diagram.

2.5.5 State Machine Diagrams

In order to model the software system behaviour, state machine diagrams are employed. State machines represent the behaviour of system by focusing on the states of the software system. System behaviour is analysed by considering changes in the states [16].

States and transitions are used to represent state machine diagrams. States are shown by circles. Arrows are used to express transitions that show changes in the states of the system. When the transition enters any state, it becomes active and once the transition goes out, the state becomes inactive. The changes in the system states is resulted from system events. Events are shown as labels on top of the transition arrows. Initial states of the state machine diagrams are shown using two concentric circles and the final states are shown using two concentric circles with a filled outer circle [16].

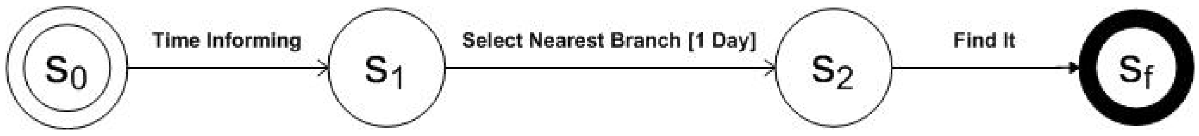

Figure 2.6: An example state machine diagram

An example state machine diagram, shown in Figure 2.6, is used to model the behaviour of a single system component. This diagram includes three events that are "Time Informing", "Select Nearest Branch [1 Day]" and "Find It". To show how component's states change, transitions are used. These transitions are labeled by the name of events that activate them.

2.6 Agent Unified Modeling Language (AUML)

Industrial demands for Agent-Oriented Software Engineering (AOSE) have significantly increased in the past decades. Since life-cycle of software design can be completely done using agent-based techniques, AOSE is becoming more accepted for software design in industrial applications. Using agent-based software development formalism results in saving costs including time and money required for a software project [15]. Agent-based methodologies can address the industrial problems that cannot be handled using only a single agent.

In order to reduce the risk of developing an agent-based system, extension of existing tools is required. Object-oriented software systems are generally similar to agent-based systems. Therefore, providing an extension of the trusted methods that are used to design object-oriented software systems can help have an appropriate and comprehensive agent-based development formalism. Agent Unified Modeling Language (AUML) is a formalism for agent-based software development that is an extension of the widely accepted UML. AUML handles the interactions among agents by extending UML in order to address the challenges in dealing with agent-based designs [18].

Developing agent-based systems by extending UML is suggested by the Foundation for Intelligent Physical Agents (FIPA) and Object Management Group (OMG). AUML represents several agent-based design approaches such as an Agent Interaction Protocol (AIP). In AIP, interactions among agents are shown with the order of messages. Therefore, AUML can be well understood and utilized for analysing agent-based software systems [6].

Considering that agents are autonomous and interactive, they can act independently or with other agents. While outside control is needed for objects to execute their methods, agents know their conditions and intended effects of their actions, hence take responsibility for their needs. However, lack of common features between objects and agents makes UML insufficient for multi-agent system (MAS) designs. In order to show communication between agents in MASs, interaction protocols are utilized. Therefore, three levels of AUML formalism are performed to express the protocol structures in order to show interactions among agents in multi-agent systems [19]. These levels are described in the following subsections.

2.6.1 AUML: First Level

In the first level of AUML formalism, to show any type of interactions among system agents, agent interaction protocols are used. Templates and packages are employed as parts of the well-understood UML for protocol representation [6].

In UML, packages and components are used to group the behaviour and structure of the object-oriented systems. For implementing a software system, its classes should be combined using system components. Each component is produced to combine several classes in order to achieve certain goals. Then, packages are used to group the system components. In addition, behaviour diagrams can be grouped using packages. For each sequence diagram, packages are defined based on the available data and requirements. However, users sometimes do not need all the packages. Therefore, only some particular packages are used according to the users' requests [19].

Figure 2.7 represents an example of producing packages for a car dealership. The customer may not need to know all the available information at the same time. Therefore, two packages, "Sale" and "Repair", are produced in this example. Based on the required information, both packages or just one of them may be used. For instance, if the customer needs to book an appointment for repairing his/her car, it is only required to use "Repair" package. Note that as it can be seen in this example, system components can be part of several packages at the same time. In this example, component "Customer Service" is

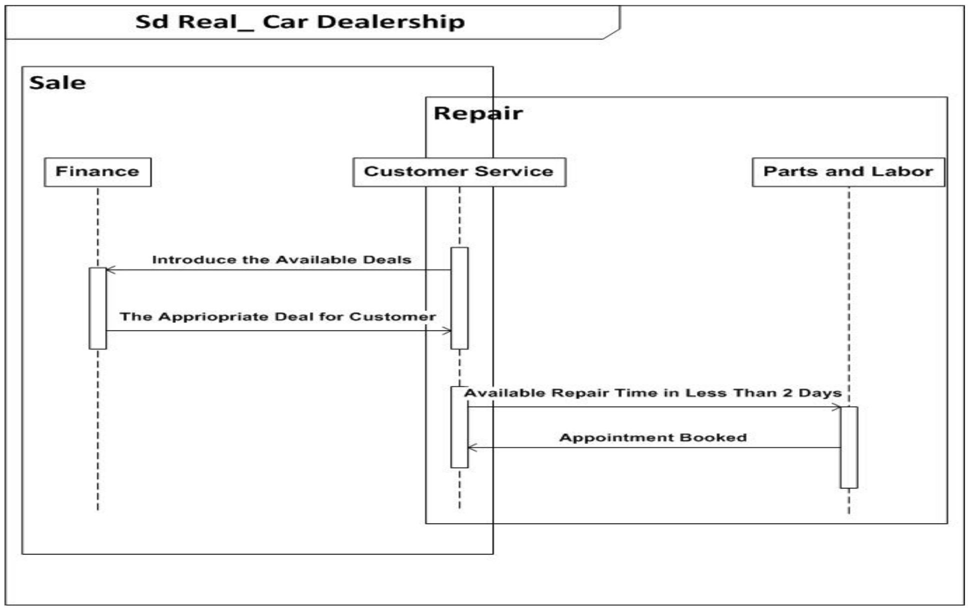
Sd Real_ Car Dealership
Sale
Repair
Finance
Customer Service
Parts and Labor
Introduce the Available Deals
The Appriopriate Deal for Customer
Available Repair Time in Less Than 2 Days
Appointment Booked

Figure 2.7: Producing AUML packages for a car dealership

included in both "Sale" and "Repair" packages.

Packages for AIP are considered as templates in UML. Templates are shown by dotted boxes placed in the upper right portion of the sequence diagrams. A template includes three rows. In the first row, the roles of system components are indicated. System constraints are presented in the second row. Finally, the communications among agents, i.e. messages, are expressed in the third row [15].

In Figure 2.8, a template for the car dealership example presented in Figure 2.7 is shown. The first row of the template includes the system components, i.e. agents. The second row includes the constraint for the car dealership example which is finding a repair appointment in less than two days. The interactions among the system components are shown in the third row.

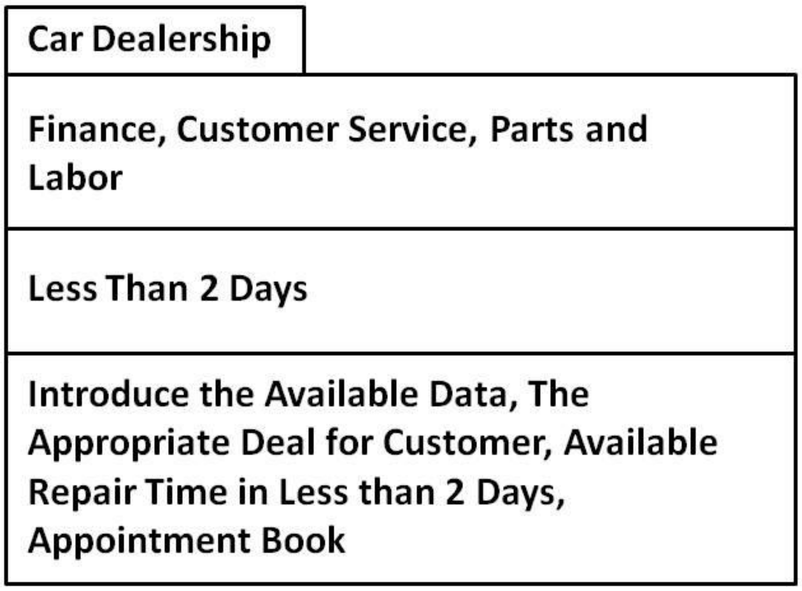
Car Dealership
Finance, Customer Service, Parts and Labor
Less Than 2 Days
Introduce the Available Data, The Appropriate Deal for Customer, Available Repair Time in Less than 2 Days, Appointment Book

Figure 2.8: Template for "Car Dealership" diagram

2.6.2 AUML: Second Level

The second level of AUML formalism represents the interactions among the system components. Similar to UML, in AUML, sequence diagrams are used to show the interactions among the agents. The sequence diagrams focus on the order of message passing. Therefore, in AUML, to show the interactions among agents, some extensions are added in addition to UML. One of these extensions is the definition of "role" that is used to categorize the agents based on their behaviour. Defining roles leads to decrease in the size of interaction diagrams [20].

In AUML sequence diagrams, agents are shown using rectangles. The name and role of each agent are written in the format, agent:name/role:class, in its corresponding rectangle [15]. For example, for the agent "Finance" in Figure 2.7 with the role of "Employee" and a class of "person", the format of the agent rectangle box will be written as Finance/Employee: person.

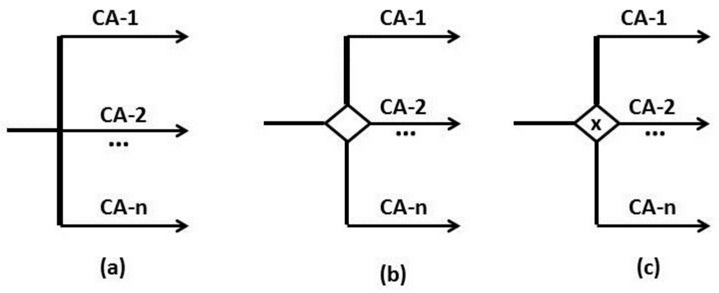

Figure 2.9: AUML Concurrent Interaction Notations

Considering that object-oriented approaches cannot handle the concurrent interactions, in AUML formalism , UML sequence diagrams are extended by adding several notations for representing concurrent agent interactions. These interactions are shown by arrows that are labeled with Communication Act (CA). These notations are shown in Figure 2.9. In Figure 2.9(a), the notations indicate that all the messages are sent concurrently. When a decision needs to be made to determine which messages are allowed to be sent, the notations in Figure 2.9(b) are utilized. Note that using these notations one or more messages can be sent simultaneously. When more than one message are sent simultaneously, the communications among agents are considered to be concurrent [20]. Finally, if only one message is allowed to be sent at a time, the notations in Figure 2.9(c) will be used.

Collaboration diagrams are other types of AUML diagrams. The goal of collaboration diagrams is showing the relations between the agents. Although, sequence diagrams and collaboration diagrams are similar semantically, in collaboration diagrams, agents are not located in a specific way and the order of interactions are numbered top down. The domain experts usually make the decision to use sequence diagrams or collaboration diagrams based on how understandable they are in that specific application. Each agent may have several roles in the collaboration diagram that are represented by dotted arrows [19].

Activity diagrams are used to provide a simplified representation of the behaviour models. In these diagrams, actions along with the events that trigger them are presented. Activity diagrams provide visual representation of software systems in a simple way. Asynchronous and concurrent messages can be expressed by activity diagrams. Activity diagrams in AUML are shown using rectangles that have a pentagon in the upper left hand side. The name of each protocol is written in a pentagon that is prefixed by sd. [20].

To show the behaviour of the software system, statechart is employed. It includes agents' states and messages that show when these states change. The main concentration of a statechart is on the interactions among agents [15].

In the second level of AUML, the interactions between agents are represented. Therefore, based on the type of multi-agent systems and required information, each of the above diagrams can be used.

2.6.3 AUML: Third Level

Interactions within an agent, intra-agent interactions, are represented in the third level of AUML formalism. In order to express intra-agent behaviour, each of the diagrams discussed in the second level, i.e. sequence diagram, collaboration diagram, activity diagram and statechart can be used. The decision on using which diagram is made based on the role of agent, required information and designer's preference.

2.6.4 AUML Notations For Multi-Agent System Design

Considering that Agent UML is an extension of the accepted UML, AUML includes all UML notations. Also, there are several extended notations for AUML in order to deal with agent interactions [20]. These extended notations are described briefly in the following.

Role: The interactions among agents can be represented by defining roles for agents. Defining roles for each agent helps model complex software systems. So, showing interactions among roles can reduce the size of modeling diagrams. To provide a representation of agent roles, activity diagrams are utilized. In activity diagrams, system events are connected to their related roles [21].

Constraint: Constraint shows which conditions should be hold in order to perform appropriate actions. There are two types of constraints in AUML: blocking/nonblocking constraints and timing constraints. These constraints are presented in square brackets on the top of messages. Lifeline avoids to take part in interactions when blocking constraints happen. The blocking constrains are shown by "≪blocking≫". However, nonblocking constraint does not stop any execution. It only avoids sending and receiving message with invalid constraints. Furthermore, to show the delay among two messages, timing constraints are used.

Continuation: Continuation is defined to show an intermediate point in a CombinedFragment or a control flow. CombinedFragment is employed in both UML and AUML diagrams to show possible traces. Based on system specifications, several operations can happen. These operations are shown as InteractionOperators which are presented in the upper left side of a rounded rectangle. There are several types of InteractionOperator for CombinedFragment such as loop, parallel, break, assertion and alternative [21]. Continuation is represented by a rounded rectangle that includes the related CombinedFragement name. CombinedFragment notations can appear before and after the label name. Continuation is indicated as a filled triangle in the CombinedFragment notation [22].

InteractionOccurrence: When an interaction diagram internally calls another interaction diagram, it is called InteractionOccurrence. The interaction diagram continues after the called interactions end. The name of the called interaction diagram should be written in a rectangle. This rectangle includes a pentagon labeled with ”ref” [21].

Gate: To associate the inner messages of an interaction diagram with messages that are part of another interaction diagram, a connection point named gate is used in AUML. Gate can be in sending or receiving points of a message [22].

Action: Action is denoted in order to show how an agent should interact with another agent. All agent activities for sending and receiving messages are defined by action. Message sender knows the content of a message while the agent that receives the message does not have any idea about message's content. Action is represented by a rounded rectangle which is linked to the message that activates it [22].

2.7 Summary

In this chapter, the preliminaries for the design of multi-agent systems are presented. Scenario-based specifications, distributed systems and multi-agent systems are introduced to provide a basis for the contributions of this thesis. The unified modeling language is described as a well-known language for object-oriented software design. Finally, an extension of UML, AUML, for enabling a comprehensive agent-based design is described in detail along with the extended notations specific to agent-based designs.

Chapter 3

Related Works

3.1 Introduction

In this chapter, the related works of Multi-Agent Systems (MAS) verification are discussed. Considering that employing several agents simultaneously can provide a more effective software system, industrial demands of MASs are growing over the years. With these increasing demands, verifying and monitoring the behaviour of the agent-based software systems with low cost has become a major step of the design. Bringing MASs to the main stream of commercial software development can be performed by verifying the system behaviour against the unwanted runtime behaviours, i.e. emergent behaviours [23–25]. Detecting and removing these behaviours in design stages help reduce the deployment costs significantly [26]. Verification of MASs can be performed using both component-level [25] and system-level approaches [24, 27].

There exist model-based techniques for software verification using UML [5]. However, these are not adequate for agent-based software system verification. In order to handle this problem, several approaches are presented for designing MASs. The modeling elements of these methods are partially different from the UML constructs to handle interactions among agents [4, 28–31]. Verifying MASs can be performed using UML, AUML, Multi-agent Software Engineering (MaSE), Message Sequence Graph (MSG), high-level Message Sequence Charts (hMSC) and non-local branching choice. Each of these methodologies will be described in this chapter.

The rest of this chapter is organized as follows: In Section 3.2, the model checking approach for MAS verification is described. MAS verification using MaSE methodology is presented in Section 3.3. In Section 3.4, the verification of MASs using several design methodologies is discussed. Finally, in Section 3.5, the chapter is summarized.

3.2 Verification of Multi-Agent Systems Using Model Checking Approach

Model checking provides a comprehensive system model for checking multi-agent system specifications. There exist two types of model checking approaches for verification of multi-agent systems that are described in the following paragraphs.

The first type of model checking, which is known as MAS model checking, focuses on the agent characteristics in MASs. Therefore, in order to deal with MAS, several agent-related logics are added. The Beliefs, Desires and Intentions (BDI) model of agents is considered in this type of model checking [32]. In this type, logic languages are used for MAS verification. In fact, BDI cannot be defined for MAS verification without using logic languages. The focus of model checking methodologies is representing the software system model rather than their implementations. A set of states are used to represent agent conditions in the MAS model checking. Agent actions are defined in order to represent how the state of each agent changes to obtain its goals. In [32], a technique is proposed to formulate the objectives of software systems.

MASs are very large-scale software systems with large number of agents and interactions. In these systems, agents can be added any time in the middle of the design stage. This can result in a rapid increase in the number states that each agent can have. Since all the states of all agents should be considered during verification, verification of large-scale MASs using the model checking approaches is a complicated task [33, 34].

In [32], in order to use model checking for MASs, first, system requirements are defined using logic languages. Then, several rules are assumed to form the system specifications. Finally, the model checking approach checks if the system specifications are valid. Although this approach can deal with large-scale software systems, it cannot support all the agent characteristics. This is because by increasing the number of agents, the number of states significantly grows that may lead to state space explosion. Therefore, some characteristics of the agents should be ignored to manage the MAS verification. In addition, model checking techniques can be used for verifying system implementation as well as the system design [35].

The second type of model checking approaches is designed to check if software systems meet their requirements and detect the possible problems such as deadlocks. Detecting deadlocks is necessary since they do not allow software systems to be implemented completely and correctly. In addition, this model checking approach is one of the popular automated methodologies for verifying multi-agent software systems [36]. Model checking can be used to model the behaviour of software systems and interactions among agents as well as verifying MASs to detect the possible problems. Model checking follows the accepted structures of software system designs and use logic languages and finite state machines [37].

MAS verification using model checking ensures that the required properties of software systems are achieved. Designers give software specifications as the expected behaviour of the system and compare the implemented system executions against them to verify system performance. Since the second type of model checking approaches analyses system behaviour and detects the system faults, they are considered as effective verification methodologies. Although the model checking approaches are known as powerful verification methods, considering all system states to show the behaviour of system agents makes them time consuming [38, 39]. The MAS verification technique proposed in this thesis, is categorized as the second type of model checking approaches which is used to detect the exact causes of system faults in the design stage to reduce deployment costs.

3.3 Verification Using Multi-agent Software Engineering Methodology

Multi-agent Software Engineering (MaSE) methodology is one of the widely accepted methods for representing agent-based system architectures and the interactions among agents. MaSE methodology is generated upon several UML diagrams. Since objects cannot handle interactions among agents, MaSE considers agents as objects that use communications, i.e. messages, for interacting with the other agents [4].

MaSE can be used to design and analyse agent-based software systems. Designing agent-based systems using MaSE is performed in four levels. First level is creating agent classes. In the second level, communications among system agents are constructed. Collecting agent classes and designing the system are performed in the third and fourth levels, respectively. MaSE has a comprehensive system analysis that includes three internal steps. In the first step of this analysis, the main goals of the system are determined. In the second step, the use cases of the system are employed. Finally, in the third step, the system goals are improved and refined [40].

Agents are the most important constructs of MAS. Therefore, in order to represent the system goals and agents' classes, specific roles are defined for agents. Agents in MASs are linked to their roles using MaSE task diagrams. Then, for each agent role, tasks are defined. These tasks, which are shown by task diagrams, provide the ability to complete and achieve the related goals of agents. Since MaSE is developed based on UML, UML state machine diagrams are utilized for representing MaSE task diagrams. In addition, the control flow diagram is used to show the goals and achievements [4].

Sequence diagrams are well-known interaction diagrams that can represent the interactions among the agents in MASs. Therefore, most of the existing MAS verification techniques use sequence diagrams [4, 5, 24]. In sequence diagrams, to show the communications among system agents, roles and agents are expressed by lifelines. Considering that MaSE does not include sequence diagrams, in order to have a comprehensive MAS analysis, sequence diagrams can be produced according to the MaSE design information. Agent classes and role diagrams available in design and analysis phases of MaSE provide these required information by representing agent communications or the interactions among roles of agents. Therefore, a verification technique that uses MaSE methodology for analysing MASs in order to catch unwanted runtime behaviours is proposed in [4].

In [4], the agent sequence diagrams are produced by converting role class diagrams of MaSE to agent sequence diagrams. By grouping the related roles of each agent and considering the messages between each pair of agents, the agent sequence diagrams are produced. In these agent sequence diagrams, lifelines represent agents and the messages between them are recognized as new agent communications. In order to catch software system goals using MaSE, corresponding roles of agent classes are used. The activities and communications among roles are considered as MaSE tasks and are represented using state machine diagrams [41]. The behaviours of agents can also be modeled using UML state machine diagrams. These behaviours are shown by indicating the interactions of each agent with the agent that activates it. Therefore, in order to analyse the system behaviours, the task diagrams of MaSE are converted to UML state machine diagrams in [41]. State machine diagrams are produced using UML notations instead of the conditions in MaSE task diagrams.

In [4], several resources, i.e. agent roles, are dedicated to the agents in MaSE task diagrams. In this method, specific resources should be allocated to each agent. Although agents can request for several resources, they can only hold one resource at a time. If all the requested resources are provided for an agent, agent task can be accomplished. Therefore, an agent is called to be a deadlock in MAS if at least one of its requested resources is already held by another task diagram run which means that agent's tasks can not be performed.

3.4 Verification of Multi-Agent Systems Using Message Sequence Charts, Message Sequence Graphs and high-level Message Sequence Charts

In order to describe and analyse software system behaviours and requirements, several methodologies are proposed. Scenario-based specifications are the systematic approaches for representing MAS requirements. The interactions among system agents can be defined by several scenarios in the scenario-based specifications. These scenarios represent the behaviours of software systems. Message Sequence Charts (MSC) are categorized as a type of the scenario-based specifications that can be used to represent software system behaviours. Each MSC is used to describe one scenario, i.e. partial behaviour of software system. In [5, 27, 42], MSCs are utilized to perform MAS verification.

In spite of MSCs, Message Sequence Graphs (MSG) and high-level Message Sequence Charts (hMSC) focus on representing several scenarios simultaneously [30]. MSGs are considered as system models that include several different analyses. MSGs are utilized as graphical representations for showing how different message sequence charts are connected. MSGs and hMSCs are usually considered to have similar capabilities and shortcomings and most researchers consider them equal [43]. In message sequence graphs, several MSCs are connected to each other in order to provide an analysis of a multi-agent system. These graphs include several nodes and arcs. Each node represents a single MSC and the arcs represent the interactions among them. High-level message sequence charts are similar to MSGs but are often used to handle relatively large analyses. In addition, hMSCs contain nodes that can represent both MSCs or MSGs.

In [30, 44, 45], two different synchronous and asynchronous models are used to perform verification for MAS with multiple scenarios represented by a MSG. In this MSG, several message sequence charts are combined using several actions such as concatenation. The techniques proposed in [30, 44] determine if the considered MAS is weak or safe realizable. This is performed by understanding the conditions of software system based on its implementations. If the MAS implementation is deadlock free, it is considered as a safe realizable system. Otherwise, it is known to be weak realizable. Since in one MSG, several MSCs are combined, designing asynchronous MASs using MSGs is a difficult task [30]. In addition, this method is not an automated technique and its precision of MAS verification is not clearly known [28].

In [46, 47], the problems that may happen as a result of the differences between MSC specifications and system implementations are presented. For instance, one of these problems occurs when the system agents do not interact with the same speed. Therefore, the expected communications cannot be occurred at the expected time. Another problem happen when different approaches are used for implementing MSC specifications. This problem is called ”non-local branching choice” and may cause deadlocks in MAS behaviour. In [46], it is proposed to address these problems by analysing the messages that are passed between system components, i.e. agents. This algorithm is implemented in a tool, called MESA, for checking non-local branching choice and analysing MSC specifications [46, 47]. However, there is no implementation for model checker of MSC specifications using MESA provided by the authors.

In [48, 49], an automated algorithm for producing UML statecharts from system scenarios is proposed. Going from scenarios to statecharts is a challenging procedure [48]. One of these challenges happens when several independent scenarios are connected to each other. Another challenge is the existence of same states in different scenarios. Finally, since this procedure is usually performed by the designer, it is not very precise.

In order to address these problems, some semantic information are added to the domain theory in [49]. In this work, an automated algorithm is proposed to obtain statecharts from scenarios that are usually defined by sequence diagrams. By going from scenario to statecharts, verifying agent-based systems can be performed completely.

Statecharts enable the designers to define agent behaviours [49]. The algorithm first converts sequence diagrams into statecharts by labeling each sequence diagram with state vectors. Object constraint language [50] is used to represent the conditions before and after each state. Then, the algorithm identifies the states with the same properties according to the object constraint language specifications in different system scenarios. Finally, one statechart is provided by combining same states of different scenarios into one state.

Presenting multi-agent systems using behaviour models is an economical approach that can model the systems precisely. Since behaviour modeling is a complex methodology, it takes a longer time for the designers to perform. Therefore, several methodologies are proposed to provide behaviour models [51,52]. These automated techniques are shown to be effective in analysing system agents [52]. As each scenario indicates specific partial system behaviours, in order to analyse the whole system behaviours, all the scenarios are considered at the same time [53]. However, considering all the behaviours of the agents simultaneously can cause unexpected system faults [31].

In [31, 54], MSCs and hMSCs are used to synthesize system behaviour models. An algorithm, called labeled tarnsition system analyzer, is proposed to perform system behaviour analysis automatically in [54]. This automated algorithm uses behavioural model specifications to analyse system behaviours. This technique is claimed to be effective in detecting the negative scenarios, i.e. scenarios that are not expected by system specifications. However, no case study is presented to demonstrate its effectiveness.

In [29], for detecting unexpected software system behaviours that are not considered in system scenarios, non-local branching choice is used. In [55], it is shown that when several agents can send the first messages in different scenarios of non-local choice, the resultant behaviour is not a desired system behaviour.

In order to analyse the system component behaviour, model-based specifications are used. Model-based testing clarifies the test sequences by test cases for implementations [29]. Considering that scenarios are incomplete specifications, using state machines can provide a complete specification. Therefore, in [29, 55], in order to convert scenarios and state machines to test sequences, TeStor algorithm is proposed.

To produce sequence of events for improving model-based testing, component-based software system verification is proposed in [56]. This verification method verifies the properties of software architectures. System executions are obtained in the first level of this methodology. Then, these executions are converted to message sequence charts for component-based software architecture verification using model checker methodologies. Although this methodology is claimed to be effective in verifying MAS, there is no case study provided.

Considering several system components at the same time makes the behaviour analysis complicated [57, 58]. This problem can be solved for one system component (agent) without considering message lablels. However, when more than one component are considered, the problem becomes complicated [59, 60]. Since MSCs cannot deal with testing specifications, MSGs can provide detection of unexpected specifications [61].

In [28, 61], a technique is proposed to prepare MSG specifications and analyse the software system specifications using MSGs. This methodology is considered as a comprehensive MSG tool for system specification analysis. It can also handle the problems of model checking and unexpected system behaviours. This method is implemented for MSGs while the application for MSCs is not presented.

Agent UML (AUML) is extended over UML to describe interactions among agents in MASs. MASs can be represented by different AUML diagrams. Since most of the existing MAS verification techniques are designed based on UML, some of these AUML diagrams may cause problems in the verification procedure [62]. In [63], a formal methodology is proposed for representing agent interactions of MASs. This methodology converts the interactions among agents described by AUML to object-oriented language Maude [62, 63]. Then, a tool is proposed to use Maude language to verify and model the concurrent system specifications. In [64], a complete description of agent communications is provided using Maude language and model checking is applied for verifying software systems [64]. The disadvantage of this technique is however lack of verification of the resultant Maude descriptions.

Collecting software system requirements is an important step in developing software systems. Scenario-based specifications are used to gather distributed system requirements. Since there is no central control in most distributed systems (similar to multi-agent systems), scenarios are used to show interactions among system components [24]. An automated methodology is proposed in [25] to detect unexpected behaviours in the components of distributed systems. This method utilizes scenario-based specifications to analyse system requirements. Since scenarios represent some information about local components, modeling the behaviours of distributed systems using scenarios is a complicated step. Therefore, in [24, 25], a technique is proposed for analysing the behaviours of system components by converting scenario specifications to state machines. When the sizes of software systems increase, analysing them manually cannot follow the design procedure to detect unwanted runtime behaviours [65].

In [42, 66], for detecting the sources of unexpected behaviours, an automated algorithm is proposed. This algorithm produces the statecharts for each system component in each different scenarios. Then, all the statecharts corresponding to each system component are merged to obtain a single statechart. In the next step, the similar states in these statecharts are identified and called identical states. In [67], the existence of these identical states in the statecharts is shown to be the source of unexpected behaviours in distributed systems [67]. Therefore, the authors proposed a technique for detecting and removing the identical states automatically. This technique has two shortcomings: (i)

This technique is a component-level approach meaning that it analyses the behaviours of system components individually and the behaviours are not modeled when considering the distributed system as a whole. (ii) This technique catches all the identical states in the statecharts of different components. However, as it will be discussed in Chapter 5, not all the identical states cause unexpected behaviour and merging all the identical states may cause overgenralization issue.

When employing a component-level approach for software system verification, some unexpected behaviours and system scenarios may be ignored. This is because in the component-level approaches, the behaviours of each component are considered separately. Therefore, the unexpected behaviours that happen as a result of communications between the system components cannot be detected. In order to synthesize a system-level behaviour model for distributed systems, all the scenarios should be combined [27, 65]. In [27], a method is proposed to combine all the behaviours of system components in different scenarios in order to represent the whole system behaviours.

The first step of this method is producing statecharts for each of the system components. Then, the statecharts corresponding to one component are connected and identical states are identified. Finally, system behaviours are verified by merging these identical states to catch unexpected system behaviours [27]. However, there is no case study presented in [27] to demonstrate the effectiveness of this method. In addition, merging all the identical states in the statecharts may result in new unexpected behaviours caused by overgeneralization.

3.5 Summary

In this chapter, the related works for verifying multi-agent systems are represented. Verifying MASs can be performed using different verification methodologies such as model checking approaches. Different verification techniques are discussed for MASs and distributed systems designed by unified modeling language, agent unified modeling language, message sequence charts, message sequence graphs, high-level message sequence charts and non-local branching choice methodologies. These different techniques are described and compared in terms of effectiveness and accuracy. Since most of these techniques are component-level approaches, they may fail to provide a comprehensive verification. In addition, the assumptions used in synthesis of behaviour models may produce new unexpected behaviours caused by overgeneralization. These two shortcomings will be discussed and addressed in the following chapters.

Chapter 4

Multi-Agent System Design Using AUML

4.1 Introduction

Several methodologies are proposed for design of Multi-Agent Systems (MAS) [1,6,19]. UML is an accepted object-oriented language for analysing system components. In several works, using UML for MAS design has been investigated. The common shortcoming of these works is that UML cannot support the interactions among agents. In order to deal with this problem, AUML formalism which is an extended version of UML is proposed [15]. In this chapter, AUML formalism is utilized to design the interactions among agents. The aim of this research is to perform MAS verification using AUML. Since most of the existing techniques for software verifications are proposed based on UML [4,5], in this chapter, a set of conversion rules are proposed to convert AUML designs to UML notations in order to prepare them for verification.

The rest of this chapter is organized as follows: In Section 4.2, the Agent Unified Modeling Language (AUML) is presented for MAS design. An industrial case study is defined in Section 4.3 to describe the procedure of AUML formalism for designing multi-agent systems. In Section 4.4, Agent Unified Modeling Language (AUML) formalism steps are presented and applied to the case study. The proposed method to convert AUML sequence diagrams to UML sequence diagrams to be used for MAS verification is represented in Section 4.5. Finally, this chapter is summarized in Section 4.6.

4.2 Agent Unified Modeling Language (AUML)

In order to reduce the risks in modern industrial applications, Multi-Agent System (MAS) design methodologies are vital. Reducing risks can be achieved by proposing effective tools and extending accepted methods to take advantage of the existing methods [15]. For denoting agent interaction protocols in multi-agent systems, several modeling techniques are proposed [15, 28, 30].

The Unified Modeling Language (UML) is a known and approved representation for object-oriented software development [23, 24]. However, lack of common features between objects and agents makes UML not sufficient for MAS design applications. In [6, 15, 19], Agent UML (AUML) is proposed to address the challenges in agent-based software methodologies. AUML is extended over UML due to the increasing acceptance of UML for object-oriented software development.

Agents conform and collaborate to each other by exchanging information in a typical MAS. This information exchange is performed based on interaction protocols which are extracted from communication protocols used in MASs. Since setting up these protocols is a fundamental task in MAS design, the first step of AMUL is dedicated to defining interaction protocols [19].

AUML formalism effectively shows the flow of information between agents. This process ranges from planning and analysis to design of system structure and maintenance [15]. Considering that UML sequence diagrams are the most commonly used techniques for synthesis of behaviour models, in this research, a technique is proposed to convert AUML notations into the UML sequence diagrams. UML sequence diagrams include Message Sequence Charts (MSC) and message sequence diagrams that represent the agents as system components, and the interactions between them as messages.

In the following sections, the AUML steps are performed for indicating agents' interactions. In order to better illustrate this procedure, first, a multi-agent system case study is introduced and then, it is designed using AUML formalism.

4.3 Multi-Agent System Case Study

In order to better demonstrate the procedure of AUML formalism, an industrial case study of a multi-agent system called Real-Time Fleet Management System is considered. This joint case study project has been negotiated with the Encom Wireless and City of Calgary to design, implement and evaluate a multi-agent simulation system for Commercial Vehicle Enforcement. In this thesis, a small scale prototype of this project is focused as a proof of concept. The specifications of this multi-agent system case study are described in the following subsections.

4.3.1 Goals

The objective of real-time fleet management system is to provide minimum delivery time by considering several different conditions which influence it. Authentic sensors such as location, weather and traffic condition sensors are needed for computing the accurate delivery time. Moreover, to forecast and update the delivery time, a dynamic timing model is used. Informing the customers by updating web page and sending text messages are the other features of the real-time fleet management system. Furthermore, to achieve customer satisfaction, a dynamic, user friendly and secure web interface is employed.

4.3.2 System Agents

Real-time fleet management system consists of several components (agents). "Weather Rec." is a sensor that collects the information about the weather condition. "Traffic Rec." agent collects the information about the traffic using the traffic cameras located at the intersections. The third sensor is "GPS Rec." which collects information about the location of the delivery cars. These three agents only interact with a single agent which is called "Radio". "Radio" agent acts as an information station that collects all the external world information and sends them to another system agent named "Server" upon request.

"User" agent includes an interface with the customer that starts a delivery request by sending the customer location to the "Server". "Server" agent's main task is to calculate the delivery time. However, it first interacts with "Selector" agent to locate the closest available branch. "Web Interface" and "SMS" (Short Message Service) agents interact with the customers to inform them about the delivery time.

4.3.3 System Functions

Considering that real-time fleet management system is a multi-agent system, all of its agents work independently and take responsibility for their tasks while there is no central controller. However, they interact with each other to transmit the necessary information in order to achieve the goal of minimizing delivery time and updating the customers in a fast and user friendly manner.

For real-time fleet management system, there exist several different scenarios to show how agents interact with each other to obtain minimum delivery time. One of these scenarios is considered in this chapter for representing different steps of designing MASs using AUML methodlogy. In the following chapters, the other scenarios are also considered.

4.4 Agent Unified Modeling Language (AUML) Formalism Steps

In this section, the steps of AUML formalism are discussed in detail and performed for the considered case study, real-time fleet management system. In Figure 4.1, the goal of real-time fleet management system is shown with hierarchical model. As presented in

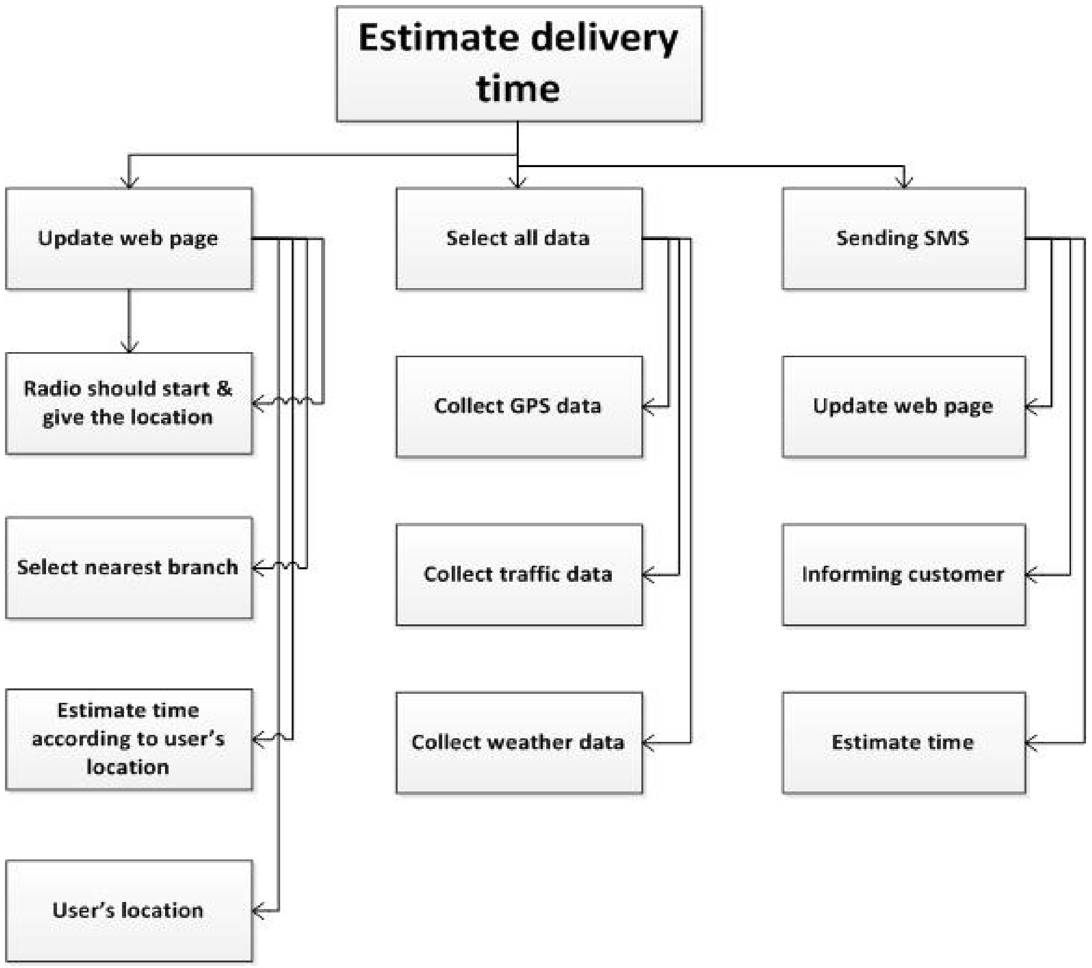

Figure 4.1: Goal hierarchical model

this figure, "Updating web page", "Select all data" and "Sending SMS" are considered as sub-goals that are requirements for calculating minimum delivery time and informing customers.

There are three levels for designing multi-agent systems using AUML formalism: In the first level, packages and templates are provided. In the second level, that is called inter-agent representation, the interactions among agents are expressed. Finally, in the third level, the internal agent processing is shown, i.e. intra-agent representation [18]. To better illustrate how the AUML formalism is performed, these three levels are represented in detail in the following:

Level 1: Components and packages are defined by UML in order to denote object-oriented structures. The purpose of defining component is to combine classes to achieve the goals of software systems [15]. In order to group the components of the system, packages are used. For each scenario, packages are defined based on data and requests. However, sometimes not all the packages are demanded by users. Therefore, only some particular packages are used according to the users' requests.

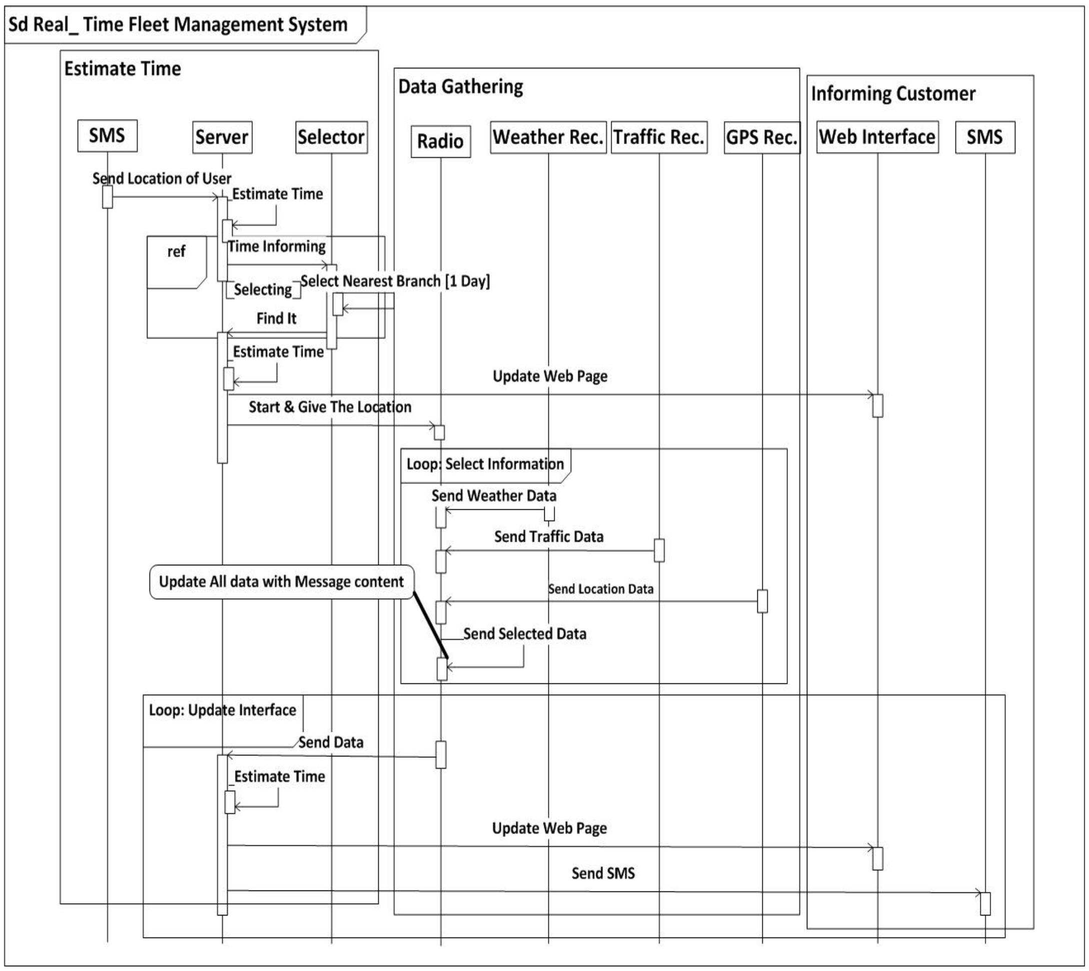

Figure 4.2: Producing packages for AUML sequence diagram

In Figure 4.2, for the real-time fleet management system case study, three packages are produced: ”Estimating Time”, ”Data Gathering” and ”Informing Customer”. For example, customers may not need to know about how data are gathered for estimating time and only need to know about the delivery time. Therefore, ”Data Gathering” package is not needed by the customers in this example.

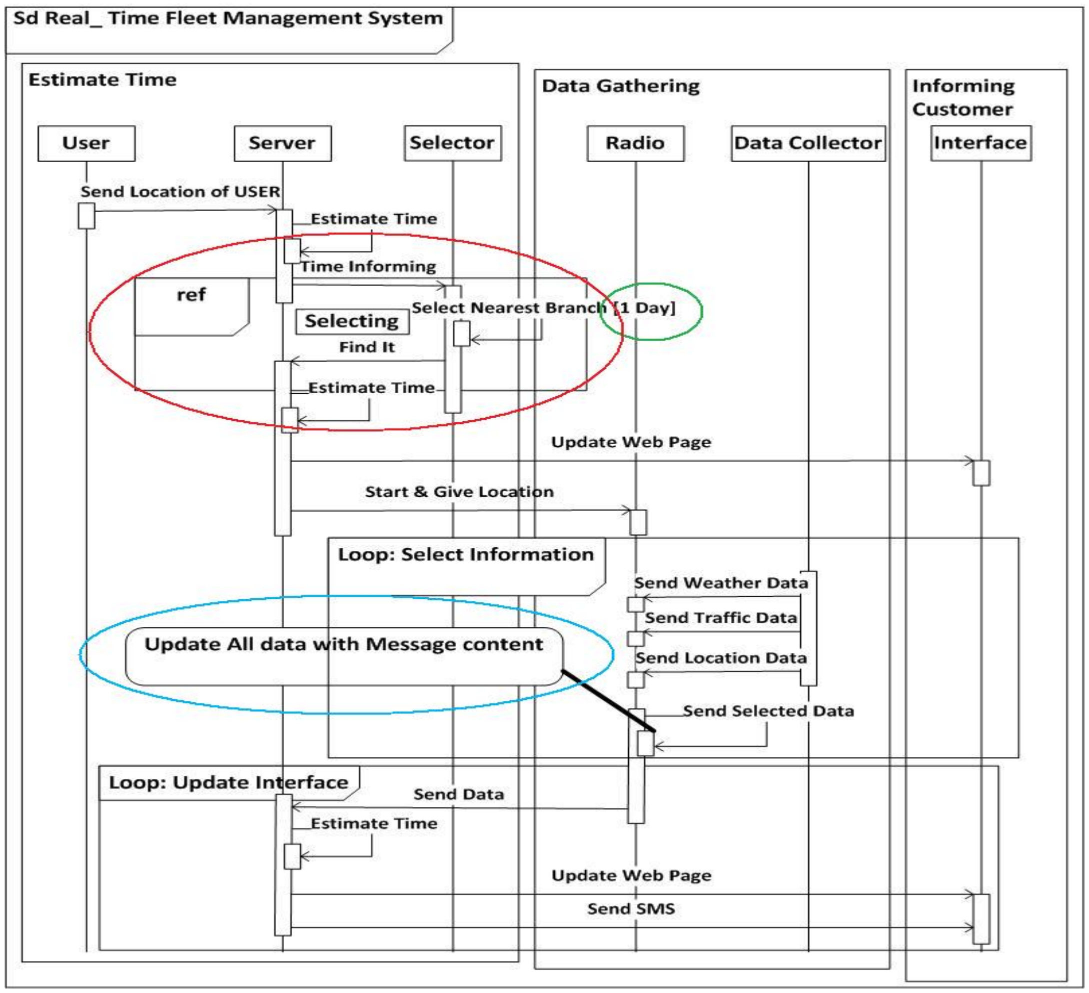

Figure 4.3: Producing roles for AUML sequence diagram

Level 2: In order to show the interactions between the agents, interaction diagrams are used. Sequence diagrams, collaboration diagrams, activity diagrams and statecharts are used for describing the interactions among agents [15]. The sequence diagrams focus on the order of message passing. Since UML is designed for object-oriented designs, it is not adequate for representing the agent interactions because of lack of common features between objects and agents. Therefore, in AUML, it is proposed to extend UML to show the interactions among agents. One of these extensions is "role" that is used to categorize agents based on their behaviour. Defining roles leads to decrease in the size of interaction diagrams [19].

In Figure 4.3, the roles are produced for the case study of real-time fleet management system. In this example, Weather Rec., Traffic Rec. and GPS Rec. agents are combined into "Data Collector" role while Web Interface and SMS agents are combined into "Interface" role. There exist several other AUML extensions that will be discussed in Section 4.5.

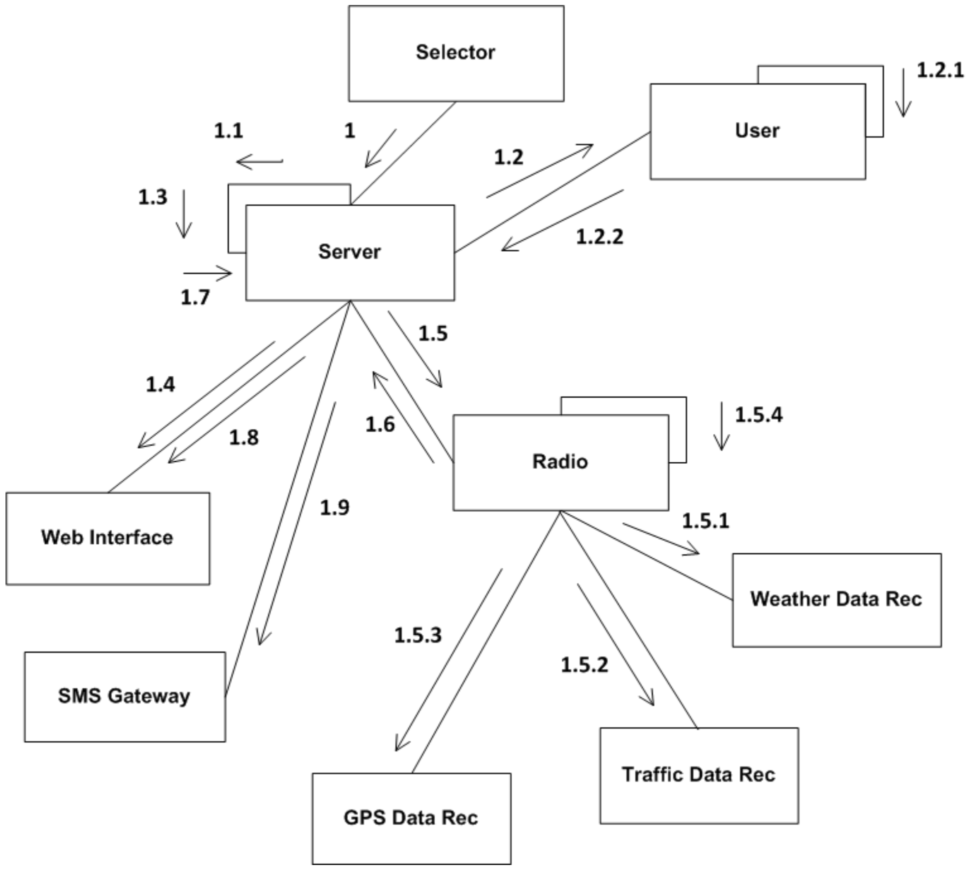

Figure 4.4: AUML collaboration diagram

The goal of collaboration diagrams is showing the relations between the agents. Although sequence diagrams and collaboration diagrams are similar semantically, in collaboration diagrams, agents are not placed in a specific way and the interactions are numbered in a top-down order. The domain experts usually make the decision to use sequence diagrams or collaboration diagrams based on their capabilities in their specific applications [18].

The collaboration diagram for the real-time fleet management system case study is shown in Figure 4.4. Since the agents' locations are chosen arbitrarily, numbers are added to messages to show the order of interactions among agents. First, the initial message is added to the collaboration diagram and labeled 1. Then, the messages that come after, i.e. nested messages, are numbered using the label of the first message and the number of the nested message separated by a dot. For example, in Figure 4.4, the collaboration diagram starts with message 1 which is the "send location of user" (see Figure 4.3) and its following messages are considered as nested messages. For instance, the message "estimate time" (see Figure 4.3) is a nested message for message 1 and is shown by 1.1.

Activity diagrams are one of the simple diagrams in UML because of the similarity between their symbols and flowchart notations. By using activity diagrams, the process of achieving goals is defined in an understandable way. Activity diagrams are used to provide a simplified representation of the behaviour model. In these diagrams, agent actions and the events that trigger them are presented [15, 18].

As shown in Figure 4.5, the activity diagram is started with an initial point shown by a filled circle and is terminated by a final node shown by two concentric circles where the inner one is a filled circle. The agents are shown in between the initial and final nodes. In order to show the interactions between the agents, arrowed edges are used [16].

State machine diagrams are represented by statecharts. The statecharts do not rep-

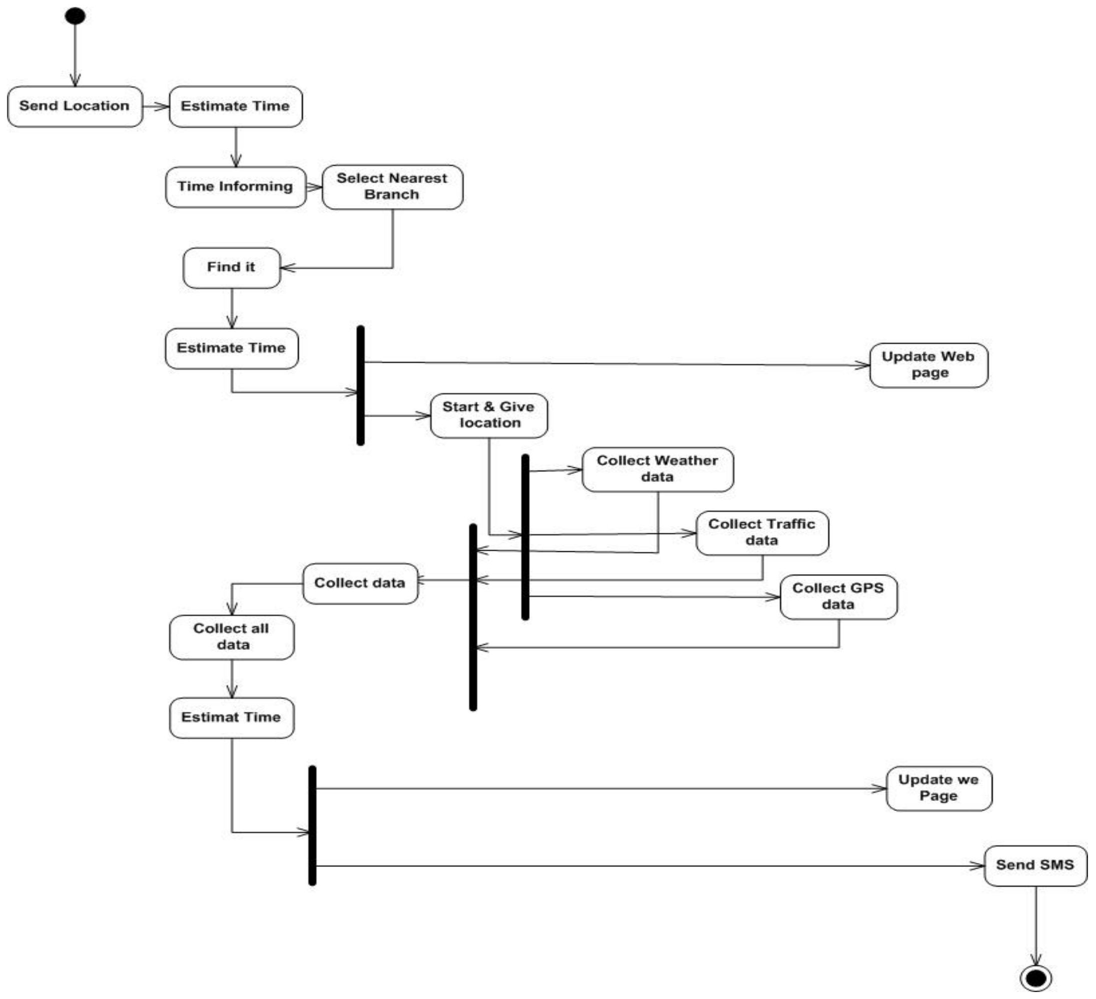

Figure 4.5: AUML activity diagram

resent the interactions among agents. The main focus of statecharts are the agents. As shown in Figure 4.6, states are denoted by circles. In order to connect the states, directed lines are used.

Since in the real-time fleet management system two agents "Server" and "Radio" are the most interactive agents, statecharts are produced for them. In Figures 4.6 and 4.7, these statecharts are shown for "Server" and "Radio", respectively. However, statecharts for the other components can be produced in a similar way.

Level 3: At the final level of AUML formalism, the internal processes in each agent

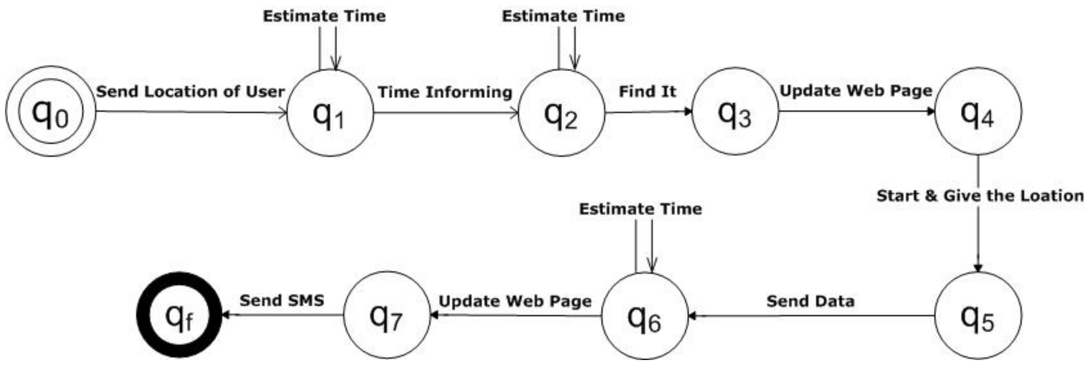

Figure 4.6: State machine for agent "Server"

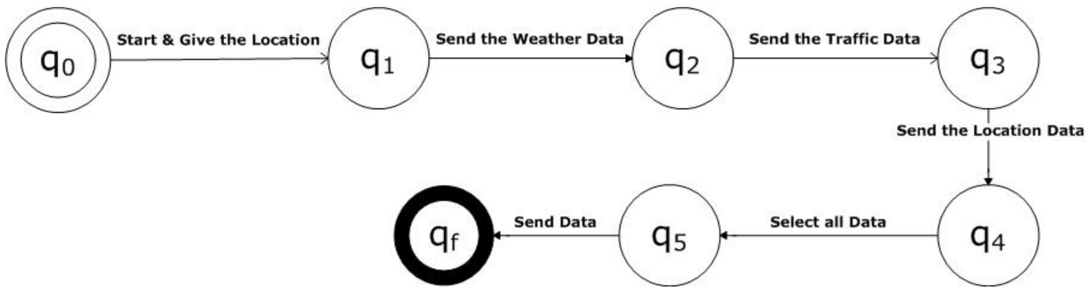

Figure 4.7: State machine for agent "Radio"

are described. Each of the diagrams discussed in level 2 can be used to show the interactions that occur in a specific agent. Usually in order to represent the internal agent (intra-agent) processing, UML statecharts are used [15].

Verification of multi-agent systems is the goal of this thesis. Since interactions among agents are important to model system behaviour, AUML sequence diagrams are more effective than the other AUML diagrams. Also, AUML sequence diagram of the provided case study, shown in Figure 4.3, is considered for validation of the proposed algorithm.

4.5 The Proposed Method to Convert AUML Sequence Diagrams to UML Sequence Diagrams

In [25,65], an effective technique for MAS verification is proposed which is based on converting scenario specifications to state machines. This technique is shown to be effective in verification process automation of UML scenario specifications. However, in [25], the scenarios are expected as the algorithm input and no methodology is proposed for producing MAS scenarios. In this research, this crucial step is also considered and AUML formalism is used to provide comprehensive agent-based designs. The resultant AUML scenario specifications will be used as the input of MAS verification process.

Considering that UML is a well-accepted design language, most of the existing verification tools, e.g. [4,25,65], are designed for UML sequence diagrams. The UML diagrams produced using UML are not however adequate for MAS designs. In this work, AUML formalism is employed to prepare AUML sequence diagrams that are able to represent agent-based designs appropriately. However, in order to maintain the flexibility of the verification technique, in this thesis, a set of conversion rules are proposed to convert these AUML sequence diagrams to UML sequence diagrams. These proposed conversion rules are used to convert the AUML extensions to the accepted UML notations. The resultant UML sequence diagrams, that are indirectly produced by AUML formalism, not only consider all the aspects of agent-based designs but also can be verified using the existing MAS verification techniques.

In order to convert the AUML sequence diagrams to UML sequence diagrams, a set of conversion rules, presented in Table 4.1, is proposed. Since AUML is an extension to UML, all the UML notations existing in the AUML sequence diagrams do not need to be converted. For example, combined fragment and termination which are used in AUML sequence diagrams to show the interactions among the agents are accepted UML notations

Table 4.1: The proposed conversion rules from AUML to UML.

AUML Extended Notations	In UML Sequence Diagrams
InteractionOccurrence	Add requested agents and messages between them to the caller diagram
Gate	Message between two partial scenarios in high level MSC
Continuation: 1. Same agent 2. Other agent in the same scenario 3. Other agent in another scenario	 1. Add messages to itself 2. Add messages to the other agent 3. Add messages to the other scenario
Constraint: 1. Blocking/nonblocking 2. Timing Constraint	 1. Add messages in order to show conditions 2. Considering as "ref" fragment that refers to a timing diagram
Action	Add message to itself

and do not need to be converted. However, there are some differences between AUML and UML sequence diagrams which are called extended notations [21]. Considering that these extended notations are only defined for AUML to represent the agent interactions, the conversion rules are defined for them in Table 4.1 and are explained in the following subsections.

4.5.1 InteractionOccurrence

When an interaction diagram calls another interaction diagram, it is called InteractionOccurrence [22]. The InteractionOccurrence is represented by a rectangle labled by "ref" in AUML sequence diagrams. The name of InteractionOccurrence is written in this rectangle. To deal with InteractionOccurrence in UML sequence diagrams, the called interaction diagram is added as a set of agents and messages to the caller interaction diagram. These agents are only added when InteractionOccurrence happens and may terminate after the call. Figure 4.8 represents an example for converting InteractionOccurrence in AUML sequence diagram, shown by "ref" rectangle, into accepted UML notations. In this ex-

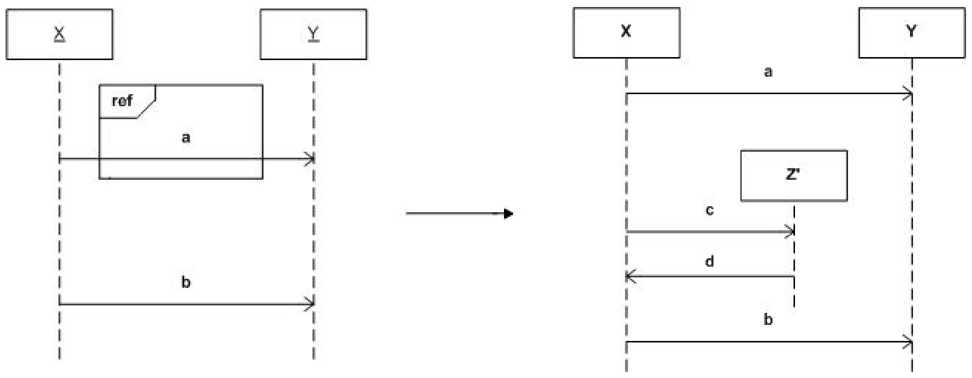

Figure 4.8: Example for conversion of InteractionOccurrence

ample, the called agent "Z′" and its corresponding messages "c" and "d" are added to the UML sequence diagram in order to deal with InteractionOccurrence.

4.5.2 Gate

To associate the inner messages of the current interaction diagram with messages that are in another interaction diagram, a connection point named Gate is introduced in AUML [21]. Therefore, it is proposed to convert Gate to a message that connects two partial scenarios in the high-level message sequence charts. It should be mentioned that connecting two partial scenarios in high-level message sequence chart is one of the popular approaches for modeling multi-agent systems [30]. In Figures 4.9 and 4.10, an example for converting Gate in AUML sequence diagram to the accepted UML notations is presented. In this example, Gate is added to represent that the message "f" in AUML sequence diagram in Figure 4.9(a) is associated to the message "u" in AUML sequence diagram in Figure 4.9(b). To convert AUML Gate to the accepted notations in UML sequence diagrams, the message "f" should be connected to message "u". Therefore, in Figure 4.10, these messages are connected to each other by using an extra message "Connected Message" that connects two partial high-level message sequence charts.

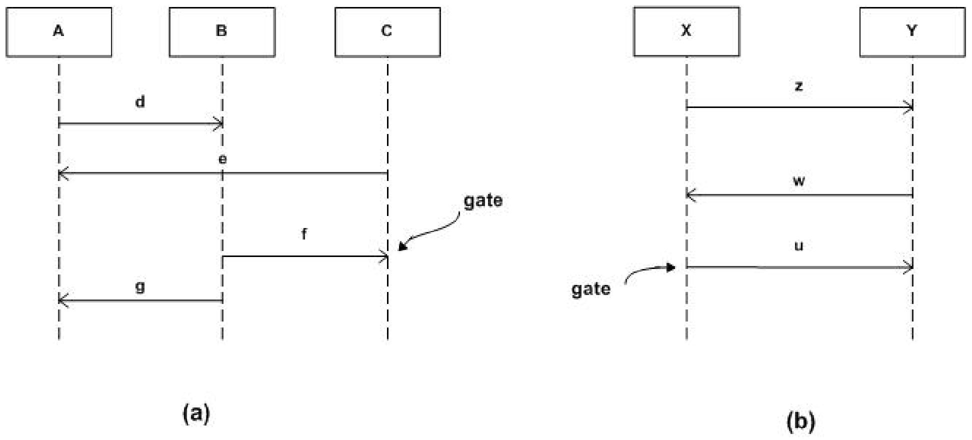

Figure 4.9: Example of Gate in AUML sequence diagram

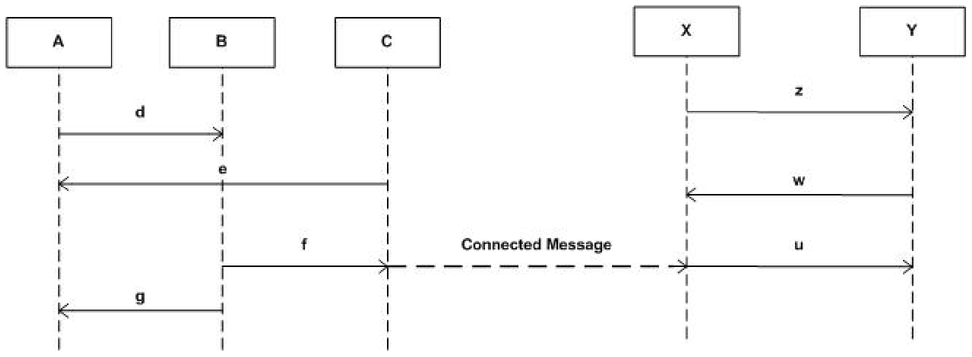

Figure 4.10: Gate converted to an extra message in UML sequence diagram

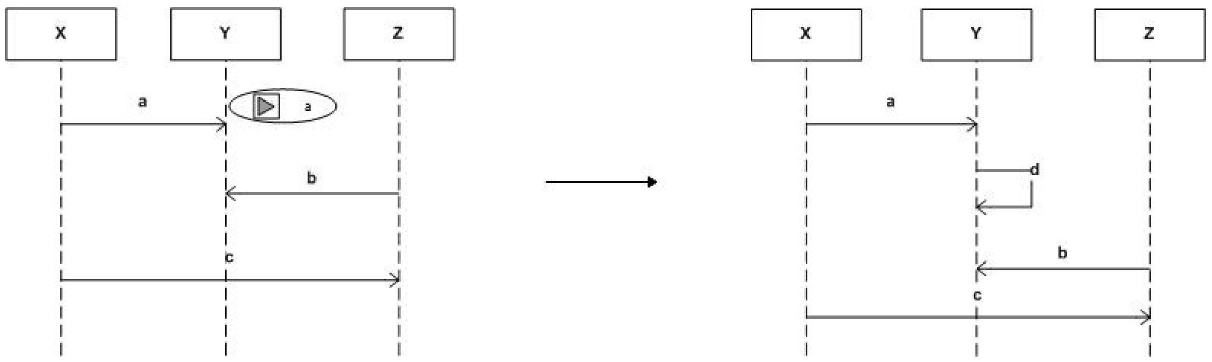

Figure 4.11: Example for Continuation for one agent

4.5.3 Continuation

Continuation is defined to show an intermediate point in a CombinedFragment or a control flow [22]. There are three types of continuation: continuation for only one agent, continuation between two agents in the same scenario and continuation between two agents in different scenarios. To deal with the continuation in UML sequence diagrams, it is proposed to convert it to messages from an agent to itself for the first type, messages between two agents in the same scenario for the second type and messages between two agents in two different scenarios for the third type. Figure 4.11 represents an example for continuation for one agent. In this example, the continuation is represented for message ”a” of agent ”Y” in the left hand side AUML sequence diagram. This continuation is converted to the message ”d” from agent ”Y” to itself in the right hand side UML sequence diagram.

In Figure 4.12, the continuation for two agents in the same scenario is presented. In the left hand side AUML diagram, the continuation is used between two agents ”Y” and ”Z”. According to the proposed conversion rules, in the right hand side UML diagram, the continuation is converted to message ”e”.

Figure 4.13 presents continuation for two agents in two different scenarios. As shown

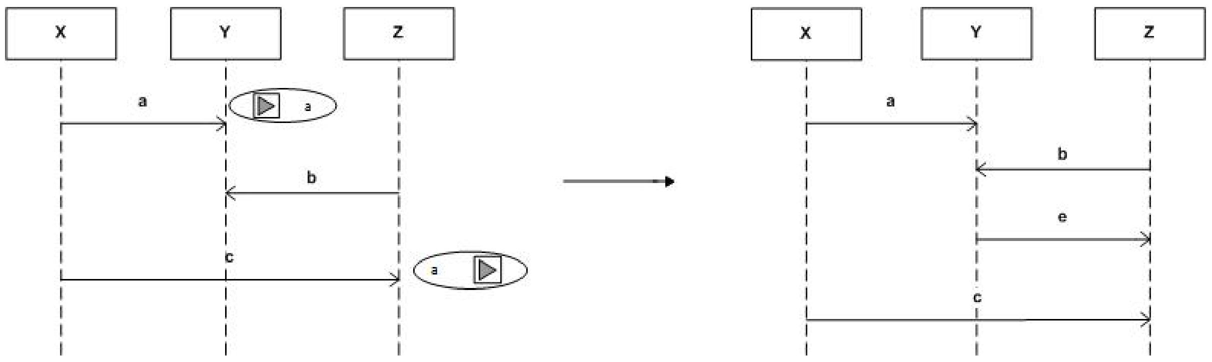

Figure 4.12: Example for Continuation for two agents in same scenario

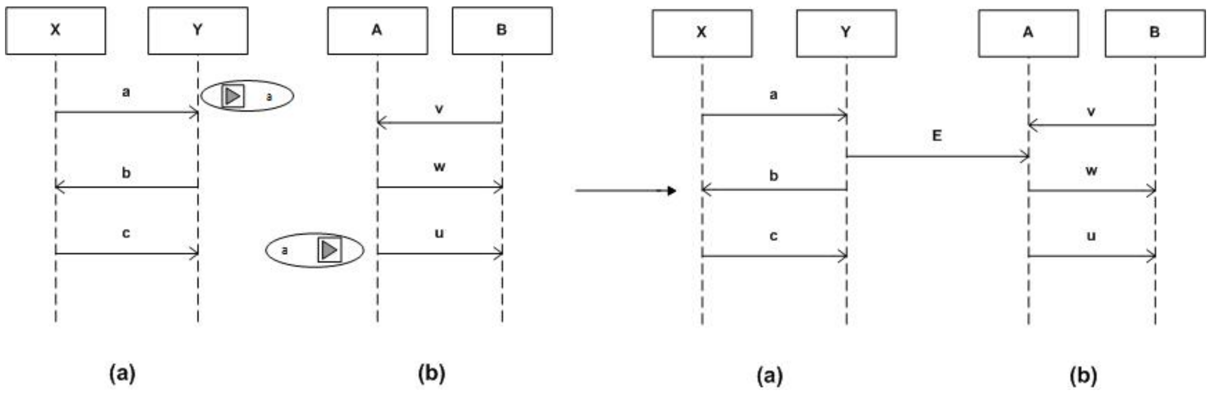

Figure 4.13: Example for Continuation for two agents in two different scenarios

in the AUML diagram (left hand side diagram), the continuation is between agents ”Y” and ”A” in scenarios ”(a)” and ”(b)”, respectively. Since in this example continuation represents an intermediate point in the control flow between two scenarios, a message ”E” is used to convert AUML sequence diagrams to UML sequence diagrams.

4.5.4 Constraints

Two types of constraints, blocking/nonblocking and timing constraints, are defined to show conditions that should hold for an event to happen [22]. Nonblocking constraints just affect a certain event and do not affect the other agents and following messages.

On the other hand, the blocking constraints may stop the other agents and following messages if the constraint does not hold. In this work, it is proposed to convert these AUML constraints to one or more messages in UML sequence diagrams that determine which messages and agents can perform actions. AUML timing constraints set a time interval in which an event can happen. Therefore, these AUML timing constraints are converted to reference operators in UML sequence diagrams in order to call the UML timing diagrams.

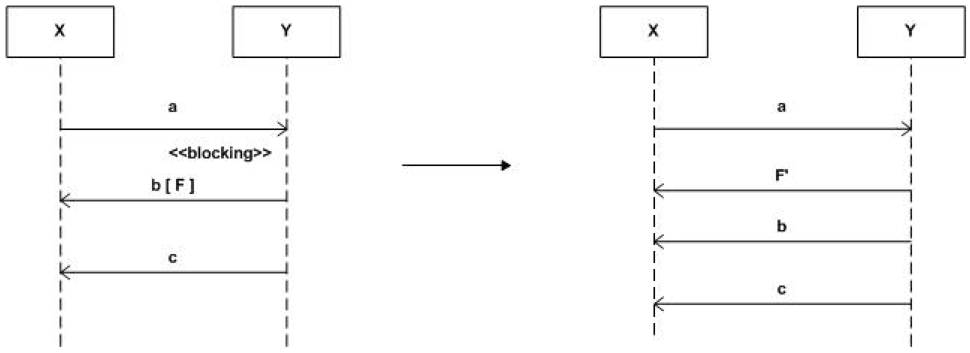

Figure 4.14: Example for blocking constraint

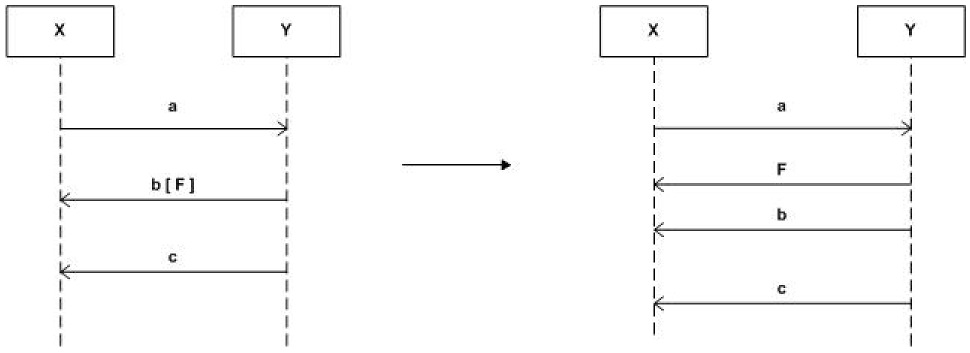

Figure 4.15: Example for nonblocking constraint

In Figures 4.14 and 4.15, the blocking and nonblocking constraints "[F]" for message "b" are shown, respectively. The only difference between representing blocking constraint and nonblocking constraint is a notation "≪blocking≫" on the top of message "b[F]". In this example, if the blocking constraint "[F]" in Figure 4.14 does not hold, not only message "b" cannot be sent but also the following message "c" cannot be sent. However, the nonblocking constraint in Figure 4.15 only prevents message "b" while the following message "c" can be sent even if the constraint does not hold. In order to convert AUML sequence diagrams to UML sequence diagrams, these constraints are converted to messages "F'" and "F" in right hand side diagrams of Figures 4.14 and 4.15, respectively. Note that the blocking and nonblocking type of the constraint is reflected in the content of the converted message in the UML sequence diagrams.

Figure 4.16 represents an example for timing constraint. Timing constraint in this example indicates that the event corresponding to message "b" should happen in a 24 hour interval. In order to convert AUML sequence diagrams (left hand side of Figure 4.16) to UML sequence diagrams (right hand side of Figure 4.16), this constraint is considered as a reference fragment that refers to a UML timing diagram.

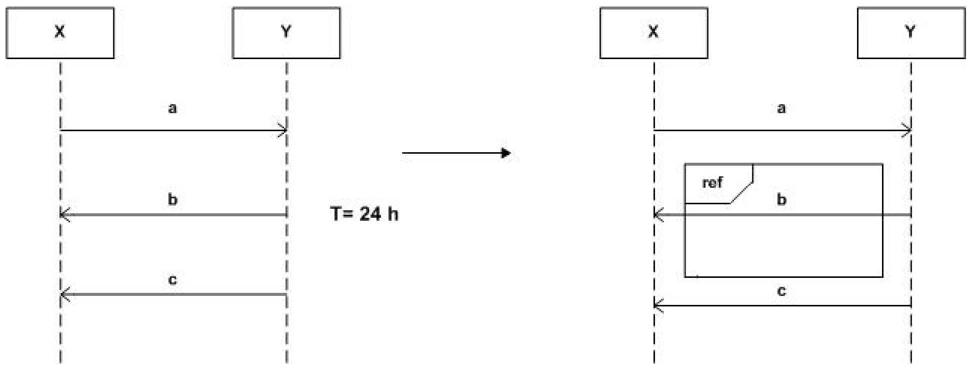

Figure 4.16: Example for timing constraint

4.5.5 Action

In AUML, action is denoted to show how an agent should interact with another agent [21]. In this work, it is proposed to convert action in AUML sequence diagrams to a message from the agent to itself based on the content of the action in UML sequence diagrams.

In Figure 4.17, action "D" is shown for message "c" sent by agent "X" in the left hand side AUML sequence diagram. This action is converted to message "C$'$" sent by agent "X" to itself in the right hand side UML sequence diagram.

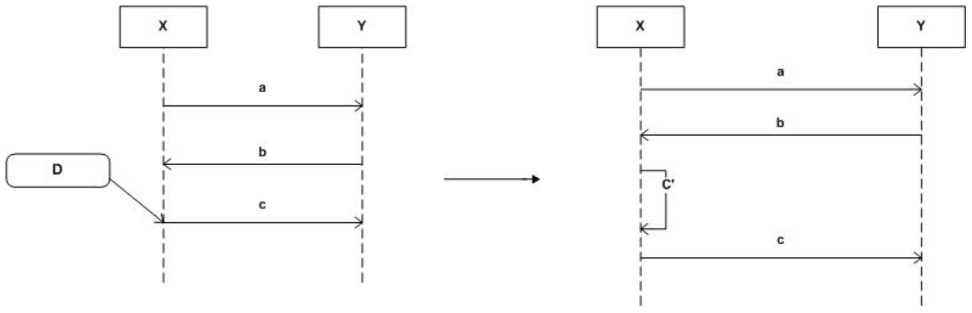

Figure 4.17: Example for action notation

4.5.6 Applying the proposed conversion rules to the multi-agent system case study

To show the effectiveness of the proposed conversion rules, the case study of real-time fleet management system is considered. In Figure 4.3, the AUML sequence diagram for this case study is shown which is produced using AUML formalism. In this figure, the extended AUML notations are marked by red, blue and green ellipsoids. This AUML sequence diagram comprehensively models the interactions among the agents. The proposed conversion rules are then employed to generate the UML sequence diagram of this case study as shown in Figure 4.18 where the extended AUML notations are converted to accepted UML notations. For an example, in Figure 4.3 (AUML sequence diagram), "selecting" diagram, shown by a red ellipsoid, is an InteractionOccurance and is shown by a reference fragment. In Figure 4.18 (UML sequence diagram), this InteractionOccurance is replaced by the "Branch Information" agent and its related messages added to the UML sequence diagram, marked by a red circle.

In Figure 4.3 (AUML sequence diagram), the nonblocking constraint "1 Day" is enforced, marked by green ellipsoid. According to the proposed conversion rules, this constraint is converted to the message "Delivery Time Less Than 1 Day", marked by green ellipsoid, in the corresponding UML sequence diagram shown in Figure 4.18. Finally, in Figure 4.18 (UML sequence diagram), the message "Update Selected Data", marked by a blue ellipsoid, is used to replace action "Update All data with Message content" in Figure 4.3 (AUML sequence diagram) that is surrounded by a blue ellipsoid.

The resultant UML sequence diagram in Figure 4.18 is resulted from converting an AUML sequence diagram and is indirectly produced using AUML formalism. Therefore, this UML sequence diagram not only considers all the aspects of agent-based designs but also can be verified using the existing MAS verification techniques.

4.6 Summary

Several methodologies are proposed to address the increasing requests for multi-agent system designs. In AUML formalism, the interactions among agents are defined as an extension to UML since UML is originally designed for object-oriented software development. Considering that most of the accepted MAS verification techniques are based on UML sequence diagrams, scenarios that are produced using AUML formalism should be converted to UML. In this work, AUML is used to design MAS and a set of conversion rules are proposed to convert AUML diagrams into UML. The resultant UML sequence diagrams can handle the interactions among agents and can be used for MAS verification. This proposed technique is validated by applying to a real-world case study.

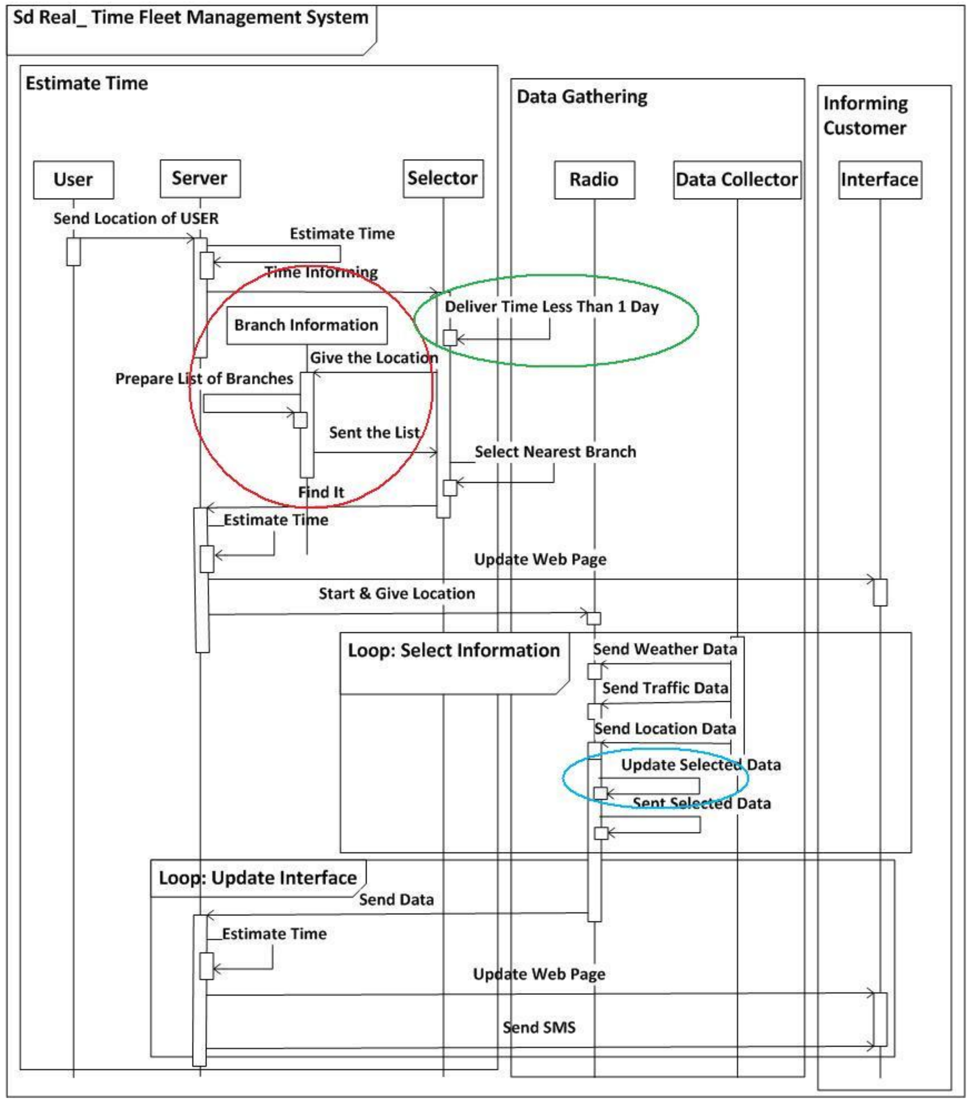

Figure 4.18: Converted UML sequence diagram for real-time fleet management system

Chapter 5

Detecting Emergent Behaviour in Multi-Agent Systems Caused by Overgeneralization

5.1 Introduction

As the demands for agent-based software engineering increase, reducing the risk of malfunctions throughout Multi-Agent System (MAS) design methodologies has become a vital task. Detecting and addressing the unexpected failures in early design stage of MASs can be cost effective. Behaviours of MAS must be verified against the unwanted runtime behaviours, i.e. emergent behaviours. Emergent behaviours are the unexpected behaviours that happen due to the partial nature of scenarios. In other words, since several scenarios are used to describe behaviours of a multi-agent system, some behaviours (emergent behaviours) may happen that are not expected by the scenario specifications and should be detected in the verification procedure.

Emergent behaviours are usually divided into two categories: The first category includes the emergent behaviours that occur due to the incompleteness of scenarios, i.e. system specifications. The second category contains the emergent behaviours that happen as a result of synthesis of behaviour models, i.e. overgeneralization. In fact, overgeneralization occurs because of the assumptions in the process of behaviour model synthesis.

The methods for detecting emergent behaviours are studied in Chapter 3. However, only a few works [3, 25] have targeted the overgeneralization problem. In this research, a technique is proposed for MAS verification by synthesis of behaviour models from scenarios in order to detect emergent behaviours. The proposed technique is shown to be effective in preventing overgeneralization in the behaviour model synthesis. To validate the proposed technique, a case study of real-time fleet management system is employed for experimentation. It is shown that the proposed technique is effective in detecting the emergent behaviours that are caused by overgeneralization in the scenario specifications of the MAS case study.

The rest of this chapter is organized as follows: In Section 5.2, the emergent behaviours in multi-agent systems are introduced and the shortcomings of different existing methods for detecting them are discussed. An industrial case study is considered in Section 5.3 to illustrate the procedure of synthesis of behaviour models. In Section 5.4, a method is proposed for MAS verification which prevents the emergent behaviours while overgeneralization is addressed. Finally, in Section 5.5, a summary of the chapter is given.

5.2 Emergent Behaviour in MASs

The specifications of multi-agent systems are usually demonstrated by scenarios because of their explicit nature. Scenarios show system agents and messages that are sent between them. Scenarios are generally shown using Message Sequence Charts (MSCs) or message sequence diagrams [10]. The behaviours of the individual agents and the whole system can be represented using the scenarios. Sequence diagrams are not however enough for analysis of the software system behaviours. A popular approach for analysing the behaviour of system agents is going from scenarios to state machines which is also known as synthesis of behaviour models. State machines that are shown using UML statecharts are used as effective tools for synthesis of behaviour models from scenarios [48, 68]. In this procedure, each agent is considered as a process with several possible states. All the messages that are sent and received by this agent are considered in the corresponding state machine as transitions between the states [3].

State machines are used in order to explicitly model the system behaviours in [25]. One state machine is designed for each agent in each certain scenario of the MAS, where all the interacted messages of that agent are included. Blending all the scenarios used for describing the MAS is necessary since it provides a comprehensive overview of the system behaviour.

Two methods are proposed for combining the scenarios: state identification and scenario composition using high-level message sequence graphs. In state identification [3, 48, 69], the agents of the scenarios are first modeled with different states in the state machines. Then, for each agent, similar states are identified in a set of scenarios. These similar states are combined in different state machines to enable the scenarios to merge. In [10], another approach for merging scenarios is proposed where scenarios are split to smaller parts with lower complexity. Then, high-level message sequence graphs are used to blend the smaller sequence of behaviours since they are simpler to manage.

Merging all similar states to achieve only one state machine for each agent in all scenarios is proposed in [3] in order to improve the synthesis of behaviour models. This technique is shown to be effective in detecting emergent behaviours that are produced as a result of the assumptions in behaviour model synthesis. However, the relationships between scenarios are rather ambiguously defined.

In [25], a technique to identify the identical states that may cause emergent behaviours is proposed and a method to address emergent behaviours by merging identical states is presented. This results in detecting the emergent behaviours due to the presence of identical states. However, although identical states are the potential causes of emergent behaviours, not all identical states may lead to emergent behaviours. The shortcomings of this technique are: first, all identical states are merged for combining the state machines, and overgeneralization is inevitable; and second, it relies on the designer to determine which states may cause emergent behaviour. Considering that ad-hoc methods

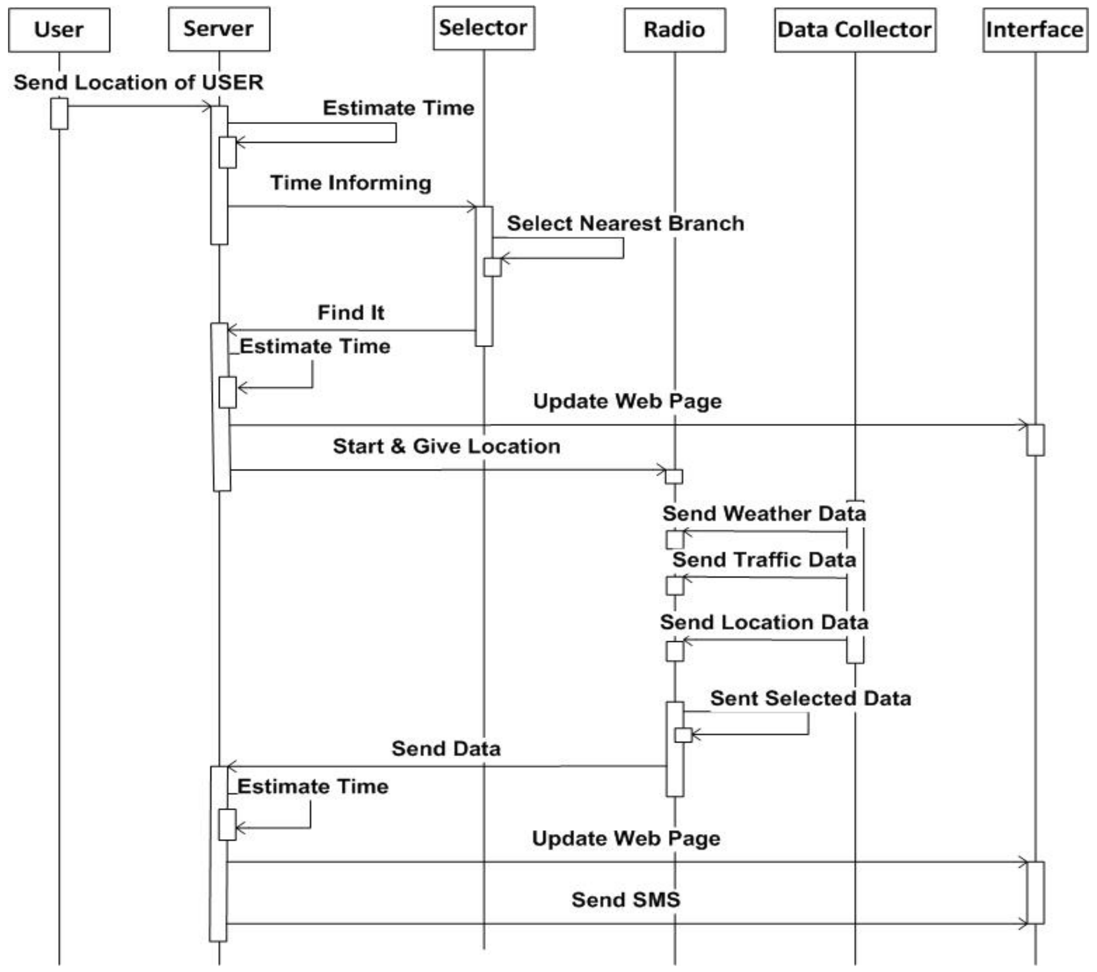

Figure 5.1: Scenario S_1 of real-time fleet management system case study

are not reliable and are time consuming for large scale systems, in this research, a technique is devised to distinguish the identical states that cause emergent behaviours and a new technique is proposed to replace the ad-hoc methodology in [25] into an automated method.

In order to validate the proposed techniques, the real-time fleet management system case study, defined in chapter 4, is used. In this chapter, in order to show the real-time fleet management system behaviours, several scenarios are utilized. In the following paragraphs, these scenarios are presented:

In Figure 5.1, scenario S_1 of the real-time fleet management system is shown. The goal of this system is estimating delivery time and informing the customer. In this scenario, agent "Server" receives the location of customer from agent "User". Agent "Data Collector" gathers all the weather, traffic and location data from the sensors and sends them to the agent "Radio". "Radio" sends all these data to the agent "Server". Then, based on these information "Server" estimates delivery time and informs the customer by updating web page and sending a text message (SMS).

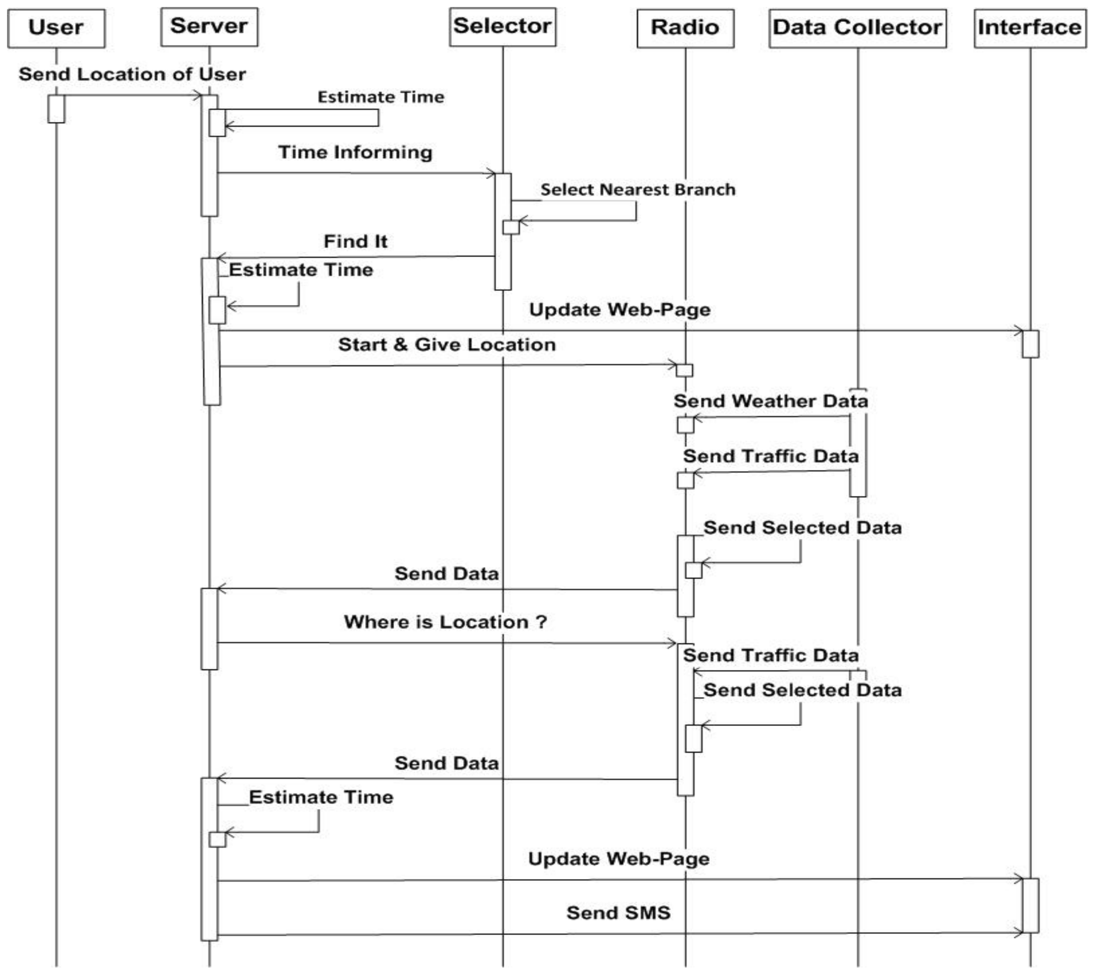

Figure 5.2: Scenario S_2 of real-time fleet management system case study

The second scenario of the real-time fleet management system S_2 is shown in Figure

5.2. This scenario considers a special case where agent "Server" does not receive any information about the location of the delivery vehicle. To deal with this issue, "Server" asks agent "Radio" about location of the vehicle in order to accurately estimate delivery time and inform customer.

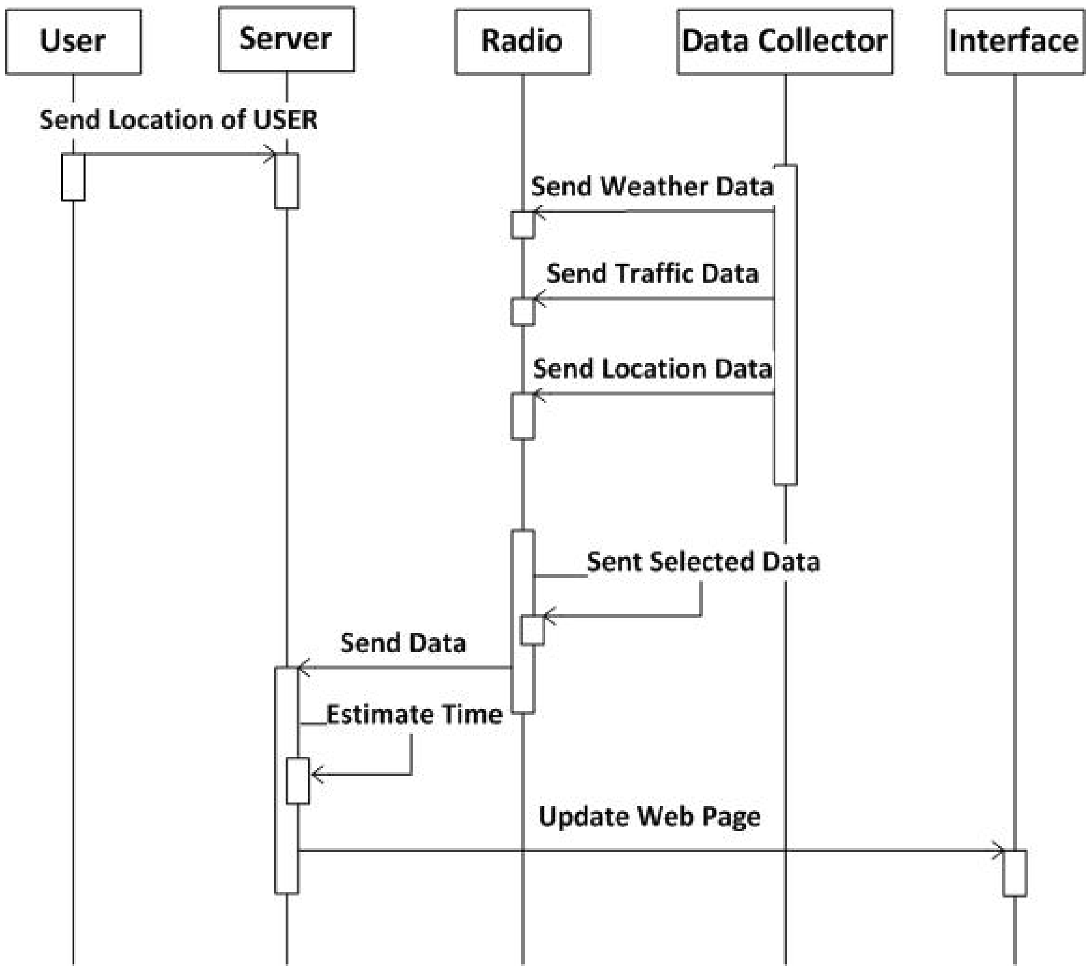

Figure 5.3: Scenario S_3 of real-time fleet management system case study

Figure 5.3 shows another scenario, S_3, representing partial behaviours of the real-time fleet management system. In this scenario, the customer intends to know an estimate of the time it takes to receive any order. Therefore, agent "Server" collects all the information about weather, traffic and location from agent "Data Collector", estimates the delivery time based on them and updates the web page.

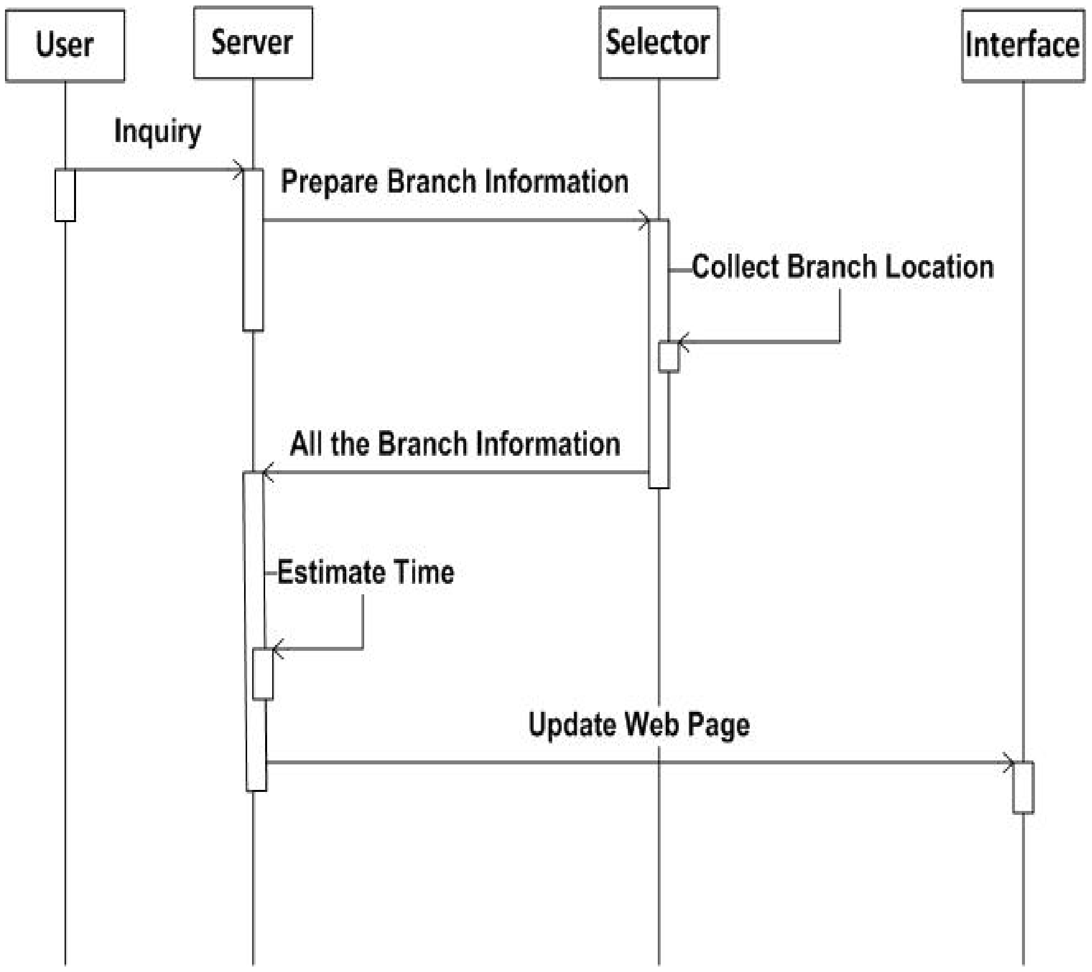

Figure 5.4: Scenario S_4 of real-time fleet management system case study

In the next scenario, S_4, shown in Figure 5.4, agent "user" sends message "Inquiry" to inquire about the availability of a specific product and request for a rough estimate of the delivery time. Therefore, "Server" interacts with agent "Selector" to find the branches with availability and roughly estimates the minimum delivery time without considering more detailed information such as weather and traffic data. Finally, the customer is informed by updating the web page.

The scenarios discussed above describe a set of partial behaviours of the case study.

These scenarios will be verified in the following sections to detect the unwanted emergent behaviours that are not expected by the designer.

5.3 Behaviour Models

Detecting and addressing faults of a system, such as emergent behaviours, in the early design stages result in significant savings in the design costs. Scenarios are useful tools for describing the system behaviours and can be used to detect the possible emergent behaviours. Considering that each scenario describes a certain part of system behaviours, a set of scenarios is required to provide an appropriate model. Therefore, any faults such as incompleteness or conflict in the scenarios can cause fatal errors in the system performance.

This demonstrates the importance of designing an automatic method to deal with these defects. In this thesis, first, the process of converting the scenarios into the corresponding state machines based on the definitions given in [25] is performed. These definitions are then applied to the real-time fleet management system case study to validate their effectiveness.

As proposed in [25, 27], for each sequence diagram, Finite State Machines (FSMs) are built for any agent. Each state has two attributes: 1) The label of the state which is shown as the state index. 2) The state value that will be discussed in the following Definitions of this section. In Figures 5.5, 5.6, 5.7 and 5.8, the finite state machines are modeled for the agent "Server" of all the scenarios discussed in Section 5.2. For example, in Figure 5.5, q_0, q_i, q_f are the initial state, the i^{th} state and the final state of the agent "Server" in scenario S_1, respectively. For the rest of agents in these scenarios, state machines can be formed in a similar way.

In the next stage, a certain value is assigned to each state of the finite state machine of

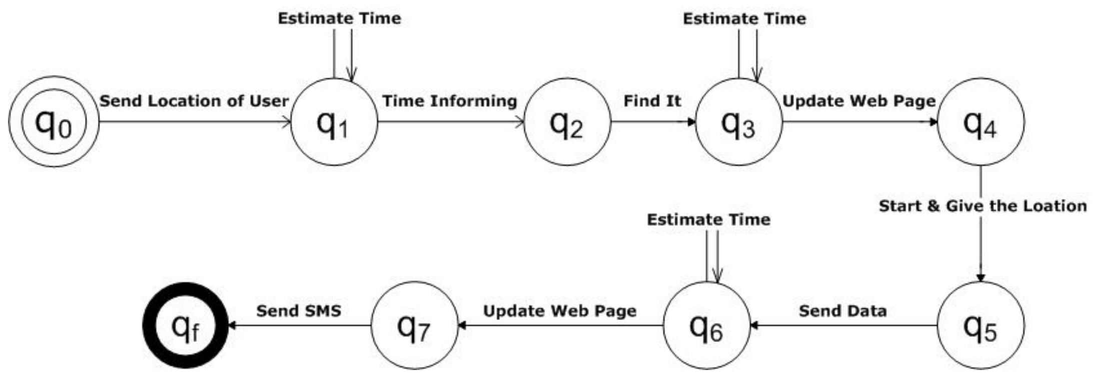

Figure 5.5: Finite state machine for agent "Server" of scenario S_1

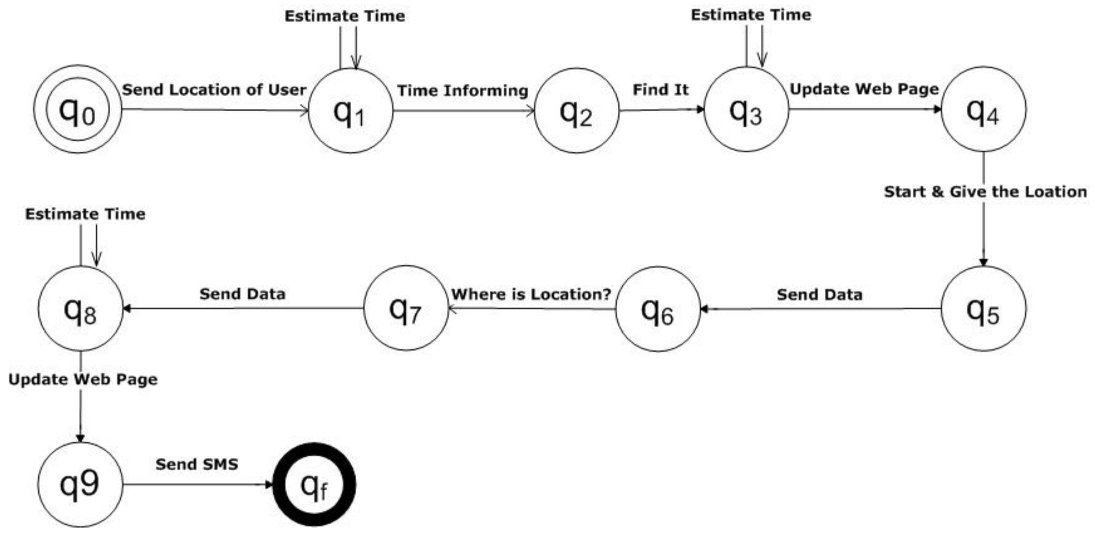

Figure 5.6: Finite state machine for agent "Server" of scenario S_2

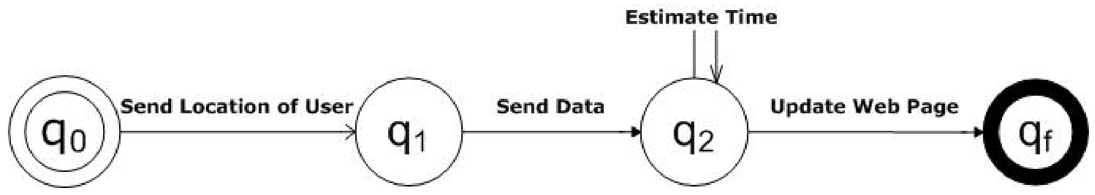

Figure 5.7: Finite state machine for agent "Server" of scenario S_3

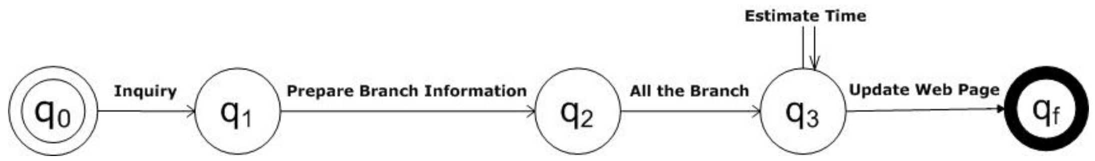

Figure 5.8: Finite state machine for agent "Server" of scenario S_4

each agent. This technique, originally proposed in [25], is performed using the definitions presented in the following paragraphs and is effective to quantify the states.

Definition 1: If agent i needs the result of message $m|_i[j]$ for doing $m|_i[k]$, then message $m|_i[j]$ is a semantical cause for message $m|_i[k]$ and is shown by $m|_i[j] \xrightarrow{se} m|_i[k]$, where $m|_i[j]$ represents the j^{th} message interacted by the agent i of scenario m.

The semantical causality is a feature of the system which does not change and is not affected by the decisions of the domain expert. As an example, in Figure 5.1, the message "Send Data" is a semantical cause for occurrence of the message "Estimate Time". Each state of an agent is defined by considering the messages that are semantical causes for the messages that come after that state. The semantical causality is resulted from the domain knowledge and indicates that the agent needs a message to construct another message. According to this definition, the domain theory is defined as:

Definition 2: For a set of message sequence diagrams M, the domain theory D_i for each agent i is defined as: if $m|_i[j] \xrightarrow{se} m|_i[k]$, then the pair $(m|_i[j], m|_i[k])$ is in the domain theory D_i.

According to Definition 2, ("Send Data" , "Estimate Time") from Figure 5.1 is in the domain theory. Having different behaviours results in producing various behaviour models for a single agent. However, choosing between these models is impossible as they cannot be compared. Therefore, this definition is used to define a method for quantifying the states. In [25], it is proposed to use the invariant characteristics of the system to find a unique way of calculating the state values. This technique is described in the following:

Definition 3: A finite state machine for agent i in scenario m is shown with tuple $A_i^m = \left(S^m, \Sigma^m, \sigma^m, q_0^m, q_f^m\right)$ where S^m is a finite set of states, Σ^m is the alphabet that is the set of all messages, σ^m is the transition relation, q_0^m is the initial state and q_f^m is the final state. In this finite state machine, for the final state q_f^m, the state value is calculated as $v_i|(q_f^m) = m|_i[f-1]$, and for $0 < k < f$, the state value is defined as follows:

1. $v_i|(q_k^m) = m|_i[k-1]v_i|(q_j^m)$, if there exist some j and l such that j is the maximum index that $m|_i[j-1] \xrightarrow{se} m|_i[l]$, $0 < j < k$, $k \leq l < f$.

2. $v_i|(q_k^m) = m|_i[k-1]$ if case 1 does not hold but $m|_i[k-1] \xrightarrow{se} m|_i[r]$, for some $k \leq r < f$.

3. $v_i|(q_k^m) = 1$, if none of the above cases holds.

The state value of q_k^m is related to message that comes before it. For case 1, the semantical cause is $m|_i[j-1]$, $0 < j < k$. For case 2, $m|_i[k-1]$ is the only semantical cause. Finally, there is not any semantical cause for case 3. According to Definition 3, the order of messages is utilized to achieve state values which is necessary to differentiate between the states in scenarios. The state values that are found using Definition 3 are effective for analysing the system behaviours. After constructing the finite state machines from various scenarios, in order to clearly analyse the system behaviours, the state machines for each agent are blended. Therefore, the concept of identical states is defined as:

Definition 4: For each agent i, two states q_j^m and q_k^n from two scenarios m and n (m can be equal to n) are identical states if:

- $j = k$ and for $0 \leq l < j$: $m|_i[l] = n|_i[l]$.

  or

- $v_i(q_j^m) = v_i(q_k^n)$.

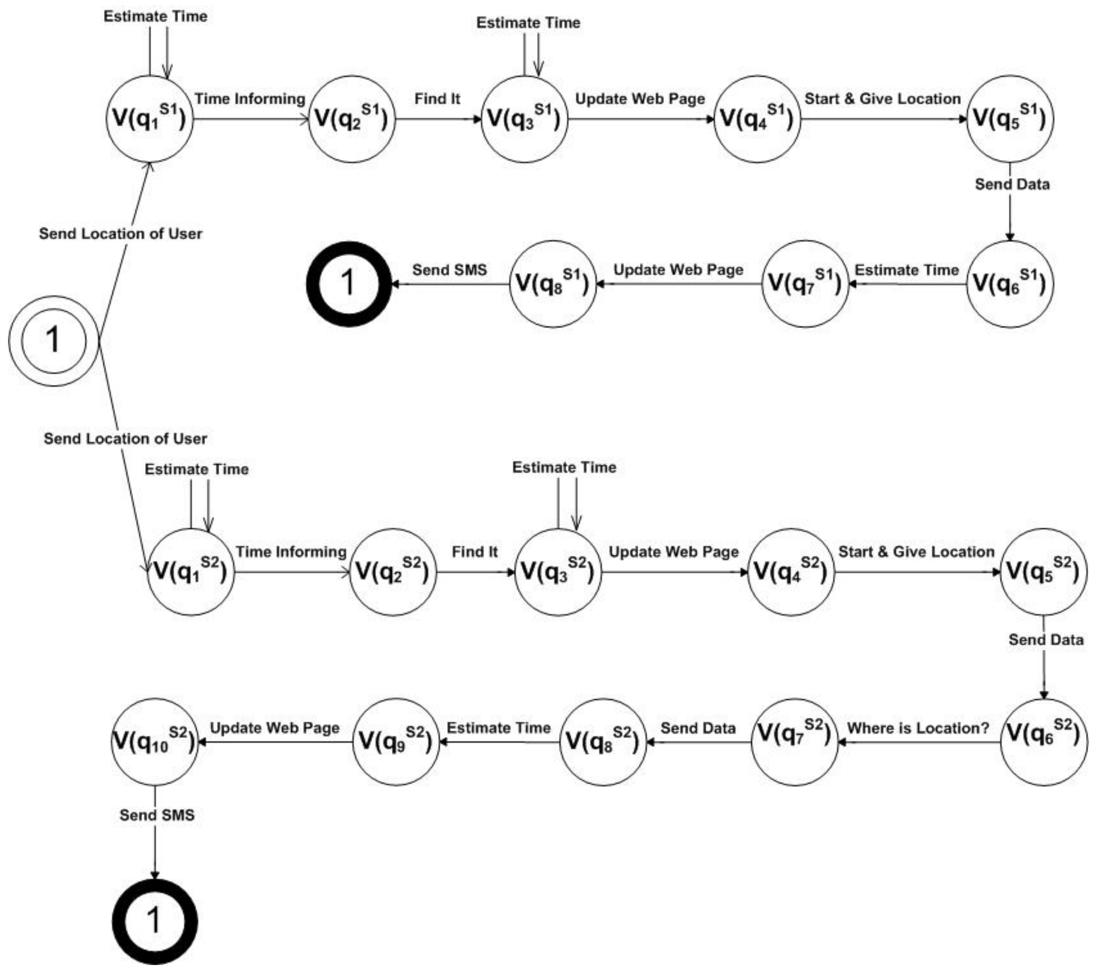

Figure 5.9: Blending finite state machines for agent "Server" of S_1 and S_2

In the example presented in Figure 5.9, the finite state machines of scenarios S_1 and S_2 are blended where the states are assigned with the appropriate state values. Note that the values of the initial and final states are supposed to be 1. To show the procedure of assigning state value, the value of identical state $q_6^{S_1,S_2}$ is calculated as $v_{\text{Server}}\left(q_6^{S_1,S_2}\right) = (\text{Send Data}) \times v_{\text{Server}}\left(q_5^{S_1,S_2}\right)$.

The presence of identical states in the behaviour models may result in emergent behaviour since the agent may be confused while performing the next message. Therefore, dealing with these issues is an important challenge in analysing the behaviour models.

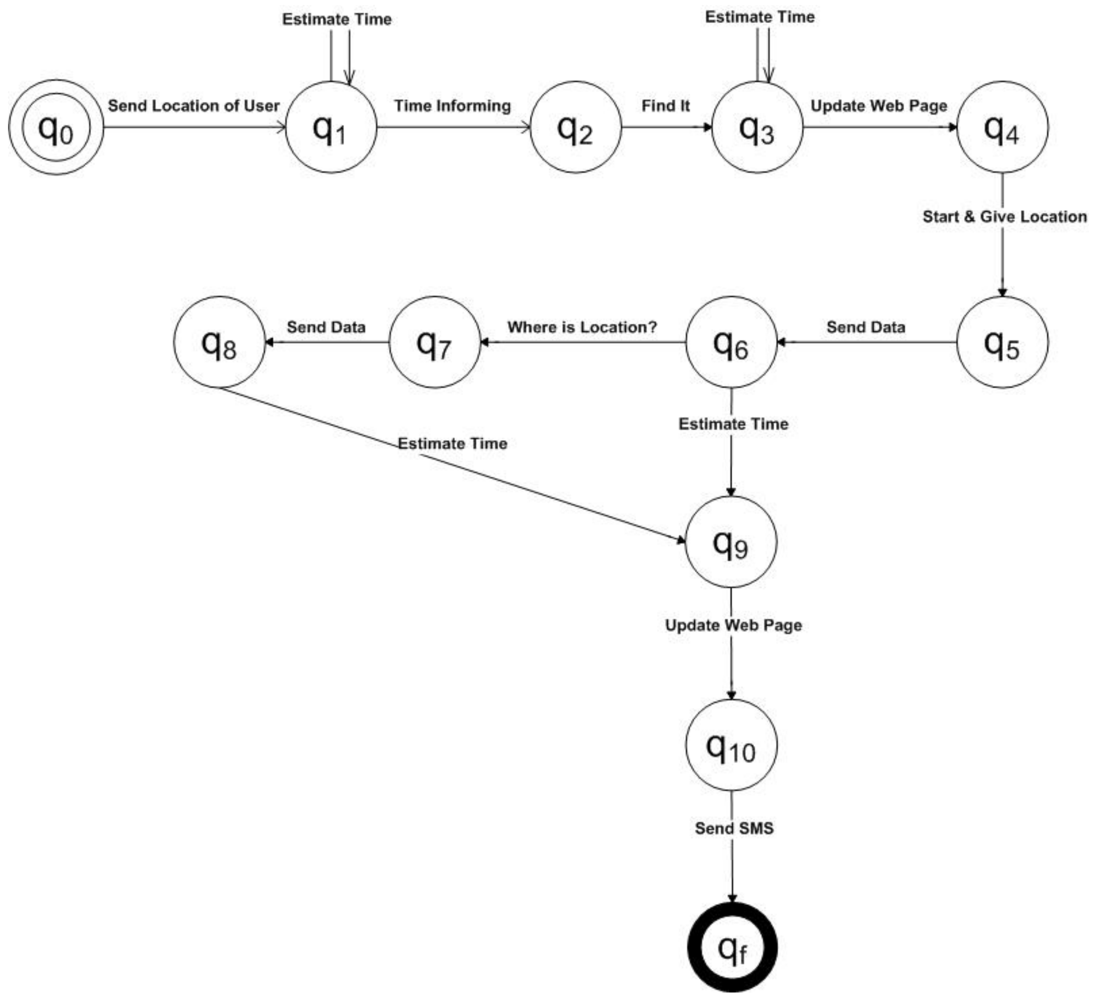

Figure 5.10: Merging the identical states of finite state machines for agent "Server" of scenarios S_1 and S_2

Once the identical states are found, the finite state machines are merged. This is done by merging the found identical states in the behaviour models. In Figure 5.9, emergent behaviour occurs for agent "Server" as a result of identical states $q_0^{S_1,S_2}$, $q_1^{S_1,S_2}$, $q_2^{S_1,S_2}$, $q_3^{S_1,S_2}$, $q_4^{S_1,S_2}$, $q_5^{S_1,S_2}$ and $q_6^{S_1,S_2}$ of scenarios S_1 and S_2. These identical states are then merged in order to remedy the occurrence of emergent behaviour as shown in Figure 5.10.

According to this figure, merging identical states can detect the emergent behaviours

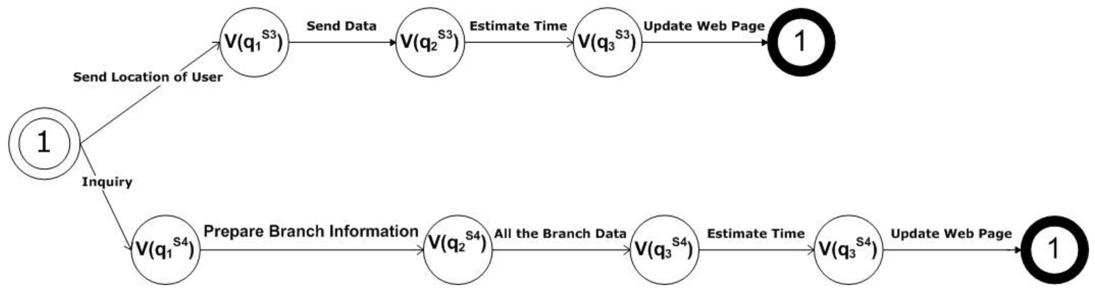

Figure 5.11: Blending finite state machines for agent "Server" of scenarios S_3 and S_4

in MAS designs. However, as will be discussed in the following section, not all the identical states cause emergent behaviours and merging all the identical states is not desired.

5.4 The Proposed Technique for Detecting Emergent Behaviour Caused by Overgeneralization

As discussed in the previous section, having identical states may lead to the occurrence of emergent behaviours. To address this issue, it is proposed in [25] to merge different state machines by merging their identical states. This prevents the emergent behaviours which happen due to the synthesis of behaviour models.

This method however requires more consideration since it merges all the identical states without considering whether they really produce any emergent behaviours or not. This is an important issue since merging the identical states that do not produce any emergent behaviours results in overgeneralization in the behaviour models. Furthermore, merging all the identical states takes too much time and memory which is unnecessary.

To better illustrate this issue, in Figure 5.11, an example is considered where the finite state machines for the scenarios S_3 and S_4 are blended. The states are then assigned by the values found based on Definition 3 and the identical states are found

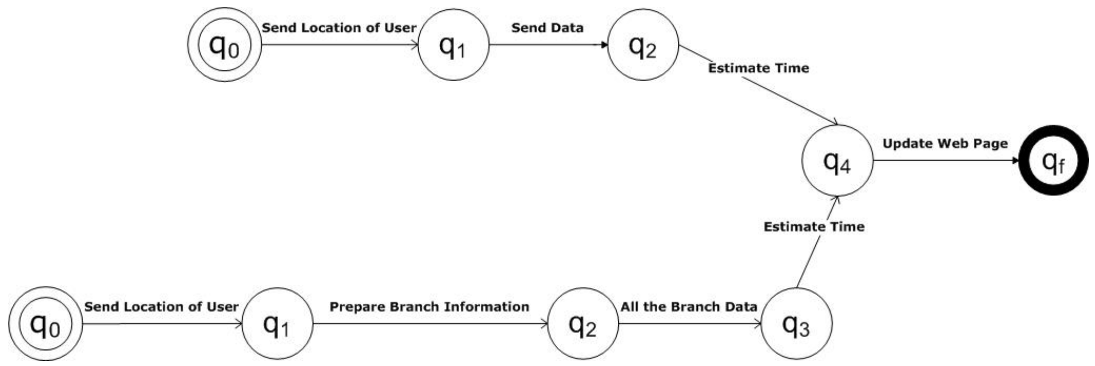

Figure 5.12: Merging identical states in finite state machines for agent "Server" of scenarios S_3 and S_4

based on Definition 4. To show the procedure of assigning state value, the values of states $q_3^{S_3}, q_4^{S_4}$ are calculated as $v_{\text{Server}}\left(q_3^{S_3}\right) = (\text{Estimate Time}) \times v_{\text{Server}}\left(q_1^{S_3}\right)$ and $v_{\text{Server}}\left(q_4^{S_4}\right) = (\text{Estimate Time}) \times v_{\text{Server}}\left(q_1^{S_4}\right)$. Based on these state values, in the finite state machines of scenarios S_3 and S_4, shown in Figure 5.11, states $q_3^{S_3}$ and $q_4^{S_4}$ are identical states which are supposed to result in emergent behaviour according to the technique proposed in [25]. However, when these states are identified and merged in Figure 5.12, it can be seen that these identical states do not lead to emergent behaviours since the agent "Server" never gets confused in performing the order of messages. Therefore, merging such identical states is not necessary.

Since most of the existing approaches merge all of the identical states, they result in over-generalized state machines. In this work, a set of criteria is presented to check the identical states and identify the ones that actually result in emergent behaviours. These criteria are defined in the following definition:

Definition 5: The agent i in scenario m has emergent behaviour in state q_j^m, if there exists a scenario n (m and n can be equal) and a state q_k^n, such that q_j^m and q_k^n are identical states and one of the following holds:

1. $m|_i[j] \neq n|_i[k] = i!l(c)$ where l is an agent and $i!l(c)$ represents a message with

content c sent from i to l.

2. $m|_i[j] \neq n|_i[k] = i?l(c)$ where l is an agent and $i?l(c)$ represents a message with content c received by i from l. Also, agent l sends a message with content c to agent i ($l!i(c)$) such that agent i does not receive this message before the event of $m|_i[j]$ in scenario m and by removing this event, agent l can still send $l!i(c)$.

3. State q_k^n is the final state of scenario n and $m|_i[j] = i!l(c)$ is a send message for agent i.

4. Case 2 holds except that by removing the event of $m|_i[j]$ in scenario m, agent l cannot send $l!i(c)$ any more. Then, there exists event e and w for agent l such that the event of $m|_i[j]$ is syntactical cause for $m|_l[e]$. Then, two states q_{e-1}^m and q_{w-1}^n are identical states and emergent behaviour for agent l has occurred.

Only the identical states that meet the criteria in Definition 5 result in emergent behaviours and should be merged. Therefore, it is unnecessary to merge the other identical states in the scenarios. These criteria can be used in an automated algorithm which performs synthesis of behaviour models while preventing the emergent behaviours which occur as a result of converting the scenarios to the state machines.

To prove the effectiveness of the above criteria, the finite state machines for agent "Server" in scenarios S_3 and S_4 are shown in Figure 5.11. In this example, after assigning the state values, the identical states are found to be $q_3^{S_3}$ and $q_4^{S_4}$. However, these states do not lead to emergent behaviour since none of the four cases in Definition 5 holds. The reason is that in all the cases of Definition 5, the events that come after the identical states should be interacting messages with different contents while in this example, the event, "Updating Web Page", is the same in both scenarios. Therefore, even though these states are identical, the agent "Server" is not confused about the next action to take. To better illustrate this issue, the identical states are merged and the resulting state machines are presented in Figure 5.12. It can be seen that merging the states does not prevent any emergent behaviour and is not necessary.

5.5 Summary

It has been reported in the literature that detecting unwanted behaviour during the design phase is about 20 times cheaper than finding them during the deployment phase [26,70]. However, many of the existing methodologies that are utilized to analyse system requirements and design documents introduce a certain amount of overhead to the software development lifecycle [42]. This chapter provides a systematic approach to analyse system requirements for defects, while saving on overhead by providing the opportunity to the designers for replacing ad-hoc methodologies with automated ones. In this work, after studying the shortcomings of the existing methods for detecting emergent behaviours, a new method is developed for behaviour model synthesis and emergent behaviour detection while preventing overgeneralization. The proposed method includes a set of criteria that enables the designer to detect and address the emergent behaviours and prevent unnecessary actions. Therefore, it is beneficial in reducing design costs and time.

Chapter 6

A System-Level Approach for Model-Based Verification of Multi-Agent Systems

6.1 Introduction

A major challenge in design of Multi-Agent Systems (MASs) is predicting and avoiding unexpected behaviours at the run time. Detecting those behaviours after the system is implemented can be very costly and detecting them during design and implementation stages is a cost effective alternative. Therefore, model-based verification at early design stages is an important step in designing MASs. Most of the existing verification techniques analyse system behaviour by going from specifications to state machines that model system behaviours. However, these methods are tailored to individual agents separately and do not consider the whole system simultaneously. Although those methods are shown to be effective in detecting unexpected behaviour for each agent, they fail to detect the unexpected behaviour that occurs due to the agent interactions. In this chapter, a system-level approach is proposed that produces a system-level behaviour model to analyse the system. To consider the interactions among system agents, a method is proposed to combine the behaviour models of interacting agents. A case study, real-time fleet management system, is presented to validate the efficiency of the proposed algorithm in detecting the implied scenarios (unexpected system behaviours) for multi-agent systems.

The rest of this chapter is organized as follows: In Section 6.2, the shortcomings of the component-level verification of multi-agent systems is presented. The proposed algorithm is explained in Section 6.3 for verifying MAS using a system-level approach.

In Section 6.4, a case study is considered to describe and validate the procedure of the proposed system-level verification algorithm. In Section 6.5, the proposed system-level verification algorithm is applied to the case study of real-time fleet management system. Finally, the chapter is summarized in Section 6.6.

6.2 Component-Level MAS Verification

Multi-agent systems are composed of system components, i.e.agents, and their interactions. In order to represent MAS specifications, scenarios that include agents and interaction messages among the agents are used. Message Sequence Charts (MSCs) and Sequence Diagrams (SDs) are popular tools for demonstrating scenarios of MASs [12].

Analysing the behaviours of MASs during design phase is performed to reduce the failures that may happen during deployment. This results in reduction of the overall costs. In order to detect unexpected behaviours in the design phase, two approaches for synthesis of behaviour models exist: component-level [25], and system-level approaches [27].

In the component-level approach, the behaviour of an individual agent is analysed by synthesizing its behaviour models from scenarios and creating state machines for each scenario in which that agent is involved [25]. In this approach, unexpected behaviours that may appear are usually called emergent behaviours. There are two methods to detect emergent behaviour. The first method compares behaviour models and scenario specifications and identifies emergent behaviours by merging identical states in different state machines of a single agent. [25]. The second method, proposed in Chapter 5, focuses on overgeneralization, which is a side effect of generalizing instance behaviour in scenarios. In Chapter 5, a component-level algorithm is proposed to address the emergent behaviours in the scenario specifications.

Although detecting and fixing the emergent behaviours of all the agents can potentially prevent major failures of the MASs, this procedure is incomplete since the behaviour resulted from interactions between the agents is not fully considered. There may be some scenarios as a result of these interactions that are not part of system specifications and cannot be detected using component-level approaches. In fact, component-level approaches can only deal with a portion of the unexpected system behaviours, i.e. emergent behaviours, since the behaviours of the whole system are not considered simultaneously. In order to handle this issue, system-level verification is studied. The current system-level verification techniques always require a designer to keep track of several variables and make final decisions [27]. Therefore, in this chapter, to verify system-level behaviours, a system-level algorithm is proposed to deal with the interactions among agents.

In this chapter, labeled transition system is used to combine behaviour models of all the agents in all the scenario specifications to produce a comprehensive behaviour model of the whole system. This comprehensive behaviour model will be validated against system's properties and will provide feedback for correcting the system specifications, i.e. scenarios. This will address the shortcomings of the component-level techniques that focus on behaviour analysis of individual agents. In this chapter, an algorithm is proposed to detect unexpected system behaviours. The proposed algorithm is then validated using a case study of real-time fleet management system. This algorithm can also be used in an automated framework to replace the existing ad-hoc system-level verification techniques.

6.3 A System-Level Verification Algorithm

In order to analyse if the implementations of a MAS are completely the same as the expected system behaviours, the concept of safe realizability is defined [27]. In system-level verification, the unexpected behaviours of the system after implementation are called implied scenarios that should be detected in design stage. To show that a system is safe realizable, the non-deterministic behaviours of the agents which result in implied scenarios should be caught. In the following, an algorithm is proposed to show whether the system is safe realizable or not. This proposed algorithm, Algorithm SV, that consists of four levels is presented in Figure 6.1. This algorithm addresses the shortcomings of the existing component-level techniques in detecting implied scenarios that happen due to the dynamic inter-agent interactions.

```
Algorithm SV: System-level Verification
Inputs: Scenarios, $m \in M$
Output: Implied scenarios
1. Constructing concurrent automata for all agents in all scenarios $m \in M$
2. Obtaining LTS$_i$ for agent $i$ by connecting all the concurrent automata $A_i$
3. Connecting LTSs of pairs of interacting agents to obtain system-level
   behaviour model
4. Determine the safe realizability of the system
```

Figure 6.1: High-level algorithm for system-level verification

The inputs to Algorithm SV are the scenarios describing the behaviour of MAS. The algorithm outputs are the implied scenarios that are not expected behaviours and should be considered to correct the system specifications. In Step 1 of the proposed Algorithm SV, in order to model the behaviours of the system agents, state machines are created for all agents in each scenario $m \in M$. Each state machine represents the behaviour of one agent in a single scenario. Therefore, all these state machines should be executed simultaneously to provide a realistic behaviour model. Then, a concurrent automaton is defined as below:

Concurrent automata: Considering asynchronous message passing among agents, a behaviour model for agent i can be described by an automaton A_i over the alphabet Σ_i with the following elements:

1. A set Q_i of states

2. A transition relation $\delta_i \subseteq Q_i \times \Sigma_i \rightarrow Q_i$, which includes all states $q_j \in Q_i$ that are accessible from states $q_k \in Q_i$ with messages $\sigma_k \in \Sigma_i$

3. An initial state $q_0 \in Q_i$

4. A set $F_i \subseteq Q_i$ of accepting states

where alphabet Σ_i is the order of messages for agent i.

After producing state machines for all agents in all the scenarios $m \in M$, automata A_i for a specific agent i in all the scenarios should be connected. This way, a path, i.e. an order of states and messages, is obtained. By connecting the state machines corresponding to several scenarios $m \in M$, an acceptable path is obtained which is a path of states and messages that ends in a defined state or can be an infinite path. To overcome the shortcoming of state machines in dealing with infinite paths, Labeled Transition Systems (LTSs) [31] are developed which can deal with both finite and infinite paths.

Labeled Transition System: Over the alphabet Σ_i, a Labeled Transition System for agent i is determined as:

1. A set Q_i of states

2. A transition relation $\delta_i \subseteq Q_i \times \Sigma_i \rightarrow Q_i$

3. An initial state $q_0 \in Q_i$

4. A set $F_i \subseteq Q_i$ of accepting states which can be empty

To obtain an LTS$_i$ for agent i, in Step 2 of Algorithm SV, all the state machines of automaton A_i are connected so that the acceptable state of A_i for scenario m is connected to the initial state of the next scenario n with ϵ-transitions. These ϵ-transitions are state transitions with no content that are used to connect different state machines. The ϵ-transitions indicate which states of the second state machine are potentially accessible from the final state of the first one. Then, ϵ-transitions will be eliminated and the LTS$_i$ for agent i is completely formed.

Considering that the verification cost and time are limited, it is not efficient to consider all the paths in the LTS. Therefore, from all these paths, only paths with shortest length, i.e. paths without any repetitive sequence of states, are considered because they have all the information needed to detect implied scenarios. These are called basic paths [27]. Basic paths are defined as follows:

Basic paths of LTS: Considering $p = q_0 w_0 q_1 w_1 \ldots w_k q_k$ as a path of LTS$_i$ for agent i where w_j is a word in alphabet Σ_i that represents an order of messages. Then, p is a basic path if it ends in an acceptable state and does not go repeatedly over any loop or the following conditions hold:

1. p does not go repeatedly over any loop
2. For all accessible states q_{k+1} from q_k, we have $q_{k+1} \in \{q_0, q_1, \ldots, q_k\}$
3. $q_{k-1} w_k q_k$ is not repeated in p
4. p is not a part of another path in LTS$_i$ for agent i

By removing all the states from a path, an execution is achieved. Therefore, the execution of path $p = q_0 w_0 q_1 w_1 ... w_k q_k$ is shown as $w_0 w_1 ... w_k$. Each LTS may contain one or more paths and several executions. A path that is derived from another path by repeating an order of states and messages is called a derived path and its execution is named a derived execution.

Step 3 of Algorithm SV is modeling the interactions between the agents, i.e. obtaining a system-level behaviour model. To model these relations and interactions, the agents that interact with the others by sending or receiving at least one message should be found. In order to combine LTS$_i$ and LTS$_j$ of the two agents i, message sender, and j, message receiver, all the paths in their LTSs are considered. Again, repeated and derived paths can be ignored to decrease verification runtime as they are covered by the basic paths. Then, each basic path of LTS$_i$ of agent i that sends a message to agent j should be connected with a τ-transition to all the basic paths of LTS$_j$. τ-transitions are semantically equivalent to the ϵ-transitions but are used to connect two LTSs. Finally, τ-transitions are removed to finalize the system-level behaviour model.

In Step 4 of Algorithm SV, to determine the safe realizability of the MAS, the results of system implementation are checked to find the implied scenarios. Therefore, all expected paths of the system are considered as the system language that is shown by L. In other words, system language includes all the expected system behaviours. On the other hand, all the paths that are achieved by connecting LTSs of the interacting agents represent the system implementation. If the system implementation and the system language L are exactly the same, the MAS is a safe realizable system. Otherwise, system has implied scenarios that need to be addressed. The paths that are in system implementation but not in the L are the culprits.

In the next section, to illustrate the procedure of the proposed algorithm and its effectiveness, an MAS case study is presented for experimentation.

6.4 Case Study Scenarios

In order to show differences between the existing component-level approaches and the proposed system-level algorithm for analysing system behaviours, a case study called Real-Time Fleet Management System, devised in Chapter 4, is used. For real-time fleet management system, there exist several different scenarios to show how agents interact

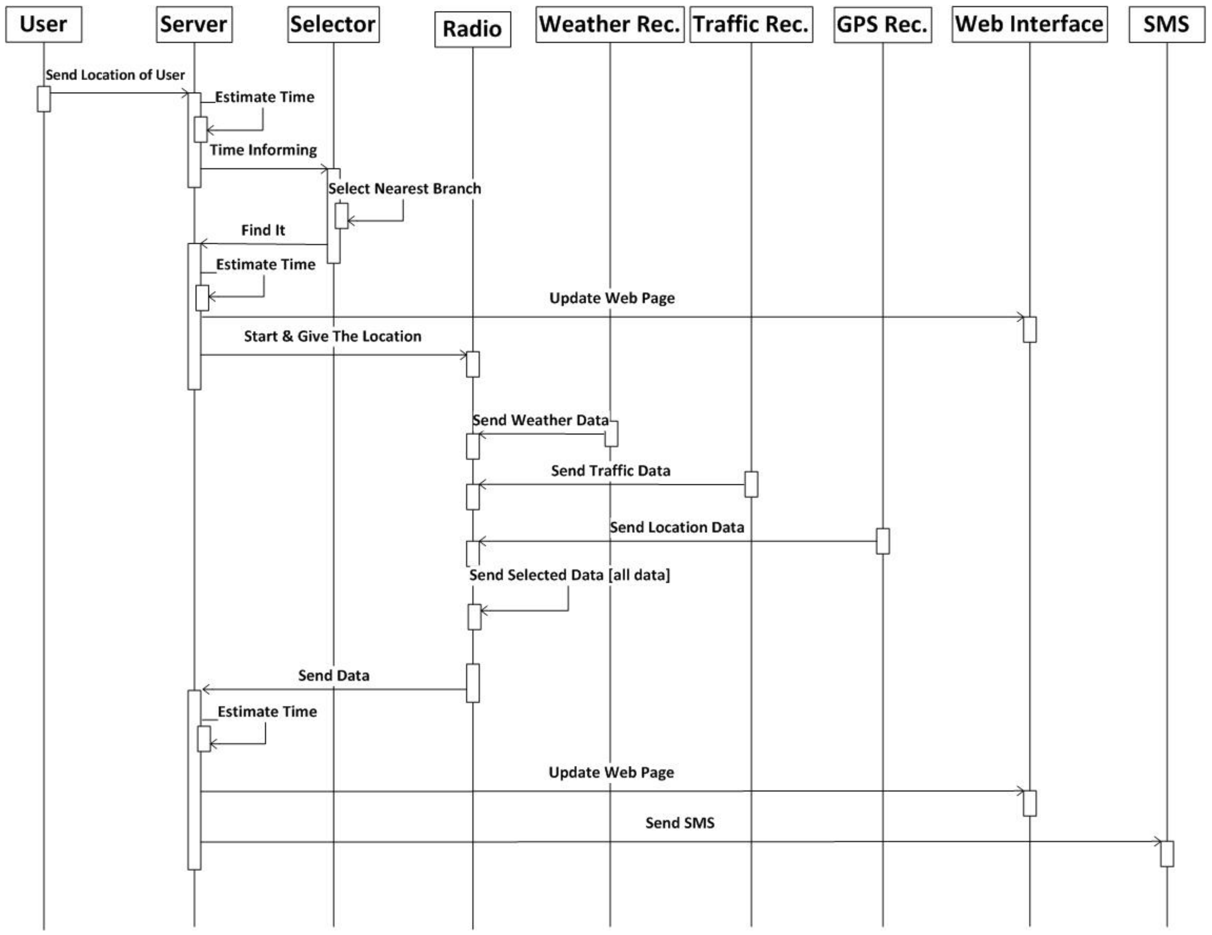

Figure 6.2: Scenario S_1 describing partial behaviours of real-time fleet management system case study

with each other in order to obtain system goals. Three of these scenarios are considered in this chapter for validating the proposed system-level verification algorithm. This system is a multi-agent system since the system agents are autonomous and interactive which take responsibility for their actions. Agents interact to obtain the system goal which is minimizing delivery time and updating the customers.

In Figure 6.2, the first scenario, S_1, is considered where agent "Server" receives the location of customer from agent "User" and asks agent "Selector" to find the nearest branch based on customer location. Then, agent "Radio" selects the data, i.e. weather,

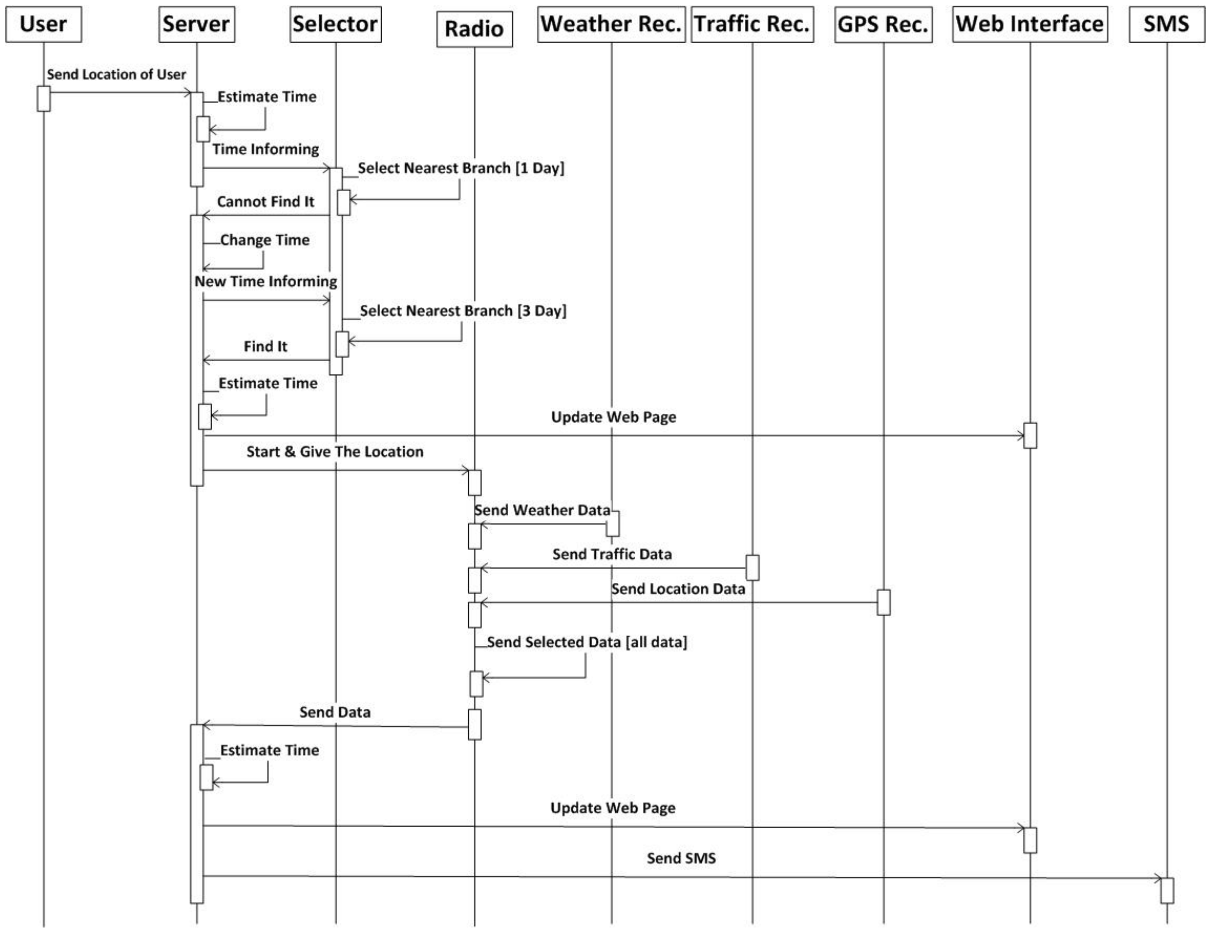

Figure 6.3: Scenario S_2 describing partial behaviours of real-time fleet management system case study

traffic and location data, that influence delivery time and sends them to "Server". After estimating the exact delivery time, the customer is informed by updating web page and sending text message.

The second scenario, S_2, is presented in Figure 6.3. This scenario considers a special case where agent "Selector" cannot find the nearest branch in the given delivery time. To deal with this issue, agent "Server" should change the period of expected delivery time in order to let the agent "Selector" to find an available branch. Therefore, as it is seen in Figure 6.3, once the agent "Selector" is unable to find a branch in the given delivery time, agent "Server" finds the closest time period and asks agent "Selector" to find an available branch in the new time period.

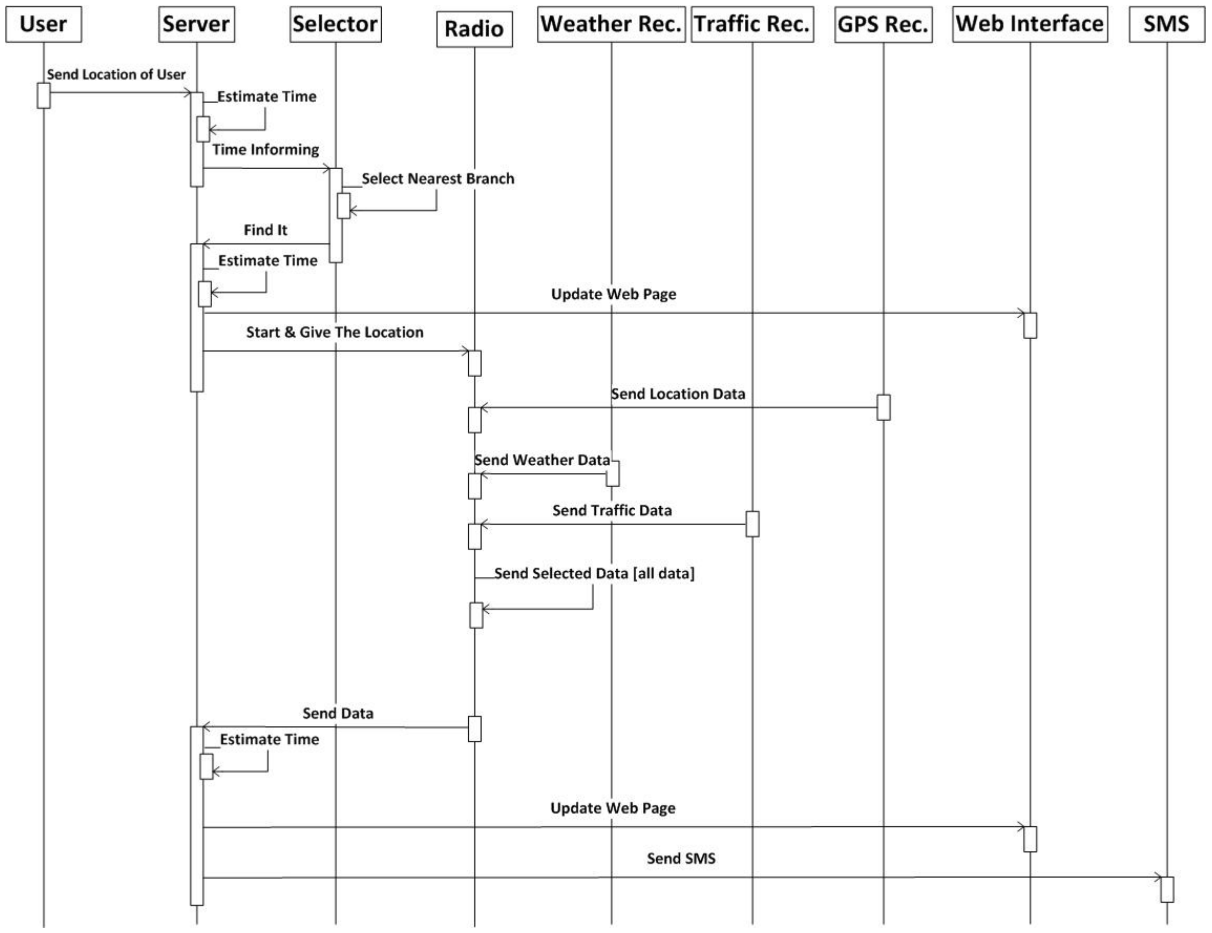

Figure 6.4: Scenario S_3 describing partial behaviours of real-time fleet management system case study

In Figure 6.4, the third scenario, S_3, which describes partial behaviours of the real-time fleet management system, is given. This scenario includes the same behaviours as scenario S_1 but the difference is in the order of messages. "Radio" agent collects all the data influencing the delivery time from "GPS Rec.", "Weather Rec." and "Traffic Rec." agents. However, the difference is in the order of receiving data from these agents which is different from the one in scenario S_1 shown in Figure 6.2.

These three scenarios describe the partial behaviours of the case study. These will be used to demonstrate the effectiveness of the proposed system-level verification algorithm in the following section in order to detect the implied scenarios of the system that are considered as unwanted system behaviours.

6.5 Applying the Proposed System-Level Verification Algorithm to the Case Study

In order to synthesize the behaviour models of the whole system, a system-level algorithm, Algorithm SV, is proposed in Section 6.3 which considers the agents' interactions comprehensively. Following Algorithm SV, in Step 1, for all system agents, concurrent automata are produced. Then, following Step 2, all the concurrent automata for each

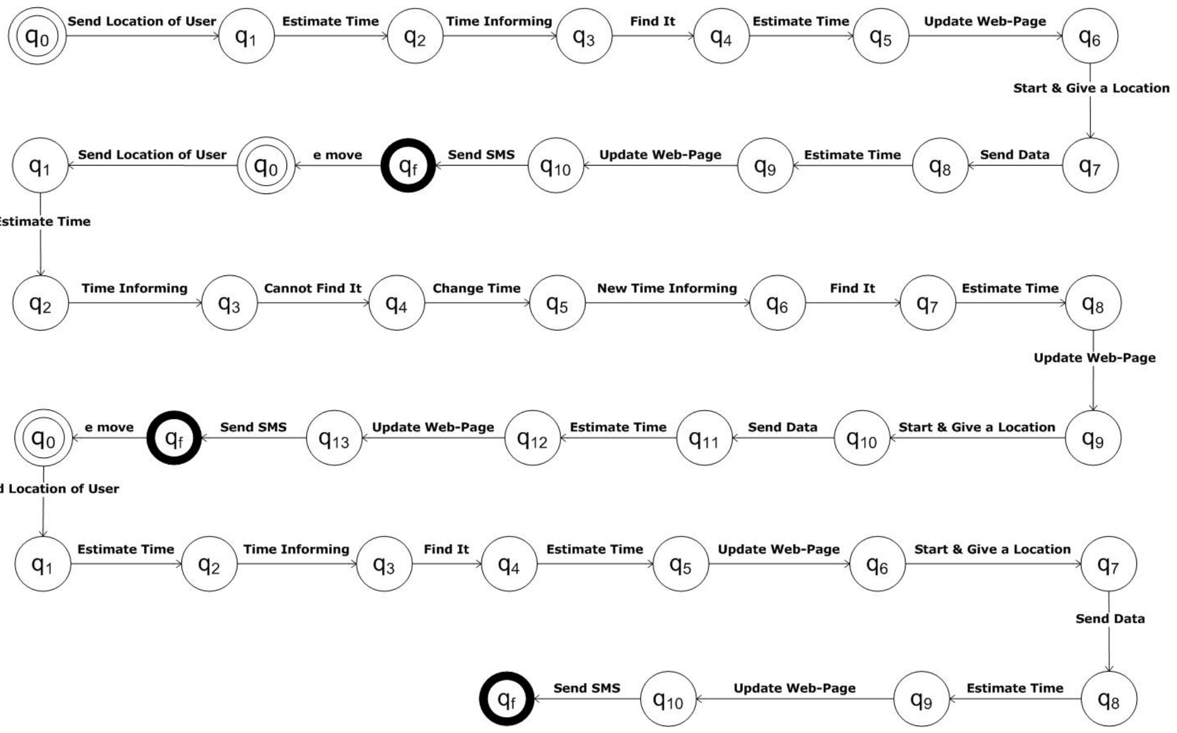

Figure 6.5: LTS for agent "Server"

agent are connected using ϵ-transitions (shown by e move in Figures 6.5 and 6.6) to obtain a Labeled Transition System (LTS). The LTSs for two major agents "Server" and "Radio" are shown in Figures 6.5 and 6.6. LTSs for the rest of the agents are developed similarly.

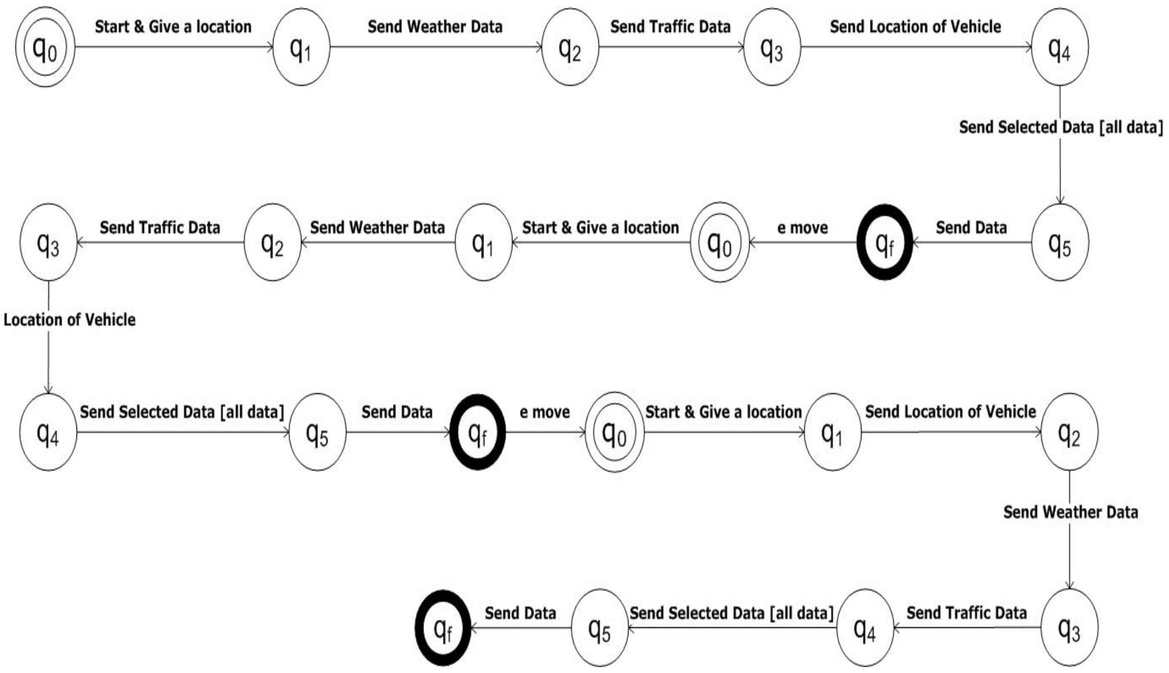

Figure 6.6: LTS for agent "Radio"

After creating LTSs for all the agents, basic paths are obtained. Then, in Figures 6.7 and 6.8, the refined LTSs for agents "Server" and "Radio" are presented by removing ϵ-transitions, repeated states and their messages which gives the basic paths. Since these basic paths cover all the other paths, they are enough to detect implied scenarios. This is done by assigning state values and merging identical states. To show the procedure of assigning state values, the value of identical state q_3 for agent "Server", which is equal in all the scenarios, is calculated as $v_{\text{Server}}(q_3) = (\text{Time Informing}) \times v_{\text{Server}}(q_1)$. In addition, the value of identical state q_1 for agent "Radio", which is equal in all the scenarios, is calculated as $v_{\text{Radio}}(q_1) = (\text{Start \& Give a location}) \times 1$.

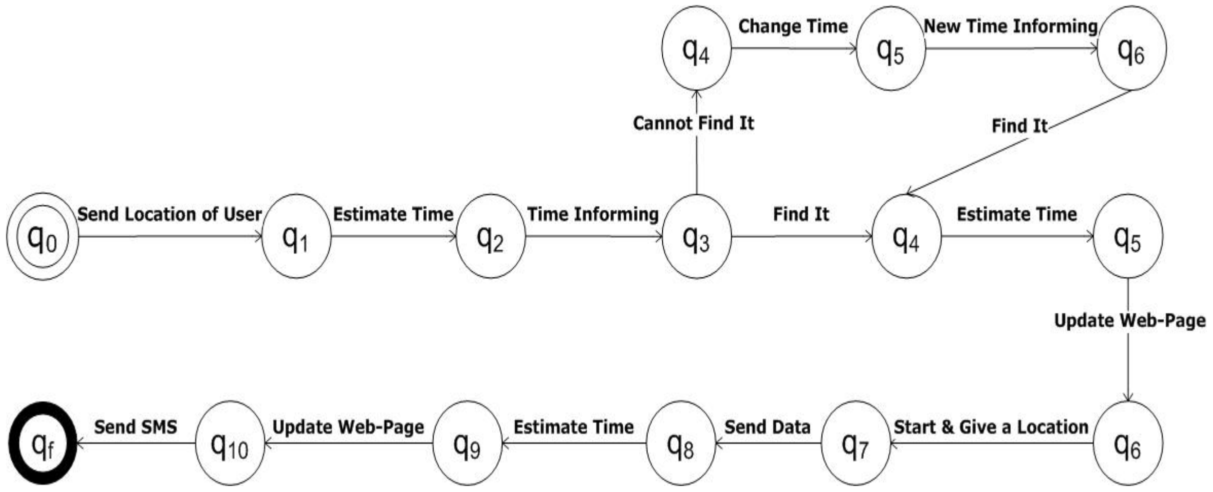

Figure 6.7: Final LTS for "Server"

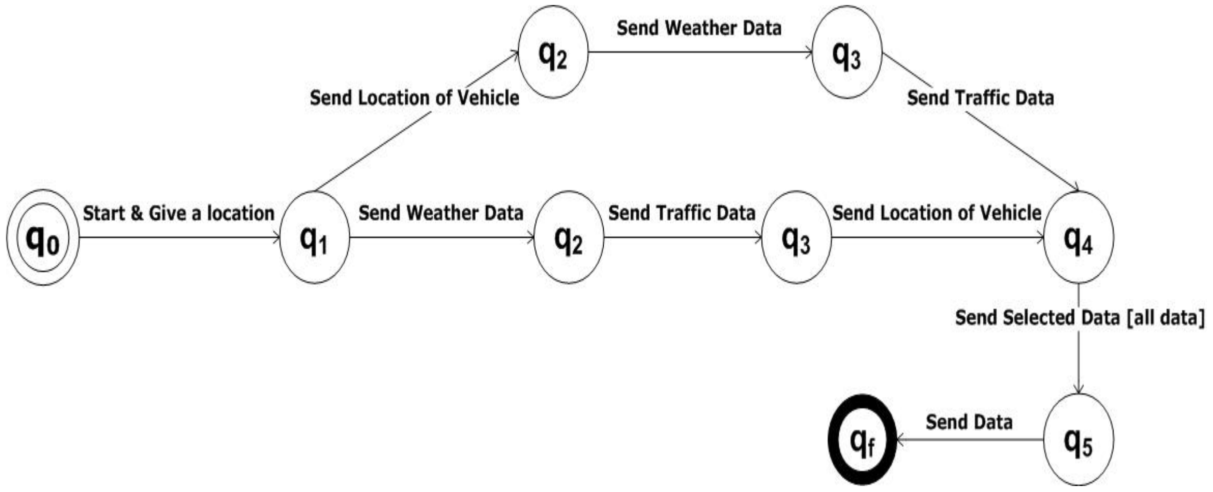

Figure 6.8: Final LTS for "Radio"

As shown in Figures 6.7and 6.8, there are two basic paths for each LTS. In Step 3 of the proposed Algorithm SV, to achieve system-level behaviour model, these basic paths of LTSs of interacting agents are connected by τ-transitions. Since "Server" agent sends a message with the content of "Start and Give a location" to "Radio" agent, the basic paths of "Server" agent will be connected to the basic paths of "Radio" agent using τ-transitions (shown by T move in Figures 6.9, 6.10, 6.11 and 6.12). Since in this example, there are two basic paths for each agent, four paths are generated by connecting the basic paths. These connected paths are shown in Figures 6.9, 6.10, 6.11 and 6.12.

By removing the τ-transitions from the connected paths shown in Figures 6.9, 6.10, 6.11 and 6.12, four paths are obtained. These paths represent the system-level behaviour model of the real-time fleet management system. To find out if the system is safe realizable, these connected paths should be compared with the expected paths. The connected paths shown in Figures 6.9, 6.10 and 6.11 are exactly the same as the paths that are expected by the scenario specifications described in Section 6.4.

The first path shown in Figure 6.9 is a part of scenario S_1. The second path shown in Figure 6.10 represents a part of scenario S_2. Furthermore, the third path that is presented in Figure 6.11 is a part of scenario S_3. However, the fourth path presented in Figure 6.12 is not part of any of the three scenarios specifications describing the expected behaviours of the case study. Therefore, this path is resulted from an implied scenario of the system since it is not expected to occur by the scenario specifications.

For better illustration, the corresponding scenario for this implied scenario is presented in Figure 6.13. As it can be seen in Figure 6.13, for "Server" agent, the order of messages is the same as scenario S_2 but the order of the messages for "Radio" agent is the same as scenario S_3. It should be noted that although the order of messages passed between other agents is the same as the expected scenarios, the interactions between agents "Server" and "Radio" results in an unexpected scenario. Detecting this implied

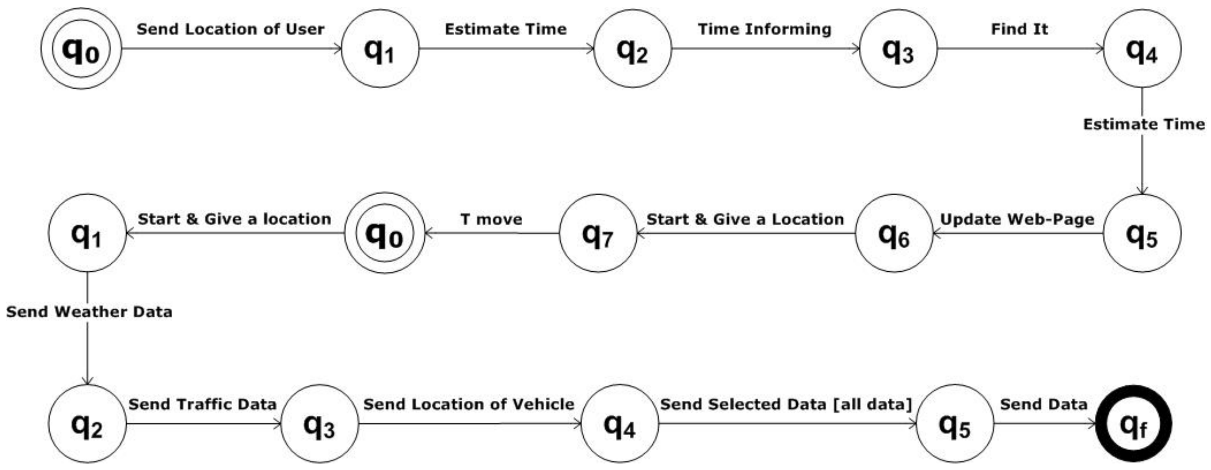

Figure 6.9: First connected path resulting from connecting LTSs of "Server" and "Radio" agents

scenario indicates that the system is not safe realizable.

The existing component-level verification techniques model the system behaviour for a single component, i.e. agent, at a time. Therefore, they are not able to detect this implied scenario. This is because in this implied scenario, if the agent's behaviours are considered individually (i.e. component-level verification), this implied scenario will be verified to be expected behaviour by the scenario specifications. However, the interactions between these agents produce the behaviours that are not expected by the system. These unexpected behaviours can only be detected using a system-level approach such as the one proposed in this thesis. Therefore, system-level verification is a major step in the design of multi-agent systems.

The proposed algorithm is implemented automatically since there is no need for designer intervention in any of its proposed steps. Thus, this method can be implemented as a fully automated tool that gets scenarios specifications of a multi-agent system as the inputs and verifies the system for implied scenarios.

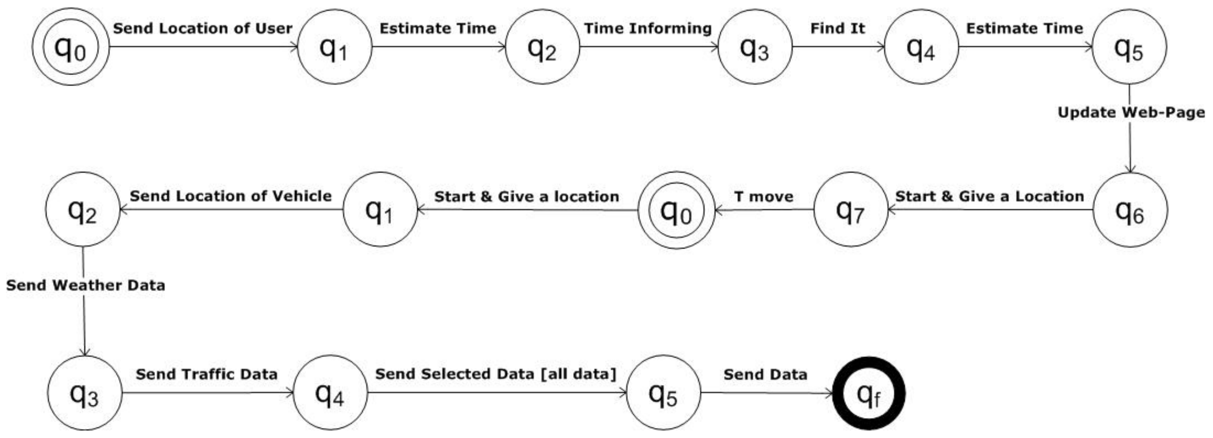

Figure 6.10: Second connected path resulting from connecting LTSs of "Server" and "Radio" agents

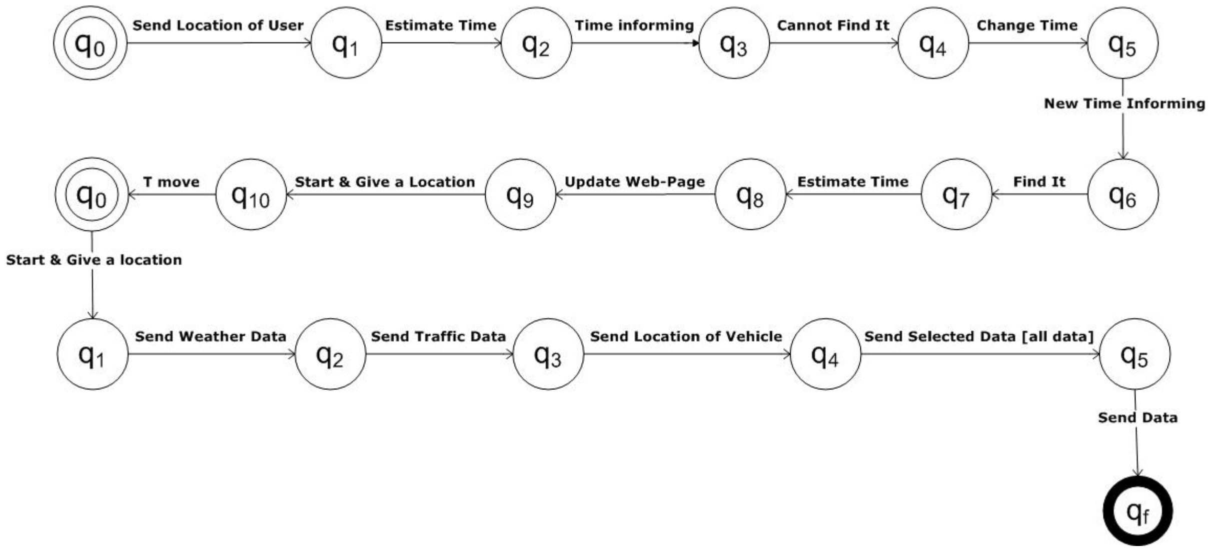

Figure 6.11: Third connected path resulting from connecting LTSs of "Server" and "Radio" agents

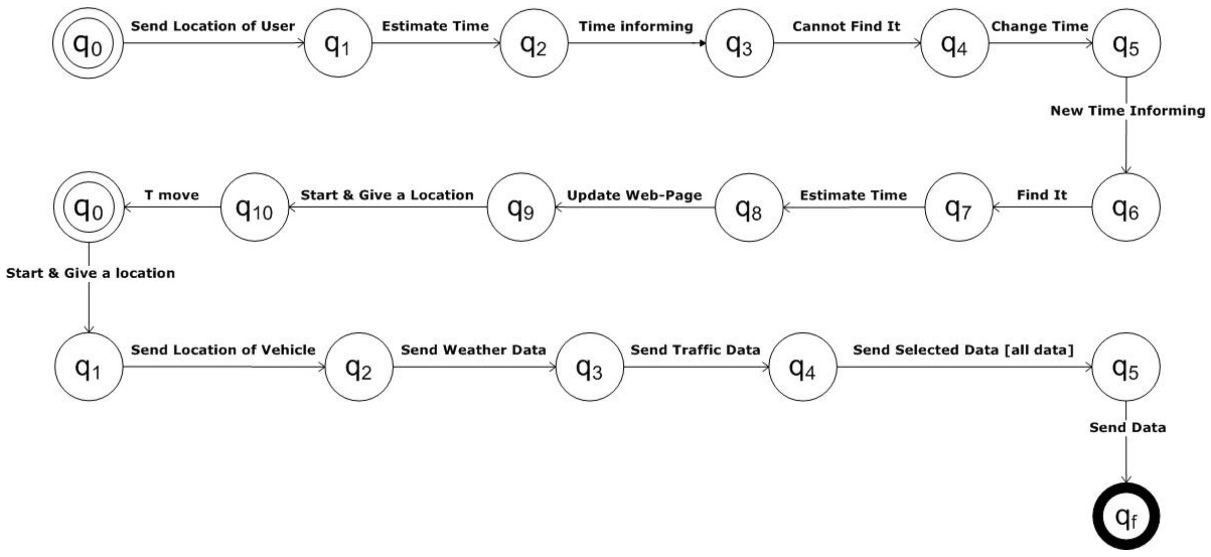

Figure 6.12: Fourth connected path resulting from connecting LTSs of "Server" and "Radio" agents

6.6 Summary

Verification of multi-agent systems in early design stages has become more important since it can be several times cheaper than finding the unexpected system behaviours in the deployment phase [26, 70].

In this chapter, a system-level verification algorithm is proposed that evaluates the input system requirements and determines if the system is safe realizable, i.e. there is no unwanted system behaviour that may occur in the implementation stage of the multi-agent system. By performing system-level verification, the unexpected behaviours that occur due to the interactions between system agents are detected and addressed. This algorithm can be used in an automated syntax checker to verify the system requirements automatically and replace the existing ad-hoc methodologies that always need the designer to make major decisions. This algorithm is validated by presenting a case study of a real-time fleet management system.

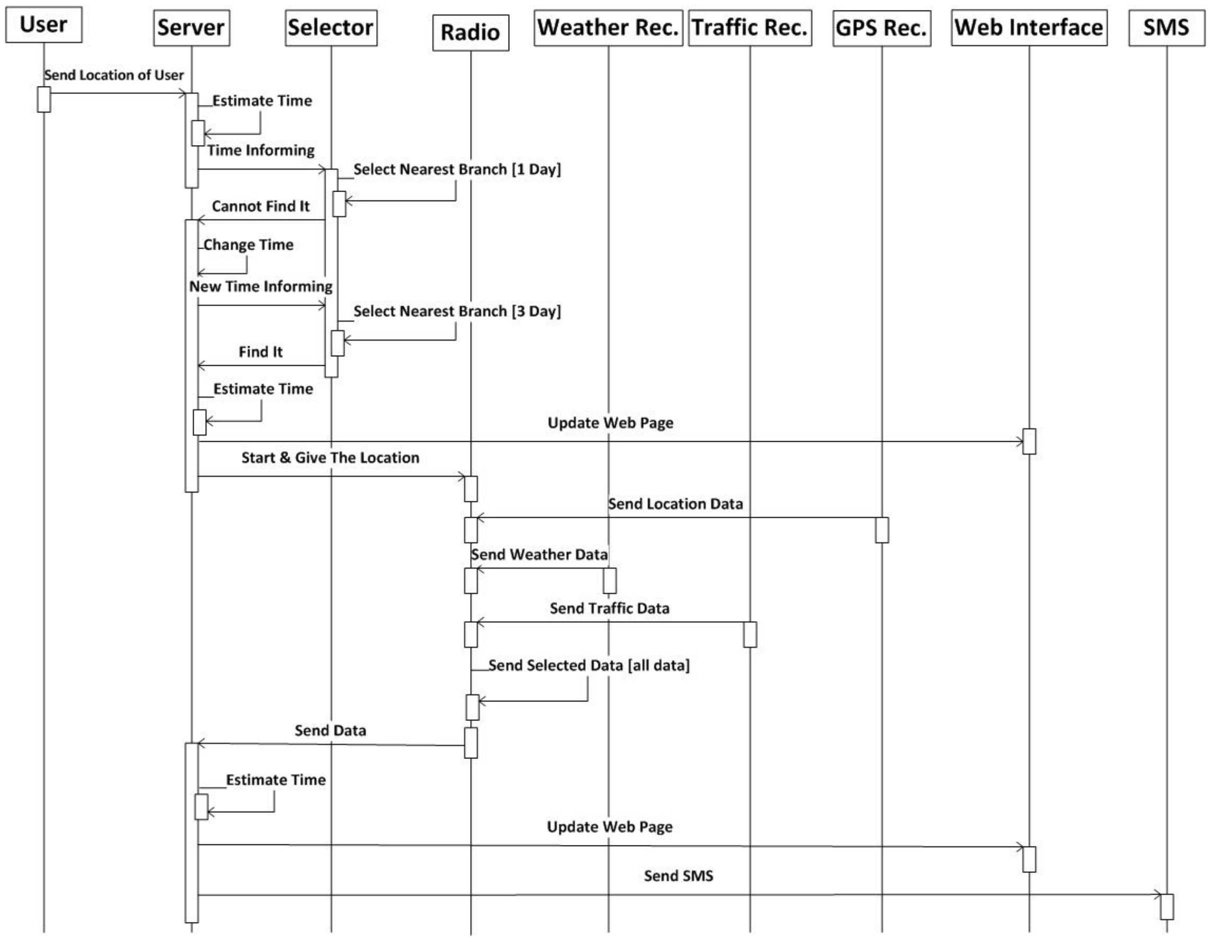

Figure 6.13: The detected implied scenario, S_4

Chapter 7

Conclusion and Future Work

7.1 Summary and Contributions

Recently, due to the growth of demands for multi-agent systems (MASs), several design methodologies are proposed. Detecting unexpected behaviours of the MASs in early design stages is a vital task. Unified Modeling Language (UML) is one of the methodologies proposed for object-oriented software design. Since UML is developed to define the object characteristics, it cannot support agents and interactions among them. Therefore, in this thesis, to comprehensively represent the agent interactions, an extension of UML called AUML is employed for MAS design. Considering that UML sequence diagrams are accepted representation tools for software system verification, in this thesis, a set of conversion rules is proposed to provide UML sequence diagrams from AUML notations. Therefore, the UML diagrams that are converted using the proposed conversion rules thoroughly represent the interactions among agents of MASs. In order to show the effectiveness of the proposed method, an industrial case study of a real-time fleet management system is presented.

Verification of MASs in early design stages to detect unexpected software system behaviours is significantly cheaper than verifying them during implementation stages [26, 70]. Although there exist several techniques for synthesis of behaviour models, most of them are ad-hoc methods that need the designers to make final decisions [42]. In this thesis, in order to analyse the behaviours of multi-agent systems automatically, a systematic component-level approach is proposed to catch emergent behaviours. This method synthesizes a behaviour model for each agent of the system and detects emergent behaviours. This method also prevents overgeneralization by presenting a set of criteria to catch unexpected behaviours while preventing unnecessary actions.

Some unexpected behaviours of the system, i.e. implied scenarios, may occur as a result of interactions among agents in MASs. These unexpected behaviours are neglected in component-level verification since it considers the behaviours of one single agent at a time. Then, in this thesis, a system-level algorithm is proposed for providing a comprehensive MAS verification technique. The proposed system-level verification algorithm, Algorithm SV, which is validated by presenting a case study of real-time fleet management system, can be used to verify the requirements of system automatically using labeled transitions systems.

In the following, the main contributions of this thesis are presented:

- Designing of multi-agent systems using AUML.
- Proposing a set of conversion rules to produce UML sequence diagrams from AUML notations.
- Preventing overgeneralization by developing a component-level approach.
- Applying labeled transition systems for synthesizing system-level behaviour models.
- Proposing a system-level algorithm for verification of multi-agent systems.
- Validating the proposed methods by presenting a case study of real-time fleet management system.

7.2 Future Work

The proposed method verifies MAS designs by converting AUML notations to UML sequence diagrams based on the proposed conversion rules. The future work may be proposal of a MAS verification method that performs verification on AUML sequence diagrams directly. In addition, in this thesis, a new method is developed for synthesis of behaviour models to detect emergent behaviours while preventing overgeneralization. The future work can be combining the proposed algorithm with a syntax checker to provide a comprehensive automated tool. Such automated tool can be employed to perform both verification and testing for multi-agent systems.

The proposed system-level algorithm is generic and can handle systems whose specifications are defined by AUML. There is a need to tailor the algorithm to the artifacts produced based on certain design methodologies for MASs such as MaSE. Therefore, the future work is performing system-level verification for the MASs that are designed using other methodologies. Another future direction of this work is implementing all the proposed methods in a unique tool to perform a complete procedure of MAS design and verification.

Bibliography

[1] S. DeLoach. The MaSE methodology. In *F. Bergenti, M. Gleizes, and F. Zambonelli, Methodologies and Software Eng. for Agent System*, pages 107–147. Boston: Kluwer Academic Publishers, 2004.

[2] Unified Modeling Language Specification. Technical report, Rational Software Corporation, 2006.

[3] A. Mousavi and B. Far. Revisiting safe realizability of message sequence charts specifications. In *Proc. of International Conference on Engineering of Complex Computer Systems*, pages 37–45, 2008.

[4] N. Mani, V. Garousi, and B. Far. A UML-based conversion tool for monitoring and testing multi-agent systems. In *Proc. of International Conference on Tools with Artificial Intelligence*, volume 01, pages 212–219, 2008.

[5] M. Moshirpour, A. Mousavi, and B. Far. Detecting emergent behavior in distributed systems using scenario-based specifications. *International Journal of Software Engineering and Knowledge Engineering*, 22(06):729–746, 2012.

[6] B. Bauer, J. Müller, and J. Odell. Agent UML: A formalism for specifying multiagent interaction. In *P. Ciancarini, M. Wooldridge, Agent-Oriented Software Engineering*, pages 91–103. Springer, 2001.

[7] Encom Wireless. `http://www.encomwireless.com`.

[8] The City of Calgary. `http://www.calgary.ca/`.

[9] J. Carroll. Five reasons for scenario-based design. In *Proc. of Hawaii International Conference on Systems Sciences*, volume 3, page 3051, 1999.

[10] ITU: Message Sequence Charts. Recommendation. Technical report, International Telecommunication Union, 1992.

[11] L. Lu and D. Kim. Required behavior of sequence diagrams: Semantics and refinement. In *Proc. of International Conference on Engineering of Complex Computer Systems*, pages 127–136, 2011.

[12] T. S. S. O. ITU. Series Z: Languages and general software aspects for telecommunication systems - formal description techniques (FDT) message sequence chart. Technical report, International Telecommunication Union, 1999. 136 pp.

[13] G. Andrews. *Foundations of Multithreaded, Parallel, and Distributed Programming.* Addison Wesley, 2000.

[14] M. Wooldridge. *An Introduction to Multi-Agent Systems.* John Wiley and Sons, 2009.

[15] J. Odell, H. Parunak, and B. Bauer. Extending UML for agents. In *Proc. of Agent-Oriented Information Systems*, pages 3–17, 2000.

[16] R. Miles and K. Hamilton. *Learning UML 2.0.* O'Reilly Media, 2006.

[17] T. Quatrani. *Visual modeling with Rational Rose 2000 and UML (2nd ed.).* Addison-Wesley Longman Ltd., 2000.

[18] B. Bauer, J. Müller, and J. Odell. An extension of UML by protocols for multiagent interaction. In *Proc. of International Conference on MultiAgent Systems*, pages 207–214, 2000.

[19] M. Huget. Model checking Agent UML protocol diagrams. Technical report, Department of Computer Science, University of Liverpool, 2002.

[20] J. Odell, P. Van, and B. Bauer. Representing agent interaction protocols in UML. In *Proc. of Agent-Oriented Software Engineering*, pages 121–140, 2001.

[21] M. Huget. Agent UML notation for multiagent system design. *IEEE Internet Computing*, 8(4):63–71, 2004.

[22] M. Huget and J. Odell. Representing agent interaction protocols with Agent UML. In *Proc.of International Joint Conference on Autonomous Agents and Multiagent Systems*, pages 1244–1245, 2004.

[23] B. Bauer, J. Müller, and J. Odell. Agent UML: a formalism for specifying multiagent software systems. In *Proc. of Agent-Oriented Software Engineering*, pages 91–103, 2001.

[24] A. Mousavi, B. Far, A. Eberlein, and B. Heidari. Strong safe realizability of message sequence chart specifications. In *Proc. of International Conference on Fundamentals of Software Engineering*, pages 334–349, 2007.

[25] M. Moshirpour, A. Mousavi, and B. Far. Detecting emergent behavior in distributed systems using scenario-based specifications. In *Proc. of International Conference on Software Engineering and Knowledge Engineering*, pages 349–354, 2010.

[26] R. Goldsmith. *Discovering Real Business Requirements for Software Project Success*. Norwood MA: Artech House, Inc., 2004.

[27] A. Mousavi. *Inference of Emergent Behaviours of Scenario-Based Specifications*. PhD thesis, University of Calgary, Calgary, Alberta, 2009.

[28] J. Chakraborty, D. D'Souza, and K. Kumar. Analysing message sequence graph specifications. In *Proc. of International Conference on Leveraging Applications of Formal Methods, Verification, and Validation*, pages 549–563, 2010.

[29] H. Muccini. Detecting implied scenarios analyzing non-local branching choices. In *Proc. of International Conference on Fundamental Approaches to Software Engineering*, pages 372–386, 2003.

[30] R. Alur, K. Etessami, and M. Yannakakis. Realizability and verification of MSC graphs. In *Proc. of International Colloquium on Automata, Languages and Programming*, pages 797–808, 2001.

[31] F. Sousa, N. Mendonca, S. Uchitel, and J. Kramer. Detecting implied scenarios from execution traces. In *Proc. of Working Conference on Reverse Engineering*, pages 50–59, 2007.

[32] R. Bordini, M. Fisher, C. Pardavila, and M. Wooldridge. Model checking agentspeak. In *Proc. of International Joint Conference on Autonomous Agents and Multiagent Systems*, pages 409–416, 2003.

[33] C. Wang, Y. Yang, A. Gupta, and G. Gopalakrishnan. Dynamic model checking with property driven pruning to detect race conditions. In *Proc. of International Symposium on Automated Technology for Verification and Analysis*, pages 126–140, 2008.

[34] I. Song, S. Jeon, A. Han, and D. Bae. An approach to identifying causes of implied scenarios using unenforceable orders. *Information and Software Technology*, 53:666–681, 2011.

[35] R. Bordini, M. Fisher, W. Visser, and M. Wooldridge. Model checking rational agents. *IEEE Intelligent Systems*, 19(5):46–52, 2004.

[36] G. Holzmann. The model checker SPIN. *IEEE Trans. on Software Engineering*, 23: 279–295, 1997.

[37] R. Bordini, M. Fisher, W. Visser, and M. Wooldridge. Verifying multi-agent programs by model checking. *Autonomous Agents and Multi-Agent Systems*, 12:239–256, 2006.

[38] P. Gluck and G. Holzmann. Using SPIN model checking for flight software verification. In *Proc. of IEEE Aerospace Conference*, volume 1, pages 105–113, 2002.

[39] S. Chaki and J. Ivers. Software model checking without source code. *Innovations in Systems and Software Engineering*, 6:233–242, 2010.

[40] M. Wood and S. DeLoach. An overview of the multiagent systems engineering methodology. In *Proc. of Agent-Oriented Software Engineering*, pages 207–221, 2001.

[41] N. Mani, V. Garousi, and B. Far. Monitoring multi-agent systems for deadlock detection based on UML models. In *Proc. of Canadian Conference on Electrical and Computer Engineering*, pages 001611–001616, 2008.

[42] M. Moshirpour, R. Alhajj, M. Moussavi, and B. Far. Detecting emergent behavior in distributed systems using an ontology based methodology. In *Proc. of International Conference on Systems, Man, and Cybernetics*, pages 2407–2412, 2011.

[43] R. Alur and M. Yannakakis. Model checking of message sequence charts. In *Proc. of International Conference on Concurrency Theory*, pages 114–129, 1999.

[44] R. Alur and M. Yannakakis. Model checking of hierarchical state machines. *ACM Trans. on Programming Languages and Systems*, 23:273–303, 2001.

[45] R. Alur. Formal verification of hybrid systems. In *Proc. of International Conference on Embedded Software*, pages 273–278, 2011.

[46] H. Ben-Abdallah and S. Leue. Syntactic detection of process divergence and non-local choice in message sequence charts. In *Proc. of International Workshop on Tools and Algorithms for Construction and Analysis of Systems*, pages 259–274, 1997.

[47] H. Ben-Abdallah and S. Leue. MESA: Support for scenario-based design of concurrent systems, 1997.

[48] J. Whittle and J. Schumann. Generating statechart designs from scenarios. In *Proc. of International Conference on Software Engineering*, pages 314–323, 2000.

[49] J. Whittle, R. Kwan, and J. Saboo. From scenarios to code: an air traffic control case study. *Software and Systems Modeling*, pages 71–93, 2005.

[50] M. Rayner, B. Hockey, N. Chatzichrisafis, and K. Farrell. OMG unified modeling language specification. In *Version 1.3, Object Management Group, Inc*, 2005.

[51] S. Uchitel, J. Kramer, and J. Magee. Negative scenarios for implied scenario elicitation. *SIGSOFT Software Engineering Notes*, 27:109–118, 2002.

[52] S. Uchitel, J. Kramer, and J. Magee. Synthesis of behavioral models from scenarios. *IEEE Trans. on Software Engineering*, 29:99–115, 2003.

[53] E. Letier, J. Kramer, J. Magee, and S.Uchitel. Monitoring and control in scenario-based requirements analysis. In *Proc. of International Conference on Software Engineering*, pages 382–391, 2005.

[54] S. Uchitel. Partial behaviour modelling: Foundations for incremental and iterative model-based software engineering. In *M. Oliveira and J. Woodcock, Formal Methods: Foundations and Applications*, pages 17–22. Springer, 2009.

[55] P. Pelliccione, H. Muccini, A. Bucchiarone, and F. Facchini. TESTOR: deriving test

sequences from model-based specifications. In *Proc. of International Conference on Component-Based Software Engineering*, pages 267–282, 2005.

[56] A. Bertolino, H. Muccini, and A. Polini. Architectural verification of black-box component-based systems. In *Proc. of International Conference on Rapid Integration of Software Engineering Techniques*, pages 98–113, 2007.

[57] J. Henriksen, M. Mukund, K. Kumar, and P. Thiagarajan. Regular collections of message sequence charts. In *Proc. of International Symposium on Mathematical Foundations of Computer Science*, pages 405–414, 2000.

[58] M. Mukund, K. Kumar, and M. Sohoni. Synthesizing distributed finite-state systems from MSCs. In *Proc. of International Conference on Concurrency Theory*, pages 521–535, 2000.

[59] R. Alur, T. Dang, J. Esposito, Y. Hur, F. Ivancic, K. Kumar, I. Lee, P. Mishra, G. Pappas, and O. Sokolsky. Hierarchical modeling and analysis of embedded systems, 2003.

[60] B. Adsul, M. Mukund, K. Kumar, and V. Narayanan. Causal closure for MSC languages. In *Proc. of Foundations of Software Technology and Theoretical Computer Science*, pages 335–347, 2005.

[61] P. Bhateja, P. Gastin, M. Mukund, and K. Kumar. Local testing of message sequence charts is difficult. In *Proc. of International Conference on Fundamentals of Computation Theory*, pages 76–87, 2007.

[62] F. Mokhati, N. Boudiaf, L. Badri, and M. Badri. Generating Maude formal specifications from AUML diagrams. *Journal of Computational Methods in Science and Engineering*, 6:73–89, 2006.

[63] F. Mokhati, M. Badri, L. Badri, F. Hamidane, and S. Bouazdia. Automated testing sequences generation from AUML diagrams: a formal verification of agents' interaction protocols. *International Journal of Agent-Oriented Software Engineering*, 2: 422–448, 2008.

[64] F. Mokhati, B. Sahraoui, S. Bouzaher, and M. Kimour. A tool for specifying and validating agents' interaction protocols: From Agent UML to Maude. *Journal of Object Technology*, 9:59–77, 2010.

[65] A. Mousavi and B. Far. Eliciting scenarios from scenarios. In *Proc. of International Conference on Software Engineering and Knowledge Engineering*, pages 466–471, 2008.

[66] M. Moshirpour, A. Mousavi, and B. Far. A technique and a tool to detect emergent behavior of distributed systems using scenario-based specifications. In *Proc. of International Conference on Tools with Artificial Intelligence*, pages 153–159, 2010.

[67] M. Moshirpour, S. El-Sherif, B. Far, and R. Alhajj. Detecting emergent behavior in a social network of agents. In *Influence of Technology on Social Network Analysis and Mining*, volume 6, pages 393–409. 2013.

[68] D. Harel and H. Kugler. Synthesizing state-based object systems from LSC specifications. Technical report, 1999.

[69] I. Kruger, R. Grosu, P. Cholz, and M. Broy. From MSCs to statecharts. In *Proc. of International Workshop on Distributed and Parallel Embedded Systems*, pages 61–71, 1999.

[70] Software Engineering Institute. `http://www.sei.cmu.edu/`.