UNIVERSITY OF CALGARY

Model-Based Detection of Emergent Behavior in Distributed and Multi-Agent Systems from a Component Level Perspective

by

Mohammad Moshirpour

A THESIS

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES

IN PARTIAL FULFILMENT OF THE REQUIREMENTS FOR THE

DEGREE OF MASTER OF SCIENCE

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

CALGARY, ALBERTA

JANUARY, 2011

UNIVERSITY OF CALGARY
FACULTY OF GRADUATE STUDIES

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies for acceptance, a thesis entitled "Model-Based Detection of Emergent Behavior In Distributed and Multi-Agent Systems from a component level perspective" submitted by Mohammad Moshirpour in partial fulfilment of the requirements of the degree of Master of Science.

Supervisor, Dr. Behrouz H. Far
Department of Electrical and Computer Engineering

Dr. Diwakar Krishnamurthy
Department of Electrical & Computer Engineering

Dr. Armin Eberlein
Department of Electrical and Computer Engineering

Dr. Reda Alhajj
Department of Computer Science

Jan 23, 2011
Date

Abstract

Requirement elicitation is one of the most challenging and critical phases of the software development lifecycle. Many faults are introduced into the system as the result of incomplete requirements. An effective approach for the design of software systems is to describe system requirements using scenarios. A scenario, commonly expressed using a message sequence chart or a sequence diagram, is a temporal sequence of messages sent between system components. However despite their simplicity and expressive power, scenario-based specifications are prone to subtle deficiencies with respect to analysis and validation known as incompleteness and partial description. These deficiencies in scenario-based specifications are the prime cause of emergent behavior. Emergent behavior, also known as implied scenarios are behavior that the system exhibits but are not explicitly defined in its requirements.

Emergent behavior is an important issue in the design of software systems; particularly ones with the lack of central control such as distributed and multi-agent systems. Detecting and removing emergent behavior during the design phase will lead to huge savings in deployment costs of such systems. In this thesis, a method for detecting emergent behavior in system requirements described using scenario-based specifications is proposed. The use of this methodology for a variety of different software systems such as distributed and multi-agent systems (MAS) is demonstrated. Furthermore this research contains methodologies for verifying the lack of existence of a particular emergent behavior in the software system. These methodologies have been demonstrated using various case studies such as distributed systems for a mine-sweeping robot and an online commerce application and a multi-agent system for a manufacturing system.

Furthermore as this research aims to develop these methodologies into a software tool, the requirement and design documents as well as the prototype of this tool are presented in this thesis.

Acknowledgements

I would like to express my deepest appreciation and gratitude to my supervisor Dr. Behrouz H. Far for his support and guidance throughout this work. I attribute much of the success of this work to his technical knowledge, patience in teaching me the ways of research and his tendency to provide me with the freedom to be creative.

I am grateful to Dr. Reda Alhajj, Dr. Diwakar Krishnamurthy and Dr. Armin Eberlein, my thesis committee members for reviewing this work and providing helpful guidance and feedback.

Finally I would like to express my most sincere gratitude to my parents, Hossein and Mahnaz and my sisters Mojgan and Mahtab for their unconditional support, unlimited kindness and constant encouragements. I am confident that without their help and support this would not have been possible. I would also like to extend my appreciation to Mahtab for her assistance in editing this work.

To My Grandmother who would have wanted to see this

Table of Contents

List of Figures

List of Tables

List of Abbreviations

AOSE - Agent Oriented Software Engineering

CEBD - Component-level Emergent Behavior Detection

DFA - Deterministic Finite Automaton

FSM - Finite State Machine

eFSM - Equivalent Finite State Machine

GUI - Graphical User Interface

ITU - International Telecommunications Union

MAS - Multi-agent Systems

MaSE - Multi-agent Software Engineering (It is one of the AOSE methodologies)

MDTM - Model-based Detection and Testing of MAS

MSC - Message Sequence Chart

MVC - Model-View-Controller

NFA - Non-deterministic Finite Automaton

OMG - Object Management Group

pMSC - Partial Message Sequence Chart

SBSE - Scenario-based Software Engineering

SD - Sequence Diagrams

SEBD - System-level Emergent Behavior Detection

UML - Unified Modeling Language

Chapter One: Introduction

1.1 Motivation

Gathering system requirements is one of the most challenging and at the same time a critical stage of the software development lifecycle. Due to the abstract nature of software, deciding on clear and concise goals and features can be a challenging task. Furthermore as software projects tend to involve several stakeholders, communicating ideas and conveying information can be an extremely difficult task [1].

An effective and efficient way to describe system requirements is using scenario-based specifications. A scenario is a temporal sequence of messages sent between system components and the actors. Scenarios are appealing because they allow stakeholders to describe system functionality by partial stories [2]. Since scenarios usually serve as abstract execution traces of the system, they provide the perfect medium through which customers, system developers and engineers and other stakeholders can communicate. Scenario-Based Software Engineering (SBSE) investigates ways in which scenarios can be used in software development [2]. Scenarios are particularly useful in describing the requirements for systems with the distribution of control such as distributed systems and multi-agent systems (MAS). The lack of central control in these systems often implies complex interactions among multiple components [2]. Therefore scenarios can be utilized to define these interactions. By following this approach the overall behavior of the system can be defined by a comprehensive set of scenarios.

However despite the advantages of using scenarios due to their expressive power and simplicity, there are several challenges particularly for concurrent systems consisting of multiple autonomous agents (MAS) as well as distributed systems which consist of multiple system components. For instance, because each scenario only gives a local and partial story of a distributed system's behavior, the challenge is how the behavior of a system can be constructed from a set of scenarios and more importantly whether the derived behavior is acceptable or not. Generally, system requirements described using scenarios are prone to several defects as follows [3]:

1. Scenarios are partial stories of the system's behavior and each scenario is only an instance of the system's functionality. Therefore defining comprehensive system requirements using scenarios raises issues of coverage and completeness.
2. Scenarios are instances of system behavior and thus they need to be properly combined to have a full description of the system.

Therefore the artefacts produced when defining the system using SBSE, the scenario, must be analyzed and verified. Unfortunately manual review of these documents is inefficient and time-consuming. In order to resolve these issues, devising systematic and automated methodologies to validate system requirements is necessary.

1.2 Objectives

The main objectives of this thesis are as follows:

- Devising systematic and automated methodologies to analyze system requirements (which are defined using scenario-based specifications) and identify cases of emergent behavior.
- Demonstrating the concept of indeterminism, formalizing the cause of implied scenarios as well as addressing and resolving the problems associated with it which are mentioned in [4].
- Applying the proposed methodologies in this thesis to distributed systems and successfully detecting emergent behavior in the requirements of such systems. Furthermore ensuring the lack of existence of certain illegal scenarios from the requirements of distributed system.
- Establishing a link between agent oriented software engineering methodologies (AOSE) such that AOSE design artefacts can be converted to scenario-based specifications. The results of this conversion are then used to verify the design documents of multi-agent systems.
- To design and implement the prototype of an easy to use and practical software tool to apply these methodologies to requirements and design documents of a variety of software systems such as distributed and multi-agent systems.

1.3 Approach and Methodologies

In this research the merits of defining system requirements using scenarios are acknowledged and the necessity of analysis and verification of scenario-based specifications is recognized. There are a number of methodologies proposed in the literature which deal with the analysis of scenario-based specifications [2, 5-9]. There are general commonalities among all approaches; however each approach makes a unique contribution in the analysis of scenarios. The general approach used for analyzing the requirements of software systems is done in two steps of *behavior modeling* and *detection of emergent behavior.* These steps are described in the following subsections.

1.3.1 Behavior Modeling

The model which describes the behavior of each system element (i.e. agent, component or processes) is called the *behavioral model,* and the procedure for building the behavioral models for the elements of scenarios, is called *synthesis of behavioral models,* or simply, the *synthesis process.* A widely accepted model for behavioral modeling of individual system elements is the state machine. Several studies have already been conducted to facilitate the procedure of converting a set of scenarios to a behavioral model expressed by state machines [2, 5-8, 10]. In the synthesis process, one state machine will be built for each system element. The state machine includes all of the messages that are received or sent by that element. Then the behavior of the distributed system is described by the product (parallel execution) of all the state machines of the system elements.

1.3.2 Emergent Behavior

One of the challenges during the synthesis process, is *implied scenarios* [3, 9, 11-13], also known as *emergent behavior* [3]. An implied scenario is a specification of behavior that is in the synthesized model of the software system and is not explicitly specified in its specification as a scenario.

Emergent behavior occurs when there exists a state, in which the system component becomes confused as to what course of action to take. This happens when identical states exist in the union of state machines obtained through behavioral modelling. A definition for identical states is needed for detection of emergent behavior. To achieve this we must first have a clear procedure to assign values to the states of the state machines. This is a very important step and is performed differently in various works. For instance, the work presented in [2, 5] proposes the assignment of global variables to the states of state machines by the system design engineer (referred to as the domain expert in this research). However the outcome of this approach is not always consistent as the global variables chosen by different domain experts may vary. This inconsistency can become problematic when several system engineers attempt to analyze the system requirements. The work in [3] proposes an approach which makes use of an invariant property of the system called semantic causality. The principle of semantic causality will be defined formally and explained in detail in Chapter 3 and will be used extensively throughout this thesis. The merit of using semantic causality is that since it is an invariant property of the system, using it in assigning state values will result in achieving consistency. This is one of the main reasons that the approach of [3] is

selected as the most efficient methodology for analyzing scenario-based specifications and is used and extended in this work.

Emergent behavior can be studied both at the system level and component level. This work contains the methodologies and applications related to emergent behavior at the component level. At the component level, emergent behavior occurs due to the generalization mechanism by which behavior models are constructed from scenarios.

These behaviors are not inherent to the specification and depend solely on the assumptions and the generalization techniques used in the synthesis approach. This is the reason that they have been referred to in the literature as a side effect of generalization; also known as overgeneralization [8]. It should be noted that emergent behaviors for components are not necessarily unwanted behaviors. Sometimes they may simply be considered as unexpected situations due to specification incompleteness.

1.4 Contributions

This research strives to establish comprehensive framework for analyzing the requirements and design of software systems. Figure 1.1 demonstrates the broad scope of this research. The parts highlighted in Figure 1.1 show the specific areas addressed in this thesis. As shown this framework will take input from a variety of different models and its outputs are component-level emergent behavior detection (CEBD), system-level emergent behavior detection (SEBD) and model-based detection and testing of MAS (MDTM).

Moreover as the analysis of the requirements of software systems were conducted at the component level in this research, research can be done in analyzing at the system level. In system level analysis, it is assumed that the emergent behavior for the components has already been resolved. Here scenarios are further analyzed for detecting possible system level implied scenarios [3].

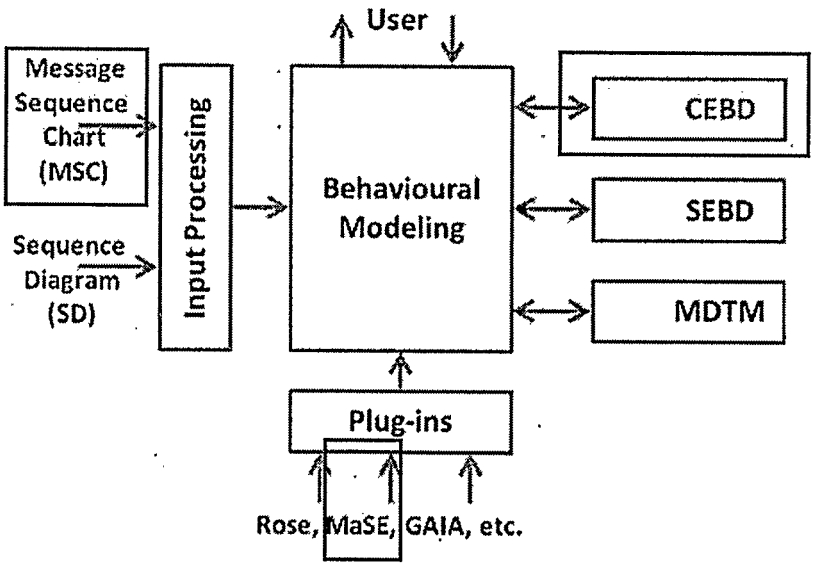


Figure 1.1 - Model-based detection system

First, this work conducts a detailed survey of several of the proposed methodologies in the literature which are devised to detect emergent behavior in the requirements of software systems. The merits and disadvantages of each methodology are explained and finally the work of [3] is selected and justified as the most effective and efficient approach to analyze scenario-based specifications. Thus the methodologies of [3] have been used as the starting point of this research. This survey and its results are presented in Chapter 2.

Next, the applicability of these methodologies is successfully tested on distributed and multi-agent systems as well as social networks. Furthermore these methodologies were extended to verify the lack of existence of particular illegal scenarios in scenario-based specifications of software systems as presented in Chapter 4.

Moreover in Chapter 5, this research attempts to link AOSE to scenario-based software engineering (SEBE) by following techniques to convert the artefacts of MaSE (which is one of the most common AOSE methodologies for the design of MAS) to scenarios.

Finally a comprehensive and practical tool has been designed to automate the proposed methodologies. Chapter 6 contains the requirement and design documents of this tool. The results of this research have been presented in a number of publications as follows:

Conference Papers

- **M. Moshirpour**, B. Far, "Formal Verification of Lack of Existence of Illegal Scenarios in The Requirements of Distributed Systems " Proceedings of the International Conference on Software Engineering and Applications (SEA 2010), Marina Del Rey, USA, November 2010.
- **M. Moshirpour**, A. Mousavi, B. Far, "A Technique and Tool to Detect Emergent Behavior of Distributed Systems Using Scenario-Based Specifications" Proceedings of the International Conference on Tools with Artificial Intelligence, Arras, France, October 2010.

- **M. Moshirpour**, A. Mousavi, B. Far, "Model Based Detection of Implied Scenarios in Multi Agent Systems" Proceedings of the International Conference on Information Reuse and Integration, Las Vegas, USA, August 2010.
- **M. Moshirpour**, A. Mousavi, B. Far, "Detecting Emergent Behavior in Distributed Systems Using Scenario-Based Specifications" Proceedings of the International Conference on Software Engineering and Knowledge Engineering, San Francisco Bay, USA, July 2010. (received Best Paper Award)

Journal Papers

- **M. Moshirpour,** A. Mousavi, B. Far, "Detecting Emergent Behavior in Distributed Systems Using Scenario-Based Specifications", International Journal of Software Engineering and Knowledge Engineering (Submitted).

Book Chapters

- **M. Moshirpour,** A. Mousavi, B. Far, "Model Based Detection of Implied Scenarios in Multi-Agent Systems", in Recent Trends in Information Reuse and Integration, Springer-Verlag (Submitted).

1.5 Structure of Thesis

This thesis is presented in seven chapters. In chapter 2 related literature is presented. Furthermore this chapter contains background information about important related topics to this work such as Scenario-based Software Engineering (SBSE) and

Agent Oriented Software Engineering (AOSE). Chapter 3 contains definitions utilized by methodologies throughout this thesis.

In Chapter 4 the effectiveness and usability of the methodologies for distributed systems are verified using two case studies of a mine-sweeping robot and an online commerce application. Furthermore this chapter contains the methodologies to formally certify that a particular illegal scenario will not emerge in a software system based on a set of given scenarios. Chapter 5 contains a novel approach to establish a link between AOSE methodologies and SBSE. This is done by following techniques to convert the artefacts of MaSE (which is one of the most common AOSE methodologies for the design of MAS) to scenarios (illustrated using MSCs) and is demonstrated using a case study of a MAS for a manufacturing system. Chapter 6 contains the design and the prototype of the software tool which automates the proposed methodologies. Finally conclusions and future work are presented in Chapter 7.

1.6 Summary

Scenarios are efficient and effective means of illustrating system requirements. Devising systematic and automated methodologies to analyze scenario-based specifications of software systems for deficiencies is highly desirable. This research attempts to establish methodologies as well as a comprehensive framework for system analysis. The motivation, objectives and the methodologies presented in this thesis have been outlined in this chapter. In Chapter 2, a comprehensive review of the related literature along with background knowledge on key concepts to this research such as multi-agent systems and Agent Oriented Software Engineering (AOSE) are presented.

Chapter Two: Related Work and Background

This chapter focuses on the related work in the literature in the area of scenario-based software engineering and proceeds to conduct a survey on the various existing methodologies devised to analyze system requirements which are defined using scenarios.

Furthermore the adopted methodology which is verified and extended in this research is presented in this chapter and the selection of this methodology is justified. In addition this chapter provides background knowledge about distributed systems, multi-agent systems (MAS) and Agent Oriented Software Engineering (AOSE).

2.1 Scenario-Based Specifications

An efficient and effective approach for defining system requirements is using scenario-based specifications. Scenarios have become popular as a powerful means of communication for system requirements due to their simplicity and expressive power [14]. Using scenarios, different groups of stakeholders can communicate their goals and ideas with regards to the software systems in a productive and efficient manner. In addition to their use in requirements engineering as shown in [14], scenarios have been utilized in other aspects of software engineering such as code synthesis [15], reverse system engineering [2] and model-based testing [16].

Scenarios are defined with variations in different works [17-20]; however in general scenarios are described as narrative stories of the interactions among system components and/or the users and the environment. Moreover scenarios are temporal

sequences of messages and thus in a scenario the order of events are clearly distinguished.

There have been several approaches proposed in the literature for representing scenarios. These approaches include using narrative text [21], annotated cartoons, video recordings, scripted prototypes and sequence charts [2, 17, 22]. Each approach entails certain merits and downfalls. For instance the textual notations are useful for documentation and are thus popular in the industry. However textual scenarios tend to be of a more informal nature and pose real challenges on automated analysis.

Among the above-mentioned approaches for presenting scenarios, sequence charts are the most efficient in terms of analysis of requirements. Moreover, due to the simplicity of their notation and expressive power, sequence charts make an efficient medium for representing scenarios. The structure of a simple sequence chart is illustrated in Figure 2.1.

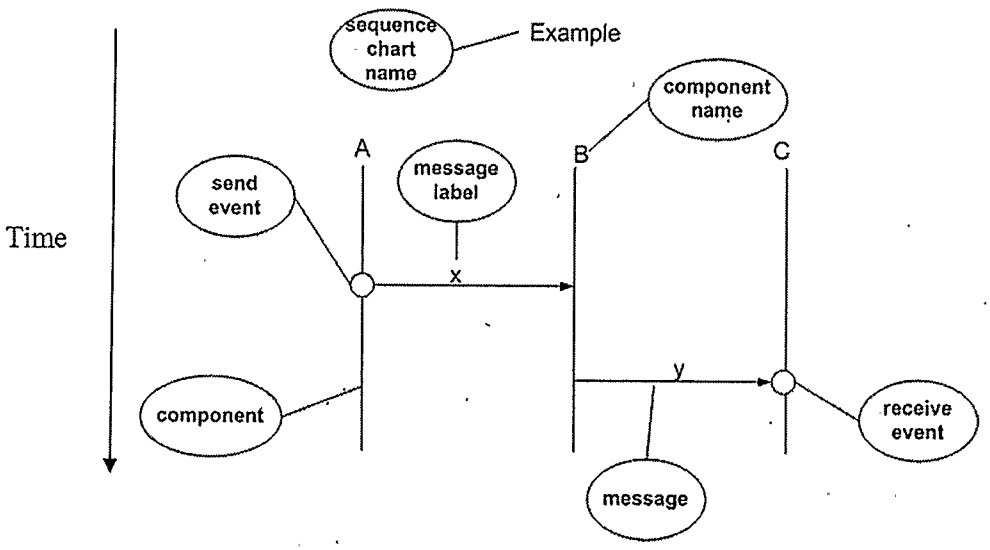


Figure 2.1 - Sequence chart

There are different variations of sequence charts in the literature. Two of the most well-known types of sequence charts that are generally used to describe scenarios are Message Sequence Charts (MSCs) standardized by the International Telecommunications Union (ITU) [23] and Sequence Diagrams developed by the Object Management Group (OMG) as a part of UML [24]. Both of these notations have undergone numerous revisions since their development.

Although MSCs and sequence diagrams vary in notations, they are both capable of representing scenarios in an efficient and intuitive manner. In this research the prime focus is on MSCs. There are several reasons for choosing to use MSCs over sequence diagrams in this research. First, the notation of MSCs is simpler than sequence diagrams; which comes as no surprise as sequence diagrams are utilized in object oriented design [23, 24]. Since in this research scenarios are used to communicate system requirements between all different kinds of stakeholders who are not necessarily computer experts, using a simpler notation to illustrate scenarios is desirable. Furthermore, due to their simplicity, MSCs serve as a powerful basis for the development of emergent behavior detection methodologies. For future work, these methodologies can then be altered to incorporate the complexities of sequence diagrams. In this case, the proposed methodologies can be used to analyze object oriented design of software systems.

However, a possible future extension for this research would be incorporating the sequence diagram notation in requirement validation methodologies. MSCs are formally defined in Chapter 3 of this thesis.

2.2 Distributed Systems

Distributed systems consist of two or more autonomous components which communicate through a network [25]. These components interact with one another in order to achieve a common goal. Concurrency and lack of central control are among the most distinct characteristics of such systems [25, 26].

Distributed systems are implemented using a variety of different architectures such as client-server, 3-tier (such as model-view-controller) and peer-to-peer [25]. The client-server and the model-view-controller architectures which are used in case studies of this research are briefly explained in the following subsections.

2.2.1 Client-Server Architecture

Client-server is a 2-tier architecture in which one or more clients request service from a centralized server. In this architecture the client is generally the consumer and requests resources and the server responds accordingly [27]. Client-server computing provides the opportunity to use cost-effective user interface, data storage, connectivity and concurrency [27].

However implementing concurrency is a challenging task. The two major problems in concurrent programming are: (1) enabling communication among two or more processes and (2) synchronizing certain actions among two or more processes [28]. This poses difficulties on the requirement elicitation and design of distributed systems which will be addressed in Chapter 4.

2.2.2 Model-View-Controller Architecture

The MVC software architecture was designed for the development of interactive applications in 1979 [29]. Based on this design, applications are divided into three different component types of models, views and controllers as shown in Figure 2.2 [29].

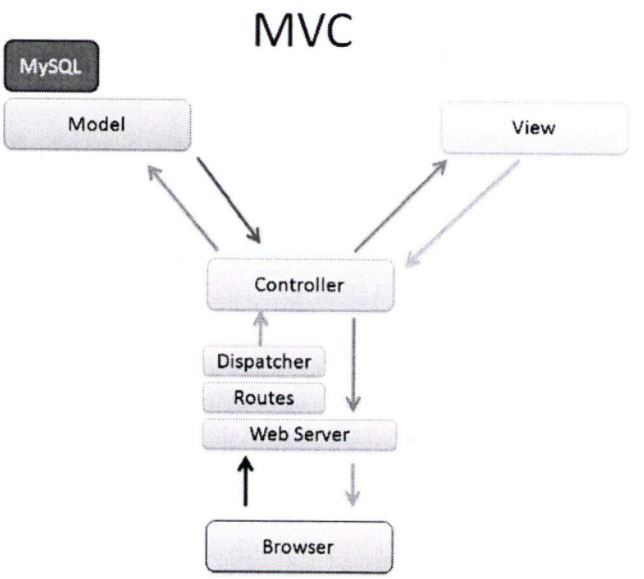


Figure 2.2 - Model-View-Controller software architecture [29]

MVC isolates the domain logic from the presentation layer and database which enables independent development along with testing and maintenance of each. The model is the layer which communicates with the database and the controller. It is responsible for maintaining the state of the application and enforces all business rules which apply to data. The view is the layer which is responsible for the interactions with the user and reporting to the controller. The controller or the business logic is the central commander of the application. It receives the input from the user through the view and

instructs the model and the view to perform certain actions based on the inputs received [29].

2.3 Multi-Agent Systems and Agent Oriented Software Engineering (AOSE)

The concept of multi-agent software systems is relatively new; dating back to the early 1980s [30, 31]. Over the years, international interest in this area has grown enormously. This is partially since agents are attractive software paradigms which provide the opportunity to exploit the possibilities presented by massive open distributed systems such as the internet [31]. Furthermore as agents are by definition automated entities, multi-agent systems (MAS) seem to be a natural metaphor for understanding and building a wide range of artificial social systems [31].

An agent is a computer system that is situated in an environment and is capable of autonomous actions in this environment in order to meet its design objectives [31]. Following this definition, it is deduced by extension that multi-agent systems (MAS) are defined as systems composed of multiple interacting computing elements, otherwise known as agents [31]. As mentioned previously, following the increase in the demand of MAS, many Agent Oriented Software Engineering (AOSE) methodologies were developed to assist the development of agent-based applications.

Part of the proposed methodology in this research is based on the MaSE analysis and design artefacts. In this section we proceed by providing an overview of the MaSE methodology and its recent applications in research and industry are presented in Sections 2.3.1 and 2.3.2 respectively. Then we provide the results of an evaluation on MaSE

methodology and its comparison with other AOSE methodologies in Section 2.3.3. We then proceed to discuss other related works on MAS verification and monitoring and various methodologies used in Section 2.3.4.

2.3.1 Agent Orientation Software Engineering Methodology: MaSE

The Multi-agent Software Engineering (MaSE) methodology is among the most well-known of AOSE techniques. MaSE strives to guide a MAS engineer from an initial set of requirements through the analysis, design and implementation of a working MAS. In MaSE, a MAS is viewed as a high level abstraction of object-oriented design of software where the agents are specialized objects that cooperate with each other via conversation and act proactively to accomplish individual and system-wide goals instead of calling methods and procedures. In other words, MaSE builds upon logical object-oriented techniques and deploys them in the specifications and design of MAS. MaSE consists of two major steps of analysis and design as outlined in Table 2.1.

Table 2.1 - MaSE methodology phases and steps [32]

MaSE Phases and Steps	*Associated Models*
1. Analysis Phase	
a. Capturing Goals	Goal Hierarchy
b. Applying Use Cases	Use Cases, Sequence Diagrams
c. Refining Roles	Concurrent task, Role Diagram
2. Design Phase	
a. Creating Agent Classes	Agent Class Diagrams
b. Constructing Conversations	Conversation Diagrams
c. Assembling Agent Classes	Agent Architecture Diagrams
d. System Design	Deployment Diagrams

The analysis phase of MaSE contains the three steps of capturing goals, applying use cases and refining goals [33] as shown in Table 2.1. This phase produces a set of roles and tasks which describe how a system satisfies its overall goals. Goals are derived from the detailed requirements and should be achieved by defined roles. A role describes an entity which acts inside the system and is responsible for achieving or assisting to achieve specific system goals. In general, the main approach of the MaSE analysis phase is to define system goals from a set of requirements and define the roles necessary to meet those goals [33].

The design phase of MaSE consists of four distinct steps of Creating Agent Classes, Constructing Conversations, Assembling Agent Classes and System Design as presented in Table 2.1. In the "Creating Agent Classes" step, the designer assigns roles to the specific agent types. During the "Constructing Conversations" step, the conversation between agent classes are defined while in the "Assembling Agent Classes" step the internal architecture and reasoning processes of the agent classes are designed. Finally in the last step of the design phase, the "System Design" step, the designer defines the number and location of the agents in the deployed system.

2.3.2 MaSE in Research and Industry

MaSE is among the most well known and powerful AOSE methodologies [31]. It has been successfully utilized in many agent-based research and industry applications. For instance the Multi-Agent Distributed Goal Satisfaction project which is a collaborative effort between Air Force Institute of Technology (AFIT), the University of Connecticut, and Wright State University, uses MaSE to design the collaborative agent

framework to integrate different constraint satisfaction and planning systems [34]. Furthermore this methodology has also been used successfully in agent-based heterogeneous database integration system [35] as well as a multi-agent approach to a biologically based computer virus immune system [36].

2.3.3 Comparing MaSE with other AOSE Methodologies

Several methodologies have been developed for the analysis and design of MAS [37]. These methodologies have been evaluated and ranked in the literature. For instance in [38], a set of 9 AOSE methodologies are evaluated based on criteria which can be considered as empirical software metrics for these techniques. Consequently in [13] AOSE methodologies were ranked according to the estimated mean effectiveness of the evaluation based on 6 dimensions of agency-related attributes, modeling-related attributes, communication-related attributes, process-related attributes, application-related attributes, and user perception attributes to support the decision of selecting the most appropriate methodology.

Among the methodologies evaluated in [13] MaSE ranked first in 3 of the proposed dimensions which were modeling-related attributes, application-related attributes, and user perception attributes. Furthermore MaSE ranked first in the overall ranking of the evaluated AOSE methodologies.

Here MaSE is compared against two other AOSE methodologies of GAIA [39] and Tropos. The aforementioned AOSE methodologies are chosen to be compared against MaSE as they are amongst the most popular and widely used techniques [40]. Similar to the approach of MaSE, GAIA utilizes roles as building blocks and captures

much of the same types of information in the design phase. However in GAIA this is done through different types of models [25]. The main difference between these two methodologies is that GAIA generates high level design and assumes details will be developed using other techniques whereas MaSE provides models and guidance on building the detailed design [41]. Tropos on the other hand takes a completely different approach compared with MaSE [26]. The focus of Tropos is mainly on the early requirements which are not addressed in MaSE at all [41]. However the Tropos early requirements approach could be used in MaSE as the goal model in the design phase [41] [42].

2.3.4 MAS Verification and Monitoring

The current work on MAS verification is divided into two categories of axiomatic and model checking approaches [41]. In [43] axiomatic verification is applied to the Beliefs, Desires and Intentions (BDI) model of MAS using a concurrent temporal logic programming language. However, it was noticed that this kind of verification cannot be applied when the BDI principles are implemented with non-logic based languages [41]. Furthermore in design by contract [44] pre and post-conditions and invariants for the methods or procedures of the code are defined and verified in runtime, and violating any of them results in an exception. However as stated in [41] the main issue is that this technique does not check program correctness, rather it simply only informs that a contract has been violated at runtime.

Model checking approaches seem to be more acceptable by the industry, because of lower complexity and better traceability compared to the axiomatic approach. Automatic verification of multi-agent conversations [45] and model checking of MAS with the MABLE programming language [46] are a few examples of model checking approaches which use the SPIN model checker [47] which is a verification system for detection of faults in the design models of software systems.

2.4 Validation Methodologies for Scenarios

As was mentioned in the previous sections, using scenario-based specifications is an efficient and intuitive approach to define system requirements; particularly for concurrent software systems such as distributed systems and MAS. Scenario-based specifications entail devising scenarios of interactions among system components and/or the users and environment such that each scenario defines a certain behavior of the system. However scenario-based specifications are prone to deficiencies such as contradictions among scenarios or incompleteness issues. Therefore devising systematic and automated methodologies to verify the correctness of requirements is very important. There are numerous methodologies proposed in the literature to verify system requirements expressed using scenarios. In this section some of these methodologies are explained and the merits and shortcomings of each are outlined. Finally the methodology chosen for this research is introduced and justified.

As mentioned previously, there are several methodologies [2, 3, 5, 7, 12, 48] which attempt to analyze system requirements (which have been expressed using

scenarios) with a systematic approach. Each methodology has devised algorithms to take scenarios as input and identify emergent behaviors in the requirements. The general structure of these algorithms is quite similar and is often done in two major steps of behavior modeling and detection of emergent behavior. The process of building behavior models has been explained in Section 1.3.1 and will be demonstrated in detail in the upcoming chapters. As mentioned in Section 1.3.1 many studies have already been conducted to facilitate the procedure of converting a set of scenarios to a behavioral model expressed by state machines [2, 5-8, 10]. Thus, the detection of emergent behavior as outlined in Section 1.3.2 is done by analyzing the state machines. To do so, a clear and concise methodology to assign values to the states of the state machine is required. This is a very crucial step and in fact is where the methodologies differ from one another. This step is important since the detection of emergent behavior is a direct result of finding identical states in behavioral models. One approach as presented in [7] is to allow stakeholders to tag scenario states. Typically labels that describe the states of the component are placed on scenario states. If two states in a scenario appear with the same label, they are considered as the same component states.

The second approach does not attempt to explicitly label the states in scenarios, but instead provides rules for identifying component states. These rules are usually based on domain-specific knowledge and additional information of the system being specified. For instance the work of [48] constructs state-charts and uses some assumptions to decide whether or not two scenario states are equal. The work of Whittle and Schumann [2, 5] attempts to use an Object Constraint Language (OCL) specification that states pre- and

post-conditions for scenario messages. OCL is part of the UML standard and is a side-effect free and set-based constraint language [49]. The OCL specifications include the declaration of *state variables.* A state variable represents some important aspect of the system such as whether or not a component is coordinating with other components. Moreover the OCL specifications enable the detection of conflicts between different scenarios and allow scenarios to be merged in a justified way [2]. The OCL specification is traversed with the MSCs to produce an evaluation of state variables for each scenario state. Scenario states that have equivalent valuations are considered to represent the same component states.

The main issue with these approaches is that their outcomes are not always consistent as the global variables and scenario labels chosen by different software engineers (referred to as the *domain experts* in this research) could vary. It is needless to say that in order to have a systematic approach for detecting emergent behavior, consistency of the methodology for different domain experts is a must. The approach followed in the work of Mousavi [3] addresses this issue by making use of an invariant property of the system called *semantic causality.* A formal definition of semantic causality is provided in definition 4 of Chapter 3 of this thesis and its pivotal role in the detection of emergent behavior has been demonstrated in the case studies presented in Chapters 4 and 5. Therefore since the methodologies presented in [3] provide an effective and efficient solution to address the issue of consistency in assigning state values in behavioral modeling, they are recognized as the better approach towards automation of such techniques and thus are closely followed in this work.

2.5 Summary

The related literature was presented in this chapter. Moreover the background knowledge for key concepts to this thesis such as scenario-based specifications, distributed systems, multi-agent systems (MAS) and Agent Oriented Software Engineering (AOSE) was provided. A comprehensive survey on the existing methodologies to analyze scenario-based specifications was conducted and the chosen methodology was introduced. In Chapter 3 formal definitions for this methodology will be presented.

Chapter Three: Methodologies and Definitions

As outlined in detail in Chapter 2 of this thesis, scenario-based specifications are prone to deficiencies such as incompleteness and contradictions. However manual review of scenario-based specifications is usually inefficient and time consuming, particularly for larger systems. Therefore devising systematic and automated methodologies to detect deficiencies in scenarios is highly desirable and cost effective.

There are several methodologies proposed in the literature for this purpose [2, 3, 5, 7, 48]. These methodologies are broken down into the two steps of:

1. Behavior modelling
2. Detection of emergent behavior

Prior to devising such methodologies, it is vital to have clear and concise syntax and definitions for the scenario notations, state models and other conceptual entities required. This chapter contains an overview of the methodologies along with formal definition of the key concepts which are used. These definitions will be further illustrated in Chapters 4 and 5 where these methodologies are applied to distributed and multi-agent systems respectively. The structure of this chapter is as follows: In Section 3.1 definitions related to scenario notations and state machines are provided. Section 3.2 contains the procedure of behavior modelling and construction of the domain theory. Detection of indeterminism is covered in Section 3.3 and the summary of the chapter is provided in Section 3.4.

3.1 Definitions

As explained in section 2.1, there are two main methods for representing scenarios; namely Sequence Diagrams (SD) and Message Sequence Charts (MSC) [24, 50]. In this research it is assumed that MSCs will be used to illustrate scenarios.

In this section, we give some definitions related to the MSC notation based on a subset of ITU definitions for MSCs [12, 23, 51].

Let P be a finite set of processes in a software system (with the total number of processes or agents $p \geq 2$) and C be a finite set of message contents (or message labels) that are passed between the processes. Let $\sum_i = \{i!j(c), i?j(c) | j \in P \backslash \{i\},\ c \in C\}$ be the set of alphabet (i.e. events) for the process $i \in P$, where $i!j(c)$ denotes an event that sends a message from process i with content c to process j, whereas $i?j(c)$ denotes an event that is received by process i a message with content c from process j. The set of alphabet will be $\Sigma = \bigcup_{i \in P} \Sigma_i$ and each member of Σ is called a message.

In the following, we try to capture a causal relationship between a message and its predecessors by defining partial Message Sequence Chart (pMSC).

Definition 1 [3] (partial Message Sequence Chart): A partial Message Sequence Chart (pMSC) over P and C is defined to be a tuple $m = (E, \alpha, \beta, \prec)$ where:

- E is a finite set of events.
- $\alpha: E \rightarrow \Sigma$ maps each event with its label. The set of events located on process i is $E_i = \alpha^{-1}(\Sigma_i)$. The set of all send events in the event set E is denoted by $E! = \{e \in E \mid \exists i, j \in P, c \in C : \alpha(e) = i!j(c)\}$ and the set of receive events as $E? = E \backslash E!$.

- $\beta: F! \rightarrow E?$ is a bijection mapping between send and receive events such that whenever $\beta(e_1) = e_2$ and $\alpha(e_1) = i!j(c)$, then $\alpha(e_2) = j?i(c)$.
- $\prec$ is a partial order on E such that for every process $i \in P$, the result of $\prec$ on E_i is a total order of its members and the transitive closure of $\{(e_1, e_2) | e_1 \prec e_2, \exists i \in P: e_1, e_2 \in E_i\} \cup \{(e, \beta(e)) | e \in E\}$ is a partial order of the members of E.

The partial order $\prec$ captures a causal relationship between the events of a pMSC. This causality basically represents two things. First, a receive event cannot happen prior to its corresponding send event. Second, a receive (or send) event cannot happen until all the previous events, which are causal predecessors of it, have already been accomplished. Obviously if all of the send events have their corresponding receive events (i.e. as defined by the function β), the structure is called a Message Sequence Chart or simply an MSC. In other words, an MSC has the same structural components as a pMSC, except that β is defined for $F! = E!$.

Following the formal definition of MSCs, it is important to define the sequence of messages between system components as shown in Definition 2.

Definition 2 [3] (projection): The projection $m|_i$ for process i in MSC m, is the ordered sequence of messages corresponding to the events for the process i in the pMSC m. For $m|_i$, $\|m|_i\|$ indicates its length, which is equal to the total number of events of m for the process i, and $m|_i[j]$ refers to j^{th} element of $m|_i$, so that if e_j is the j^{th} interaction event for process i according to the total order of the events of i in m, then $\alpha_m(e_j) = m|_i[j-1], 0 \prec$

$j < \|m|_i\|$. In $m|_i$, we call every element $i!j(c), i,j \in P, c \in C$, a send message and every element $i?j(c)$, a receive message.

State machines have been used for the behavioral modeling of scenarios in the literature [2, 3, 5, 12] and will be used for that purpose in this research as well. The formal definition of state machines is given in Definition 3.

Definition 3 [3] (Equivalent Finite State Machine for a projection): For the projection $m|_i$, we define the corresponding deterministic finite state machine $A_i^m = (S^m, \Sigma^m, \delta^m, q_0^m, q_f^m)$ such that:

- S^m is a finite set of states labelled by q_0^m to $q_{\|m|_i\|}^m$.
- Σ^m is the set of alphabet
- q_0^m is the initial state
- $q_f^m = q_{\|m|_i\|}^m$ is the final state (accepting state)
- δ^m is the transition function for A_i^m such that $\delta(q_j^m, m|_i[j]) = q_{j+1}^m, 0 \leq j \leq \|m|_i\| - 1$. Thus the only word accepted by A_i^m is $m|_i$.

Note that scenarios can be treated as *words* in a formal language, which are defined over send and receive events in MSCs. Then, a *well-formed word* for a process is one that for every receive event there exists a send event in that word, which in fact captures the essence of the definition given for a pMSC (Definition 1). On the other hand, a *complete word* for a process is one that for every send event in it, it contains the corresponding receive event. In practice, a system designer must look for complete and well-formed words for each process which is not necessarily an easy task. For any MSC

m in the set of MSCs *M*, any sequence ω of *m*, obtained from a sequence of events in *m* that respects the partial order of the events defined for *m*, is called a linearization of *m*, and is a word in the language *L(M)* of *M*.

3.2 Behavior Modeling

The model which describes the behavior of each system element (i.e. agent, component or processes) is called the *behavioral model,* and the procedure of building the behavioral models for the elements from a scenario-based specification, is called *synthesis of behavioral models*, or simply, the *synthesis process*. A widely accepted model for behavioral modeling of individual system elements is the state machine. Several studies have already been conducted to facilitate the procedure of converting a set of scenarios to a behavioral model expressed by state machines [2, 5-8, 10]. In the synthesis process, one state machine will be built for each system element: The state machine includes all of the messages that are received or sent by that element. Then the behavior of the distributed system is described by the product (parallel execution) of all the state machines of the system elements. The automation of this process has been outlined in Chapter 6 of this thesis.

The process of behavior modelling for distributed systems and multi-agent systems is illustrated in Chapters 4 and 5 respectively. In this section key definitions and concepts related to synthesis of behavioral models are explained.

A pivotal step in behavior modeling is to assign state values. This is done differently in different works as outlined in detail in Section 2.4. In this research this task

is done by making use of an invariant property of the system referred to as semantic causality as defined formally in Definition 4 [3].

Definition 4 [3] (Semantic causality): A message $m|_i[j]$ is a semantical cause for message $m|_i[k]$ and is denoted by $m|_i[j] \overset{se}{\rightarrow} m|_i[k]$, if agent i has to keep the result of the operation of $m|_i[j]$ in order to perform $m|_i[k]$.

Semantic causality is an invariant property of the system and is part of the system's architecture and the domain knowledge. Therefore it is independent of the choices made by the domain experts. In other words, we let the current state of the system component to be defined by the messages which that particular component needs in order to perform the messages that come after its current states.

3.2.1 Domain Theory

Based on the concept of semantic causality introduced in Definition 4, it is deduced that in order to evaluate state values of the resulting FSM, a domain theory which consists of the domain knowledge of the system must be constructed. A formal definition for the domain theory is provided in Definition 5.

Definition 5 [3] (Domain theory): The domain theory D_i for a set of MSCs M and agent $i \in P$ is defined such that for all $m \in M$, if $m|_i[j] \overset{se}{\rightarrow} m|_i[k]$ then $(m|_i[j],\ m|_i[k]) \in D_i$.

However building a domain theory can be very time-consuming. Therefore as a part of this systematic approach, building a light domain theory is introduced. The concept of light domain theory is closely tied to the calculation of state values as defined in Definition 6.

Definition 6 [3] (State value): The state value $v_i|(q_k^m)$ for the state q_k^m in eFSM $A_i^m = (S^m, \Sigma^m, \delta^m, q_0^m, q_f^m)$ is a word over the alphabet $\Sigma_i \cup \{1\}$ such that $v_i|(q_f^m) = m|_i[f-1]$, and for $0 < k < f$ is defined as follows:

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

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

iii) $v_i|(q_k^m) = 1$, if none of the above cases hold

Using this definition, it becomes evident that only states with the same incoming transitions have the potential to exhibit indeterministic behavior. Assigning state values to states of eFSMs is done by making use of semantic causality as defined in Definition 4.

3.3 Detection of Indeterminism

The concept of emergent behavior and the process of detecting indeterminism have been presented in section 1.3.2 and will be illustrated in detail in Chapters 4 and 5.

Upon constructing the behavior model and by assigning state values based on semantic causality, the basis for comparing states and consequently discovering identical states is established. Identical states are defined in Definition 7 as follows:

Definition 7 [3] (Identical states): Two states q_j^m and q_k^n of process i, (m and n could be the same) are identical if one of the following holds:

i) j = k for $0 \leq t \prec j: m|_i[t] = n|_i[t]$

ii) $v_i(q_j^m) = v_i(q_k^n)$

As stated previously, emergent behavior usually happens when the system components become confused as the result of identical states. Definition 7 can be used to systematically detect emergent behavior at the component level.

3.4 Summary

Prior to devising systematic and automated methodologies to analyze software requirements, it is vital to have clear and concise definitions for scenario notations and key concepts of the techniques. This chapter contains formal definitions for MSCs which are used in representing scenarios and FSMs which are used in behavior modelling. Furthermore the steps of the methodologies used along with the necessary formal definitions for each step are formally defined.

These definitions are illustrated in great detail in Chapters 4 and 5 where the proposed methodologies are applied to verify the requirements of distributed systems and multi-agent systems respectively.

Chapter Four: Detection of Emergent Behavior in Distributed Systems

This research proposes two distinct approaches towards prevention against emergent behavior. The first approach involves compiling all system scenarios and conducting behavioral modeling in order to discover all cases of emergent behavior in system's requirements. This approach is applied to an illustrative example of a mine sweeping robot. The robot has been designed to have multiple independent processing units, indicating that there is no centralized control.

The second approach involves ensuring the lack of emergence of particular behaviors [52]. In this approach system engineers will have a set of undesired scenarios and the system requirements (which are also expressed using scenarios) are checked to verify that the illegal scenarios from this set cannot be derived from them. This approach is demonstrated using a case study of a common online commerce system.

Furthermore it has been demonstrated in this chapter that scenario-based specifications can be used in agile software development to verify the consistency of user stories.

4.1 Background

To analyze system requirements with the proposed methodologies in this research, first the behavioral model of the system is built and then the behavioral models are checked for emergent behavior. Both processes of behavioral modeling and detection of emergent behavior were explained in the first two chapters and are illustrated in this chapter using real-life examples of distributed systems. It is important to note that while

these examples are kept simple for the sake of brevity, they are quite illustrative of the proposed methodologies in this research.

4.2 Case Study: Mine Sweeping Robot

Let's consider the prototype of an automated mine-sweeping robot shown in Figure 4.1.

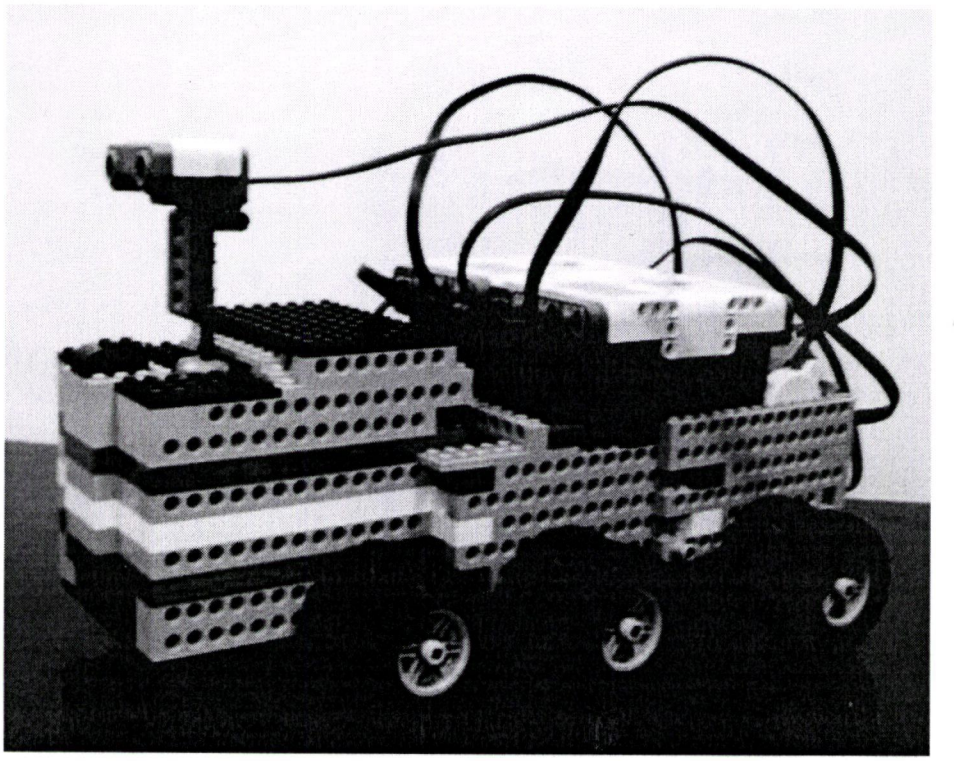


Figure 4.1 - Prototype of a mine sweeping robot

The robot's mission is to navigate through a maze-like course, which resembles the layout of the streets of a city, for which the robot has no map and has to investigate it by utilizing its sensory information (i.e. ultrasonic and/or GPS data). At the same time, it has to identify and mark the location of mines. For this prototype version it is assumed

that mines emit infrared signal which is detectable via the infrared sensor. In order to provide the robot with more computation power and additional control for the motors and different types of sensors, two multi-core CPU units are utilized. The units are built on separate boards connected via a simple but reliable connection protocol. The two CPUs interact using the client-server architecture; one of the CPUs acts as the client and the other acts as the server. As there is no sophisticated operating system in charge of the control and scheduling of the processes and threads, the design of the robot must account for the proper management of all processes and their interactions in a logical and efficient manner.

For the sake of simplicity, let's assume that the robot has only two sets of sensors: an ultra-sonic sensor which is used for navigation purposes, and an infrared sensor to detect mines. Given the sensors of the robot, its design is as follows: Both of these sensors are connected to the client CPU. Thus the client receives signals from sensors, processes the message and sends them to the server CPU to act upon them accordingly. The processes *Client Controller* and *Server Controller*, depicted in Figure 4.2, are in charge of the motors responsible for the wheels on the left and right sides of the robot. Each process is also responsible for sending and receiving messages to and from the other. The server controller is also in charge of the motor of the mechanism which dispenses a flag on the location in which a mine has been detected. The *ULTS Motor Controller* process is responsible for the motor in charge of the rotation of the ultra-sound sensor which is necessary for optimum navigation through the maze.

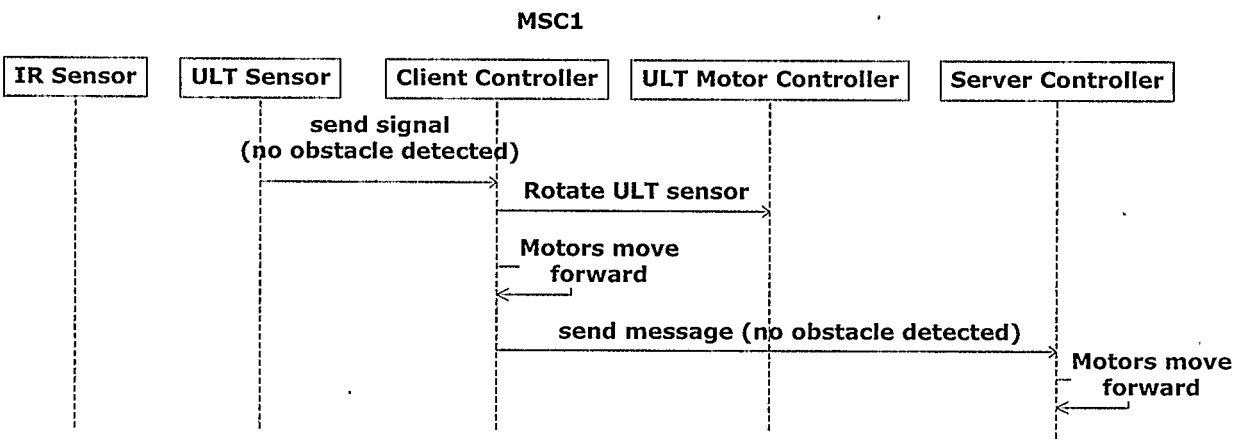


Figure 4.2 - Robot is moving forward with no obstacle in its way

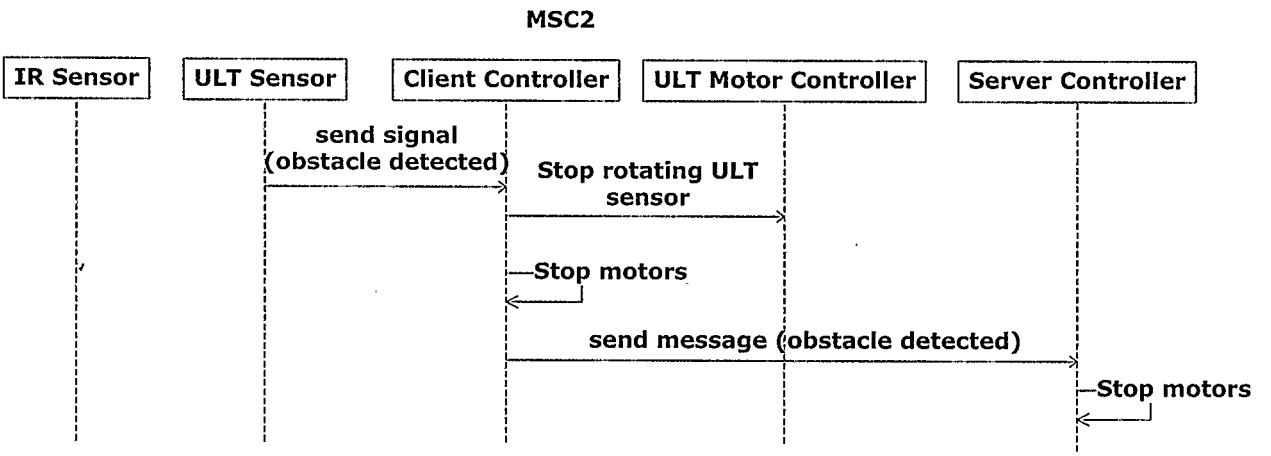


Figure 4.3 - Robot is halted due to the detection of an obstacle in its path

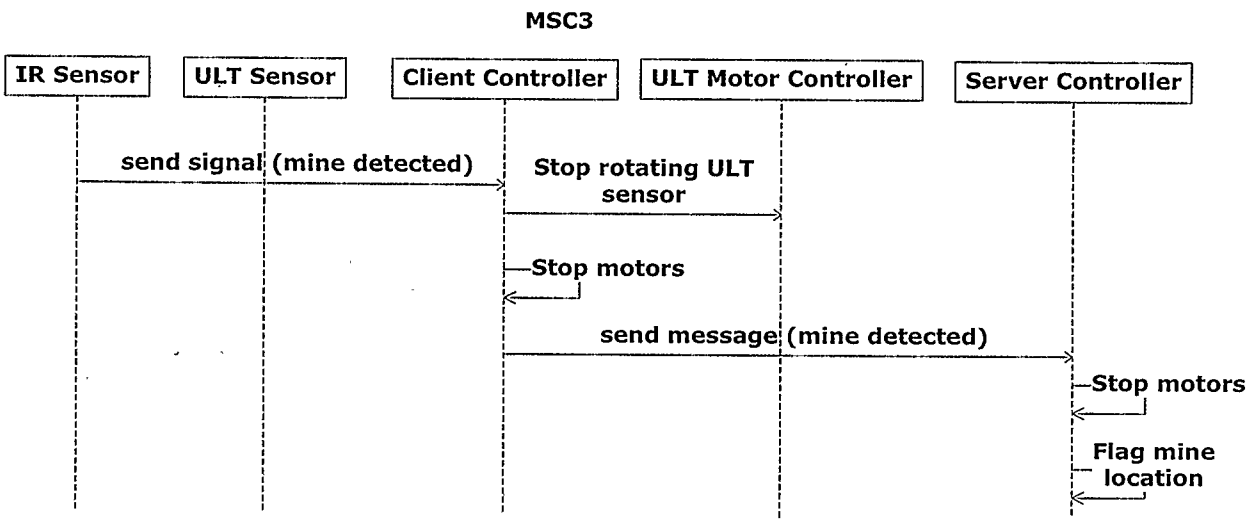


Figure 4.4 - Robot stops due to the detection of mines

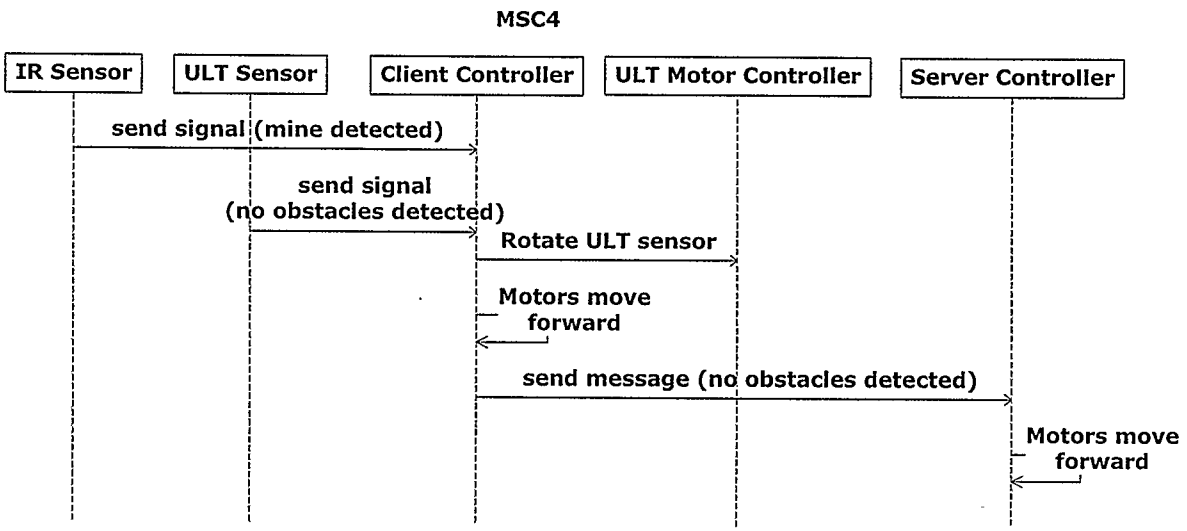


Figure 4.5 - Client controller receives a "no obstacle" detection message from the ultra-sound sensor after receiving the mine detection message from the IR sensor which results in missing the mine

Partial behavioral scenarios for this robot are represented by message sequence charts. The message sequence chart 1 (MSC1), shown in Figure 4.2, illustrates a scenario where the robot is moving forward with no obstacles in its way. MSC2 (Figure 4.3), shows a scenario where the robot has been halted due to the detection of an obstacle in its path while MSC3 (Figure 4.4) illustrates a scenario where the robot stops because of the detection of a mine (based on the signal received from the IR sensor) which is a pre-requisite for the mine-flagging operation.

As can be seen from MSC2 and MSC3 (Figures 4.3 and 4.4) there are two events which cause the robot to halt: (1) detecting an obstacle on the way performed by the ultrasound sensor; and (2) detecting a mine which is done by the infrared sensor. Similarly there are two events which trigger the motion of the robot: (1) detection of a free path (i.e. no obstacles in the way) by the ultrasound sensor; and (2) the completion of the mine-flagging operation. MSC4 shown in Figure 4.5 illustrates an emergent behavior that might occur as a result.

An important observation to be made is the generalization of messages to indicate their purpose rather than their specific implementation. Consider message "send signal (some signal)" which is sent from either of the sensors to the client controller process as shown in each of the MSCs (Figures 4.2 - 4.4). For instance in MSC1 (Figure 4.2) the content of the message sent from the sensor to the client controller is "no obstacles detected" while in MSC3 (Figure 4.4) the content of the message is "mine detected". It is important to note that although the content of these messages are different, the purpose of the two messages remains the same. That is, the client controller process expects to

receive a message (regardless of the content of the message) from a sensor. Therefore the message sent from either sensor to the client controller process is "send signal" and the content is included in brackets only for clarity.

4.3 System Behavior Modeling

In this section, the synthesis of state machines from MSCs which is the first part of the systematic approach to analyze system requirements (expressed using scenarios) is illustrated using the example of a mine sweeping robot.

As mentioned previously, the procedure of construction of finite state machines (FSMs) from message sequence charts (MSCs) is referred to as behavior modeling. For any process *i* of a partial MSC described in Definition 1, an equivalent finite state machine (Definition 3) can be constructed. For instance, Figure 4.6 shows the eFSM constructed for the client controller process in MSC1.

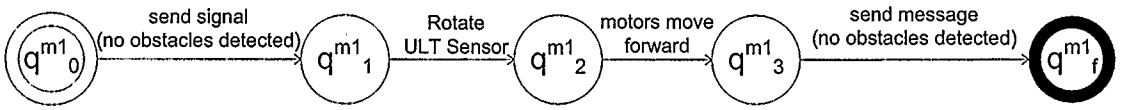


Figure 4.6 - eFSM for the client controller in MSC 1

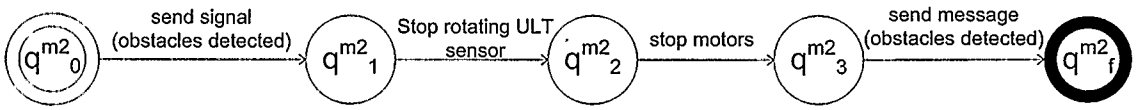


Figure 4.7 - eFSM for the client controller process in MSC 2

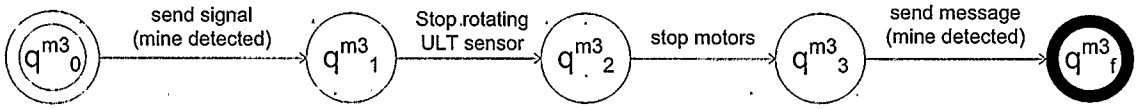


Figure 4.8 - eFSM for the client controller process in MSC 3

A comprehensive definition for identical states is needed for synthesis of behavior models from scenarios. To achieve this we must first have a clear procedure to assign values to the states of the eFSMs. As outlined in detail in Chapter 2 of this thesis, this is a very important step and is performed differently in various works. For instance, the work presented in [2, 5] proposes the assignment of global variables to the states of eFSMs by the domain expert. However the outcome of this approach is not always consistent as the global variables chosen by different domain experts would vary. Therefore to achieve consistency in assigning state values, the approach of [3] which is making use of an invariant property of the system called semantic causality given in Definition 4 (Chapter 3) is followed.

For example, in MSC1 in Figure 4.2, the message "send signal" is a semantic cause for message "rotate". As semantic causality is an invariant property of the system and is part of the system's architecture and the domain knowledge, it is independent of the choices made by the domain experts. In other words, we let the current state of the process to be defined by the messages that the process needs in order to perform the messages that come after its current states. Therefore using semantic causality, we proceed to build the system's domain theory which is defined in Definition 5 of Chapter 3.

Following the robot example, since the message "send signal" is a semantic cause for message "rotate", both messages are part of the domain theory. However building the domain theory can be very time-consuming. Therefore as a part of this systematic

approach, building a light domain theory is introduced. The concept of light domain theory is closely tied to the calculated state values as defined in Definition 6. Using this definition, it becomes evident that only states with the same incoming transitions have the potential to exhibit indeterministic behavior. Assigning state values to states of eFSMs is done by making use of semantic causality as defined in Definition 6 of Chapter 3.

For instance in order to calculate the state value for state q_2^{m1} we proceed as follows: from the domain theory of the system (Definition 5) we learn that the maximum index j for which $m1\,|_{client\ controller}\,[j-1]$ is a semantic cause for a message in the transitions after q_2^{m1} is j = 1 for which $m1\,|_{client\ controller}\,[j-1] = send\ signal.$ That is to say that for example the message "send signal" is a semantic cause for message "motors move forward". Therefore from case (i) of Definition 6 we obtain: $v_{client\ controller}|(q_2^{m1}) = m\,|_{client\ controller}\,[2-1]v_{client\ controller}|(q_1^{m1})$.

In order to calculate $v_{client\ controller}|(q_1^{m1})$ we observe that "send signal" is the only semantic cause after q_1^{m1}, thus case (ii) of Definition 6 holds and we get $v_{client\ controller}|(q_1^{m1}) = send\ signal.$ Therefore we have $v_{client\ controller}|(q_2^{m1}) = (rotate)(send\ signal)$. By following the same approach we get the state value for q_2^{m3} to be $v_{client\ controller}|(q_2^{m3}) = (rotate)(send\ signal)$.

From these examples it becomes clear that semantic causality is an invariant property of the system and is not affected by the preferences of the domain expert. To complete behavior modeling for the system, for each process a final FSM which is the union of its corresponding eFSMs from different scenarios is to be built. Figure 4.9

demonstrates the union of the three eFSMs built from MSCs 1-3. As established in Definition 6 the value of start and end states are defined to be equal to 1.

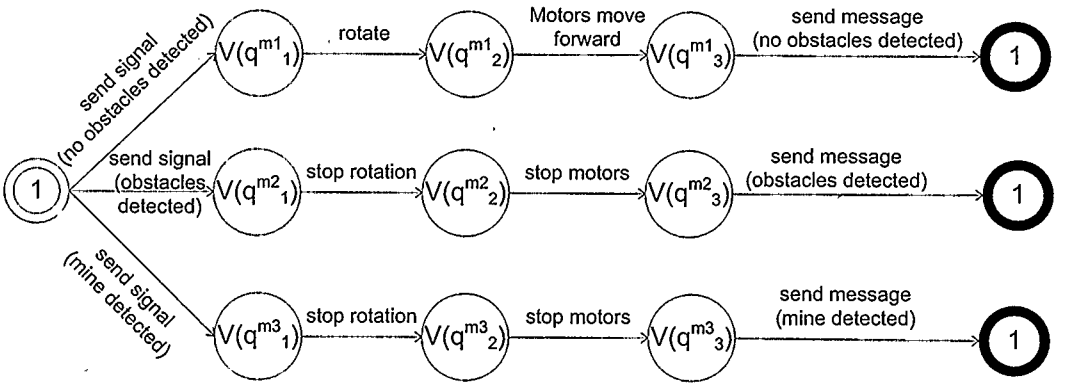


Figure 4.9 - The union of eFSMs built from MSCs 1-3

4.4 Detection of Emergent Behavior

As demonstrated in the previous section, by assigning state values based on semantic causality, the basis for comparing states and consequently discovering identical states is established. Identical states are formally defined in Definition 7 of Chapter 3.

By considering the union of the eFSMs demonstrated in Figure 4.9, the identical states that correspond to case (ii) of Definition 7 are determined. As identical states are possible areas in which the system might get confused over what course of action to take, these states are recorded and presented to the domain expert to be analyzed and reconsidered.

The FSM shown in Figure 4.10 demonstrates the manner by which identical states in the mine sweeping robot example can result in emergent behavior. As mentioned

earlier, the message "send signal" sent from either sensor to the "client controller", should be considered as a transition regardless of the content of that message since the client controller process is waiting to take any message that is sent from the sensors. This causes $q_1^{m1}, q_1^{m2}, q_1^{m3}$ to have identical state values and satisfy case (ii) of Definition 7.

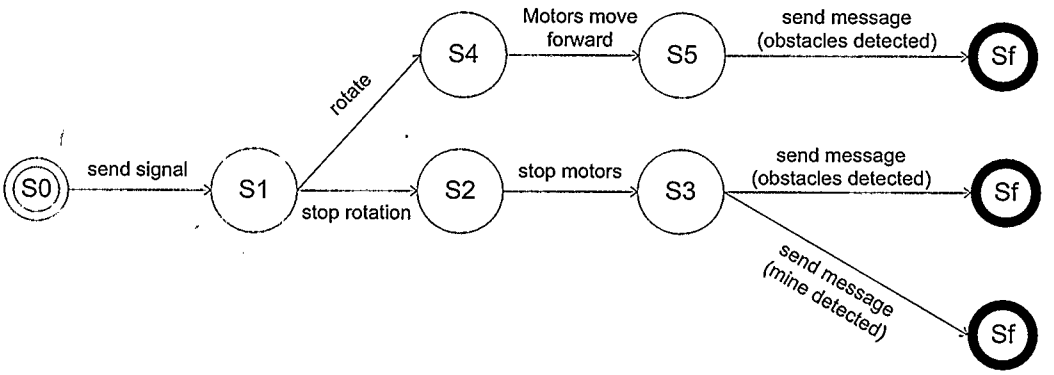


Figure 4.10 - Resulted DFA after merging identical states

As stated in Section 4.2, as a result of generalization, the content of the message "send signal" is not considered. However the content of the "send signal" will make a difference in the behavior of the robot. Therefore these identical states may result in emergent behavior. As shown in Figure 4.10, by having the states q_1^{m1}, q_1^{m3} as identical states, the robot gets stuck in a state of confusion between moving forward as it detects no obstacles in its path, and setting a flag that a mine is detected by the infrared sensor. This state of deadlock is illustrated by MSC4 in Figure 4.5.

4.5 Verification of Lack of Existence of Illegal Scenarios in Distributed Systems

In the development of larger distributed systems, it is often desirable to ensure that certain scenarios do not emerge in the system's behavior. This section introduces

methodologies which ensure formal verification of lack of such scenarios. These methodologies are demonstrated using a case study of a common online commerce system.

4.5.1 Case Study: Online Commerce System

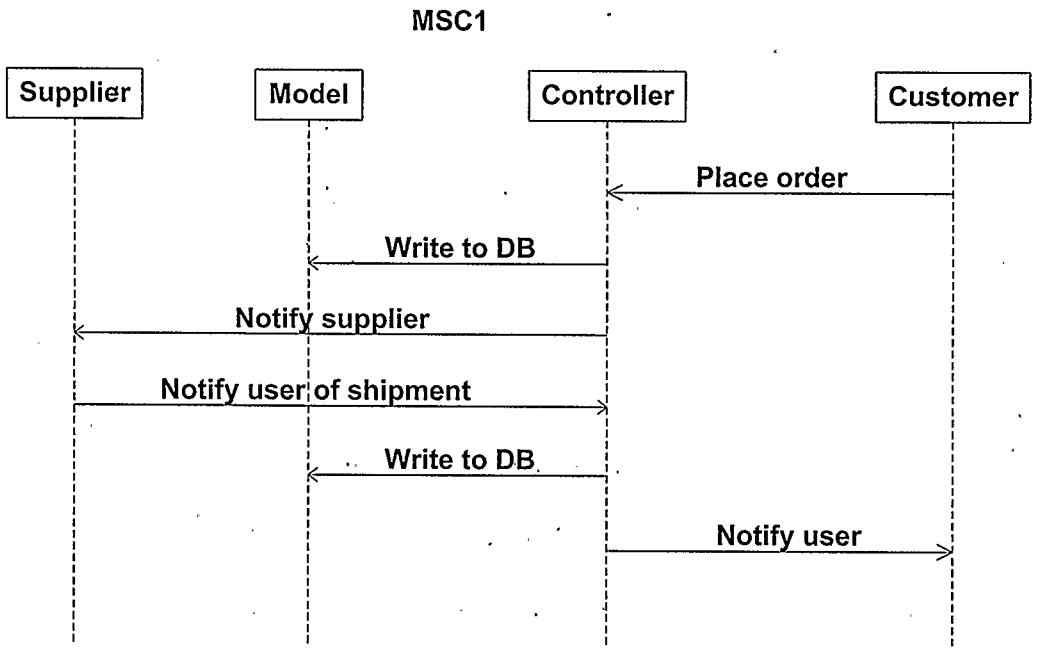
Online commerce applications are among the most widely used distributed systems which have helped transform the nature of free trade as was traditionally known. Websites such as Amazon and eBay are among the largest of such applications. The case study used in this paper is a typical example of a large scale online commerce application; very similar to the likes of amazon.com. This application is a distributed client-server system with potentially thousands of users. This system is constructed based on the widely accepted Model-View-Controller (MVC) architecture which is explained in section 2.2.2.

4.5.1.1 System Requirements

A subset of system requirements are expressed using message sequence charts (MSC) as shown in Figures 4.11-4.13. As it can be seen, these charts essentially present the interactions among the three layers of the MVC architecture and system users. In this case study, two different classes of users are assumed; the customers and the suppliers. The users belonging to the costumer class use the system to browse or search for goods and ultimately to place orders. The users on the supplier side use the system to view orders and update shipping information and order statuses. It is needless to say that both of these groups of users interact with the view layer of the MVC architecture.

In the interest of simplicity and efficiency, four different processes are described in these scenarios which incorporate both the user classes as well as the three layers of the MVC architecture. As illustrated in Figures 4.11-4.13, the supplier and customer user classes are combined with the view layer of MVC while the controller and model layers are each distinctly represented.

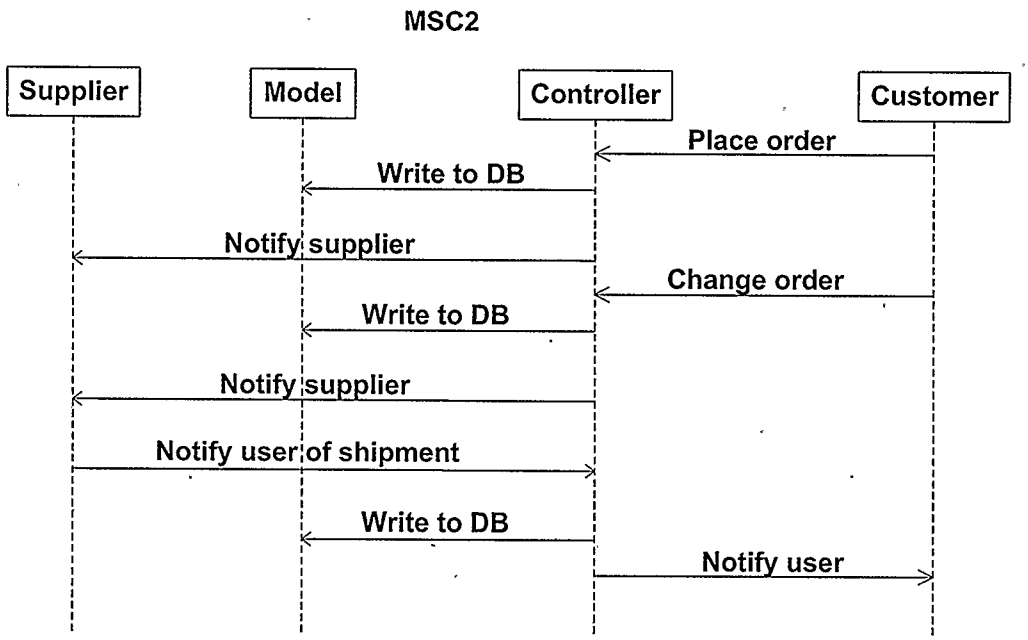
These scenarios describe the process of placing orders by the customer and shipment of orders by the supplier. MSC1 illustrated in Figure 4.11 demonstrates a scenario where the customer places an order which is received by the supplier. The supplier then ships the order and the customer is notified.



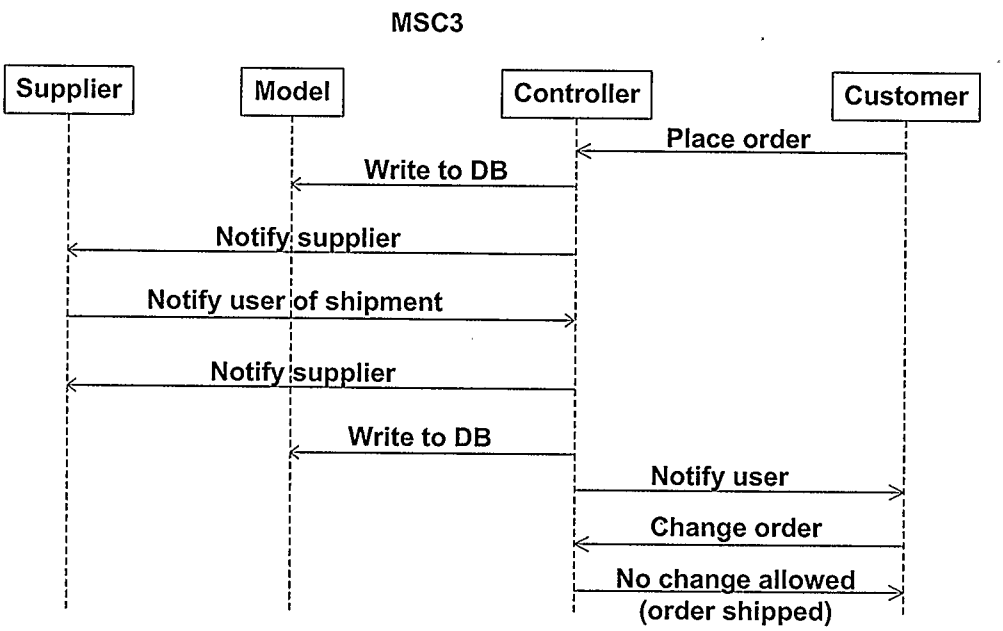
4.11 - Customer places an order which is shipped by the supplier

MSC2, illustrated in Figure 4.12 presents a scenario where the customer places an order which is received by the supplier. However before the order is shipped by the

supplier, the customer applies changes to the order which are also received by the supplier. The supplier ships the order according to the changes made by the customer. MSC3 shown in Figure 4.13 illustrates a scenario where the customer tries to make changes after the order has been shipped by the supplier but this request is rightfully denied.



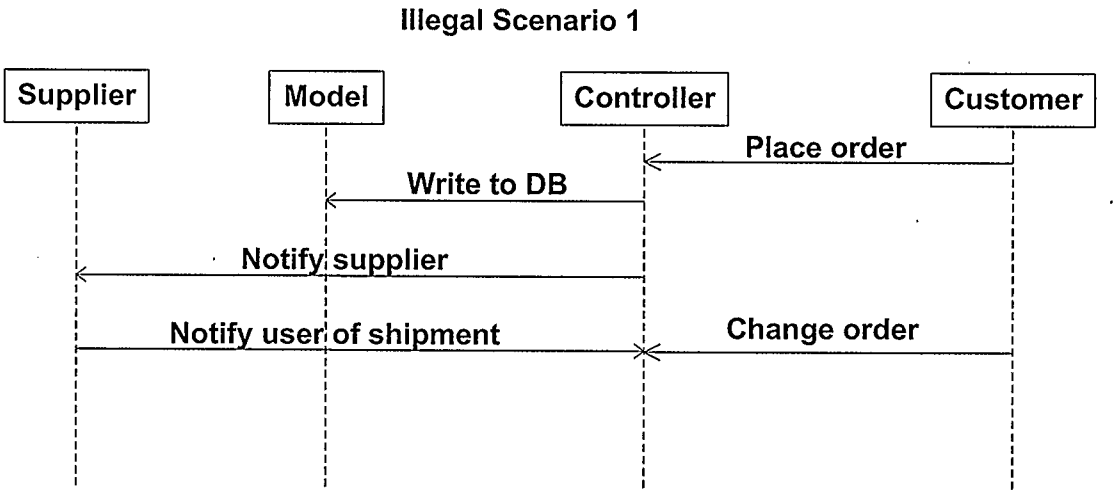
4.12 - Customer places an order, then makes changes. Changes are received by supplier and order is shipped



4.13 - Customer places an order. Order is received by the supplier and is shipped. Customer attempts to change order but is denied.

4.5.1.2 Illegal Scenarios

Figure 4.14 represents a possible illegal scenario which is undesirable to occur. Therefore it is required to verify that such a scenario will not emerge from the system requirements, or in other words, from the scenarios which describe the system's behavior.



4.14 - Illegal scenario

The MSC shown in Figure 4.14 illustrates a scenario where the customer attempts to make changes at the exact same time as the supplier notifies the system of the order shipment. That is, the controller process receives the two messages of "notify user of shipment" and "change order" almost simultaneously. Needless to say that in this case the process controller would be confused as to what course of action to take. It is important to note that although this is a simple scenario and is very unlikely to occur due to the available technology, it is still illustrative for the purpose of illegal scenario.

4.5.2 Formal Verification Methodology

This methodology takes two different sets of scenarios which are expressed using MSCs as follows:

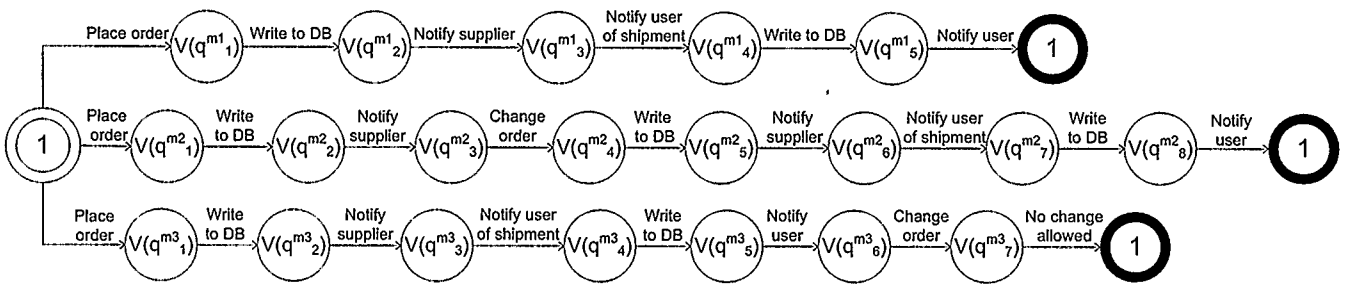
A. A set of MSCs containing scenarios which describe system's behavior

B. A set of illegal scenarios which are undesirable to occur

By having these two sets of scenarios which were given in Section 4.5.1, this methodology is to verify that scenarios in set B cannot be derived from scenarios of set A.. In other words this methodology ensures that system's behavior does not contain scenarios from set B. As for the previous approach (done for the mine-sweeping robot), this methodology is divided into two parts of constructing the behavioral model and ensuring the lack of invalid scenarios in the built models. Since these steps were demonstrated in detail in the previous sections of this chapter, only the results of each are presented for this approach.

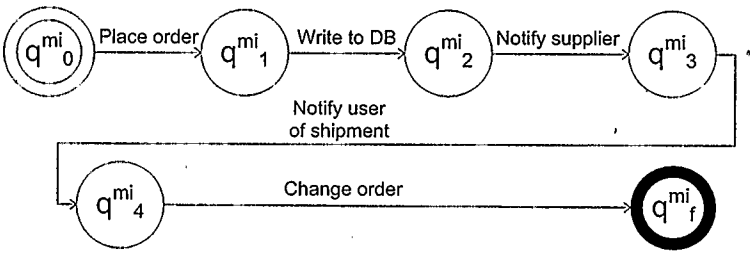
4.5.2.1 Synthesis of Behavior Model

The behavior model for the controller process of the scenarios in set A is shown in Figure 4.15.



4.15 - The union of all eFSMs built for process controller from MSCs 1-3 (Figure 4.11-13)

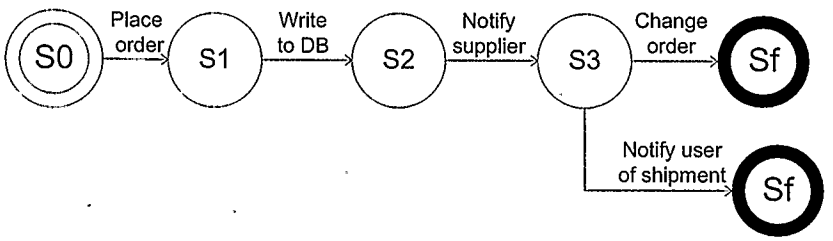
Consequently, the behavior model of the controller process from the scenario in set B is constructed as shown in Figure 4.16.



4.16 - Behavior model for the controller process from scenario of set B

4.5.2.2 Detection of Emergent Behavior

Figure 4.17 illustrates the constructed FSM as the result of the merging of identical states.



4.17 - Resulted FSM after merging identical states

Upon the identification of cases of emergent behavior in system requirements (case A), a new set C can be constructed to contain their related behavioral models. Therefore if a behavior model built based on scenarios in set B does not match a behavioral model in set C, it is verified that the system will not contain that particular illegal scenario. Conversely if a behavior model constructed based on the scenarios of set B is equal to the behavioral model in set C, the verification has failed. By comparing the FSM in Figure 4.17 with the behavior model constructed from set B (Figure 4.16) it becomes evident that the illegal scenario of set B can potentially emerge from the scenarios of set A.

4.6 Using the Proposed Methodologies in Agile Development

Agile software development has been adopted as one of the most practical approaches to software development. The main reason for this is the tendency to incorporate changes in requirements [53]. In general, agile follows iterative development which advocates frequent and regular software releases [53]. This allows new versions of the software to be released to users quickly and frequently. Therefore users can respond to these releases with feedback, changes of their requirements and general comments. These changes and comments can then be incorporated in the future iterations and releases of the software [53].

Developing software in iterations implies the execution of software development lifecycle in each release. That is each iteration consists of requirements elicitation, analysis and design, development and testing [53, 54]. A common belief about agile methods is that they can benefit from using more quantified approaches across the entire development life cycle. The research presented in [55] discusses such things as quantification of the requirements, design estimation, and measurement of the delivered results and proceeds to highlight the advantages of adopting such approaches. In this section some of the merits of quantification of agile methodologies particularly in the requirements engineering portion of each iteration is discussed. Furthermore the advantages of incorporating the methodologies proposed in this thesis into agile development to detect defects in software requirements are outlined.

As it was mentioned in the previous sections of this thesis, collecting and analyzing user requirements is very complex and is often a problematic process in

software development projects. There are several approaches, which suggest ways of managing user's requirements; some of the most well-known are IEEE 830 software requirements specification (SRS), use cases, interaction design scenarios, etc [56]. Many software experts believe the real user requirements emerge during the development phase. By constantly viewing functional sub-systems of the whole system and participating, in fact, in all phases of system development, customers/users can revise their requirements by adding, deleting, or modifying them [56].

However it is needless to say that the traditional waterfall model does not allow such flexibility concerning not only the management of user's requirements, but also the entire software development process in general. Agile methodologies represent this different approach since the iterative and incremental way of development they propose includes user requirements revision mechanisms and user active participation throughout the development of the system [56]. The most famous approach concerning requirements specification among the supporters of the agile methodologies is probably user stories [56]. In fact, user stories are one of the primary development artefacts for extreme programming (XP) project teams. XP creator Beck defines a user story as: "*One thing the customer wants the system to do. Stories should be estimable at between one to five ideal programming weeks. Stories should be testable. Stories need to be of a size that you can build a few of them in each iteration*" [57]. Figure 4.18 demonstrates the use of user stories in agile development.

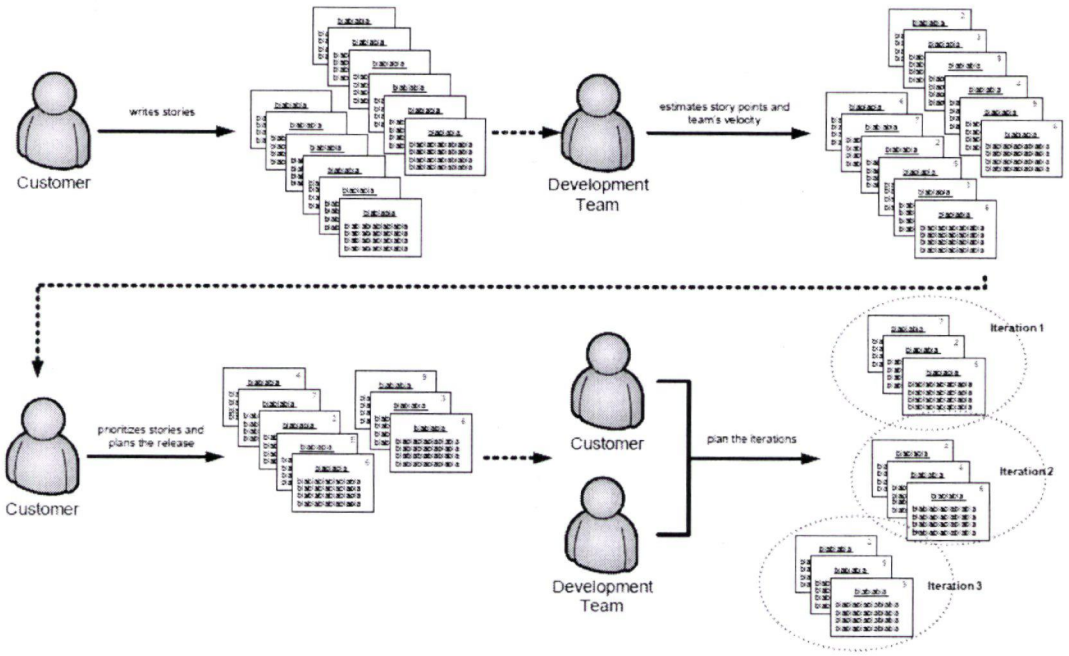


Figure 4.18 - Agile development with user stories

There are several ways to represent user stories: it can be done using text [55-57] or as suggested in this thesis, they can be illustrated using sequence diagrams or message sequence charts (MSC). As mentioned previously in this research, the advantage of using MSCs is that they are easy to understand and have a great expressive power. Furthermore due to the concise notation of MSCs they can be used in systematic methodologies proposed in this research to test and verify the correctness of user stories. Incorporating requirement and design validation methodologies in agile development goes a long way towards quantifying agile approaches and can be particularly useful in building large scale and complex software such as distributed or multi-agent systems.

4.7 Summary

In this chapter two different approaches for the validation of software requirements for distributed systems were introduced. The first approach which was illustrated using the case study for a minesweeping robot involves compiling all system scenarios and conducting behavioral modeling in order to discover all cases of emergent behavior in system's requirements. The second approach which was illustrated using an online commerce application involves ensuring the lack of existence of particular scenarios in the system behavior.

Furthermore it has been demonstrated in this chapter that scenario-based specifications can be used in agile software development and that the proposed methodologies in this research can be utilized effectively in quantifying agile approaches. In Chapter 5, these methodologies are used to analyze and validate design documents of multi-agent systems.

Chapter Five: Detection of Emergent Behavior in Multi-Agent Systems

Multi-agent systems (MAS) are efficient solutions for commercial applications such as robotics, business commerce applications, information retrieval and search engines. In MAS, agents are usually designed with distribution of functionality and control. Lack of central control implies that the quality of service of MAS may be degraded because of possible unwanted behavior at runtime, commonly known as emergent behavior. Requirements and design of multi-agent systems is particularly challenging due to the sophisticated interactions of automated entities. Therefore system faults such as deadlock or feature interaction may arise in MAS. A feature is defined as an identified piece of functionality which is added as an extension to a base system. By extension feature interaction is the situation where two features contradict or have a negative effect on each other [58]. However it is important to notice that emergent behavior is not necessarily always negative. Emergent behavior by definition is a behavior exhibited by the system, but is not explicitly a part of its specifications. Thus although feature interactions are classified as a type of emergent behavior and can be detected using the proposed methodologies in this research, they are only a subset of emergent behavior.

Detecting and removing emergent behavior during the design phase of MAS will lead to huge savings in deployment costs of such systems. Effective and efficient design validation of MAS requires the development of systematic and automated methodologies to review MAS design documents [59]. Although the increasing demand for multi-agent systems (MAS) in the software industry has led to the development of several Agent

Oriented Software Engineering (AOSE) methodologies, the AOSE methodologies usually do not fully cover monitoring and testing. In this thesis, a methodology to help MAS developers verify, test and monitor MAS design is introduced. In Chapter 4 the detection of emergent behavior in the requirements of distributed systems were illustrated. This chapter goes further and introduces a methodology to analyze the design of multi-agent systems for emergent behavior.

This method uses MAS design and analysis artefacts created by MaSE which is one of the most powerful and famous AOSE methodologies. In this method, the design artefacts of MaSE are converted to scenario-based specifications, which are very similar to UML's sequence diagrams [60, 61]. These specifications are then used to analyze the design of MAS to ensure the lack of emergent behavior.

5.1 Background

Over the years, international interest in multi-agent systems (MAS) has grown enormously. This is partially since agents are attractive software paradigms which provide the opportunity to exploit the possibilities presented by massive open distributed systems such as the internet [31]. Furthermore as agents are by definition automated entities, multi-agent systems (MAS) seem to be a natural metaphor for understanding and building a wide range of artificial social systems [31]. As the result of this growth in agent technology, many Agent Oriented Software Engineering (AOSE) methodologies such as GAIA and MaSE have emerged to assist in the development of MAS [37].

As mentioned previously, this research proposes a systematic methodology that can be automated to review MaSE design artefacts in order to discover and remove

emergent behavior. MaSE provides a comprehensive and detailed approach for the analysis and design of MAS. This methodology utilizes several diagrams and models which are driven from the standard Unified Modeling Language (UML) to describe the architecture-independent structure of agents and their interactions [33]. The main focus in MaSE is to guide a MAS engineer from an initial set of requirements through the analysis, design and implementation of a working MAS. In MaSE, a MAS is viewed as a high level abstraction of object oriented design of software where the agents are specialized objects that cooperate with each other via conversation and act proactively to accomplish individual and system-wide goals instead of calling methods and procedures.

MaSE incorporates models which illustrate the interactions among different roles within agents as well as the conversations between the agents themselves. In other words, different scenarios that make up the overall functionality and behavior of the MAS can be extracted from these models. Having access to the scenario-based specifications of MAS is considered greatly valuable as scenarios are not only an efficient way to describe the system's requirements and behavior, but they can also be used to examine the system for possible design faults such as emergent behaviors. In this research a systematic approach is proposed to extract MSCs from MaSE artefacts. These MSCs are then used to examine the design of the MAS.

This chapter is organized as follows: In section 5.2 the multi-agent system for manufacturing which is used as the case study throughout this paper is introduced. The analysis and design models of MaSE for the system are provided in this section and the processes of extracting MSCs from these models is explained. Section 5.3 consists of the

behavioral modeling of the manufacturing MAS. Detection of implied scenarios and emergent behavior is discussed in Section 5.4 and the summary of the chapter is presented in Section 5.5.

5.2 Case Study: MAS for Manufacturing System

The methodologies proposed in this research are explained using a MAS of a manufacturing system. An Automated Manufacturing System (AMS) is an integrated system of equipments and processes controlled via computer applications or a network of them that is capable of producing a variety of products with flexibility and efficiency. A manufacturing system automated by agent-based technology is composed of several autonomous and intelligent agents that can communicate and exchange information to manage the product line processes and solve challenging problems collaboratively such as resource allocation for production tasks. Robots and machines are the resources in these systems which are used by agents in completing tasks and achieving the overall systems' goals.

An automated manufacturing system usually consists of a set of cells, a material handling system connecting the cells, and service centers including material warehouse, tools room, and equipment repair. A cell can be either a machine, inspector, or a load/unload robot. Therefore, an automated manufacturing system can also be defined as a set of machines in which parts are automatically transported from one machine to another for processing. However due to the lack of central control, allocating resources (i.e. machines and robots) in such systems are prone to emergent behavior.

In this research, we consider a multi-agent manufacturing system which consists of several interacting agents; one of which is the controller agent. The agents are responsible for the production tasks in the system. Each agent can play different roles. The machines and robots are shared among the agents. The controller agent assigns tasks to each agent. Once the tasks are assigned to an agent, that agent would be responsible for their completion. All the communications regarding those tasks (e.g. transportation requests) are initiated by the agent. While the tasks are being completed, the agent makes transportation requests to the controller agent and the controller responds accordingly. Once the transportation is complete, a message is sent to the agent informing that the parts are located on the requested machine. Once the tasks are done, the agent sends a message to the controller agent and informs it about the task completion.

This system is analyzed and designed using MaSE methodology. Among the MaSE models produced, the sequence diagrams in the "Applying Use Cases" step from the analysis phase of MaSE along with the "Agent Class Diagrams" from the design phase of this methodology are used to construct the partial scenarios for the system in this research. This is since the role sequence diagrams of the "Applying Use Cases" step of MaSE contain the conversations among roles assigned to each agent [16]. The agent class diagrams of MaSE on the other hand represent the complete agent system organization consisting of agent classes and the high-level relationships among them. An agent class is a template for a type of agent with the system roles it plays. Multiple assignments of roles to an agent demonstrate the ability of the agent to play assigned roles concurrently or sequentially.

The agent class diagram in MaSE is similar to the class diagrams used in object oriented design but the difference is that the agent classes are defined by roles, not by attributes and operations. Furthermore, relationships are the conversations among agents [62]. Figures 5.1-5.3 demonstrate MaSE role sequence diagrams for the multi-agent manufacturing system while Figure 5.4 shows the agent class diagram.

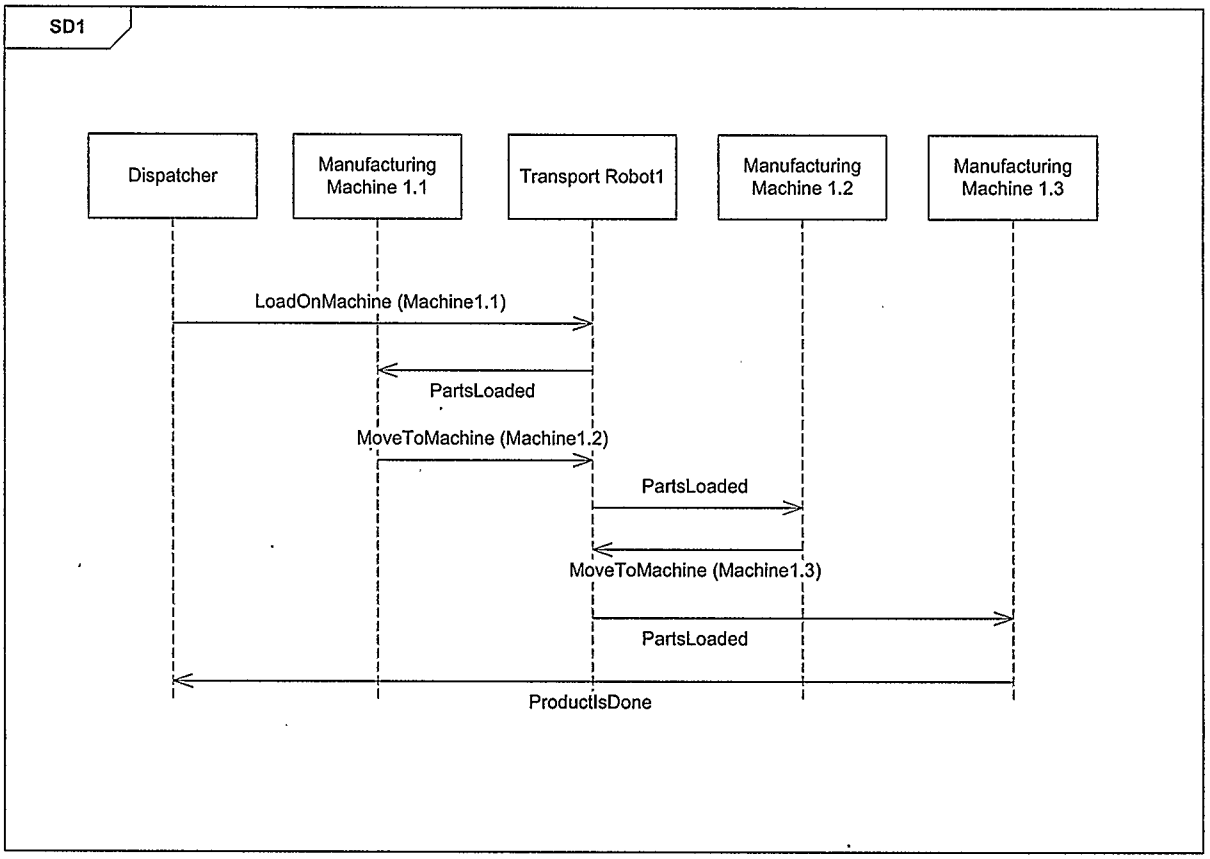


Figure 5.1 - Roles within Agent 1

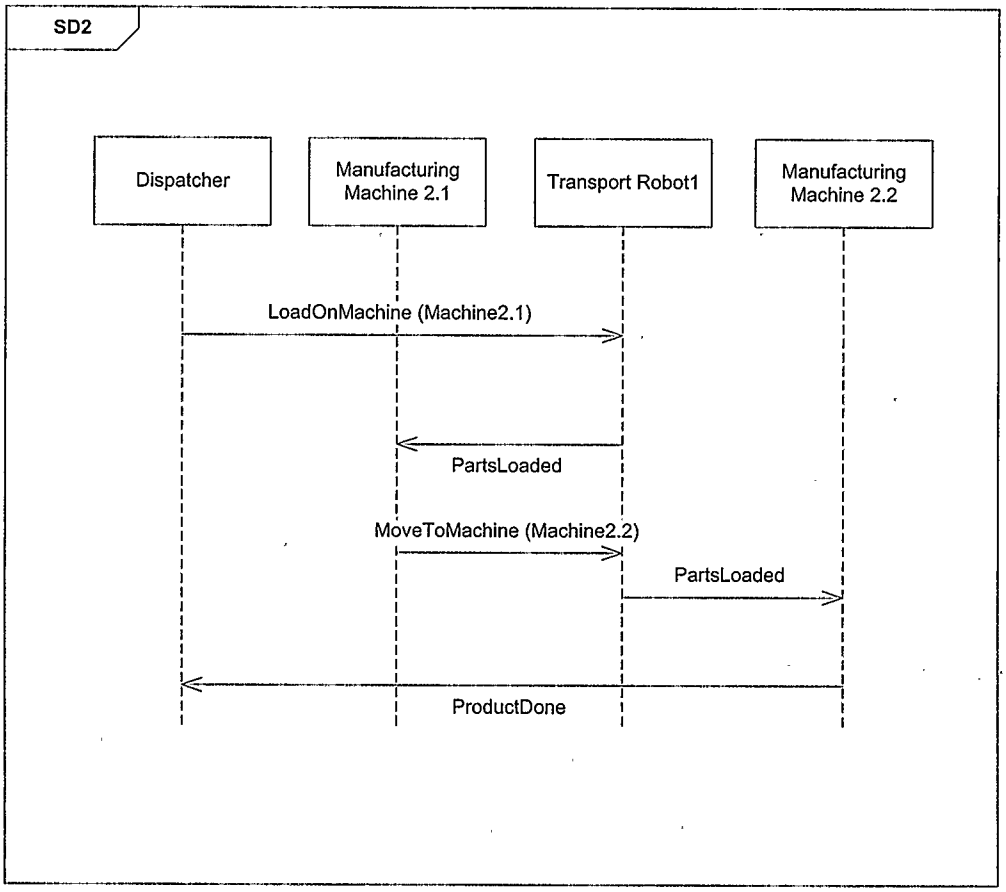


Figure 5.2 - Roles within Agent 2

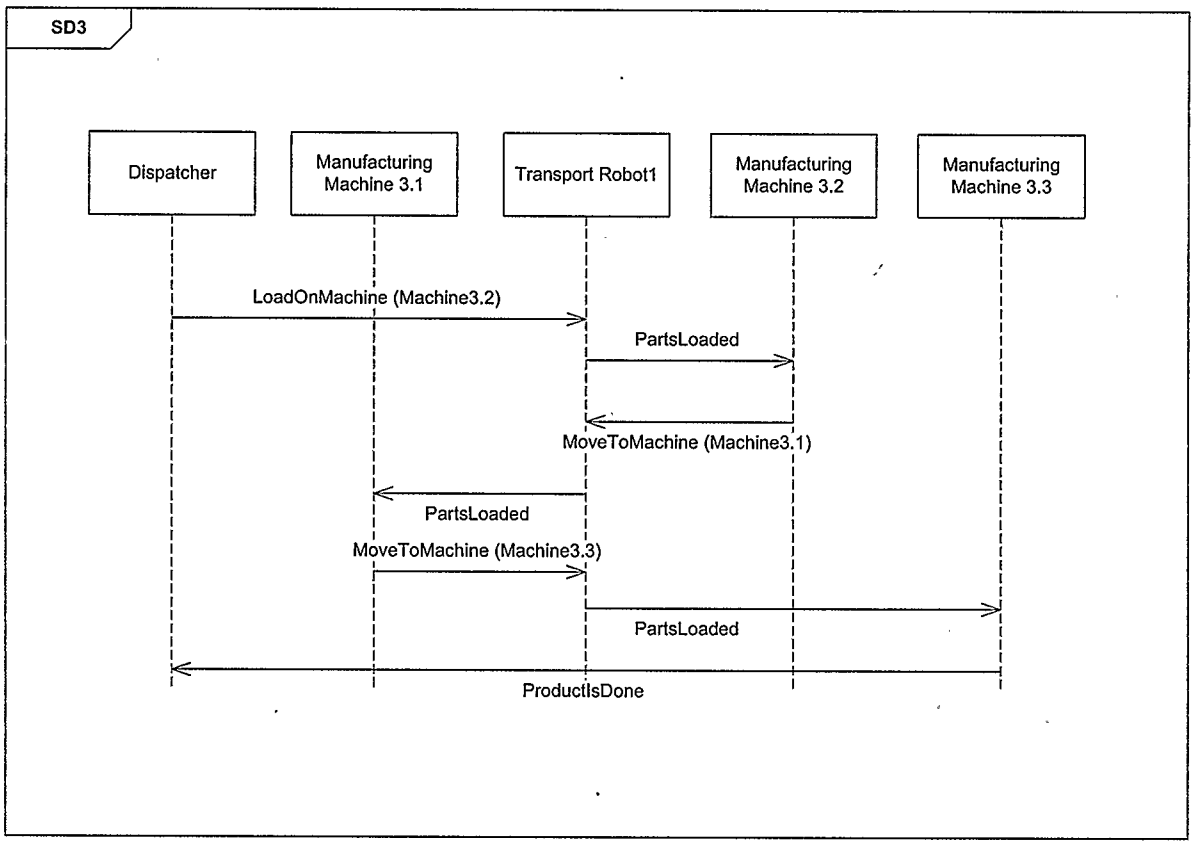


Figure 5.3 - Roles within Agent 3

The role sequence diagrams illustrated in Figures 5.1-5.3 illustrate the roles in agents 1-3 respectively, along with the conversations among these roles. The agent class diagram on the other hand demonstrates the communications of the agents with the controller agent as shown in Figure 5.4. For the sake of the simplicity of this case

study, the conversations among Agents 1-3 are omitted from the agent class diagram.

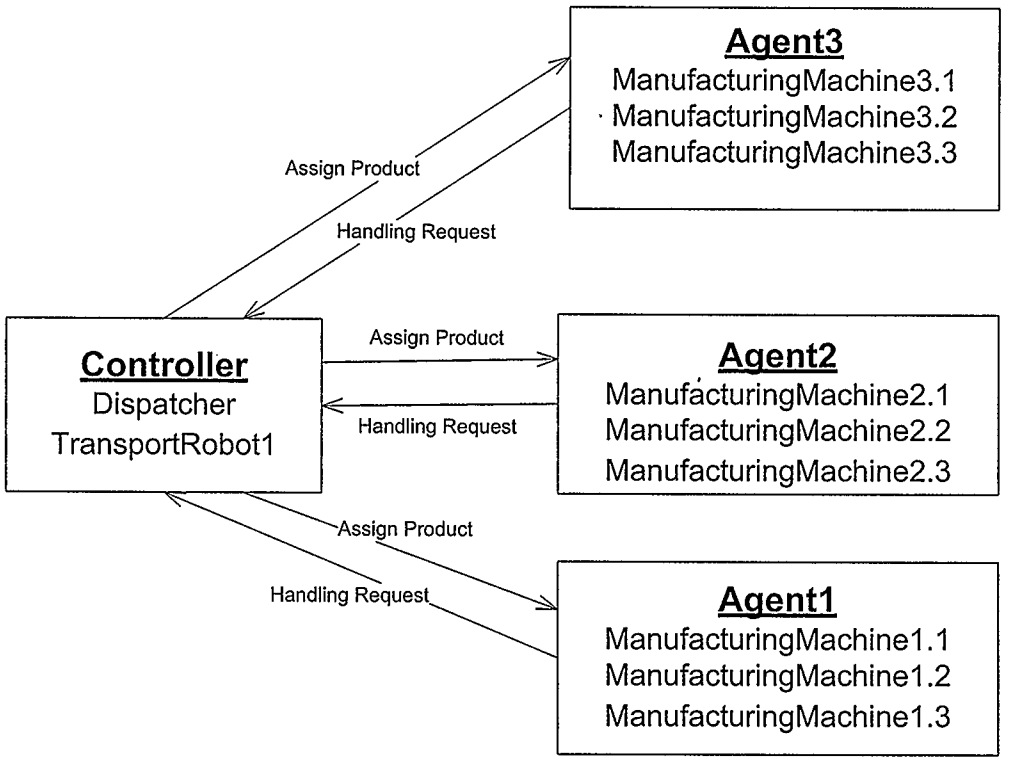


Figure 5.4 - Agent class diagram

The approach for extracting message sequence charts (MSCs) from the two above mentioned MaSE models is defined as follows: Each role sequence diagram is searched for the roles which are listed in the same agent class shown in the agent class diagram (Figure 5.4). Following this, all of the roles in each role sequence diagram are categorized based on their agent. Thus each category corresponds to an agent class of the agent class diagram and the messages which it exchanges with other categories are recognizable. From these two models an MSC can be generated which would display the

recognized messages between each two categories. Figure 5.5 demonstrates the extracted MSCs from the MaSE artefacts shown in Figures 5.1-5.4.

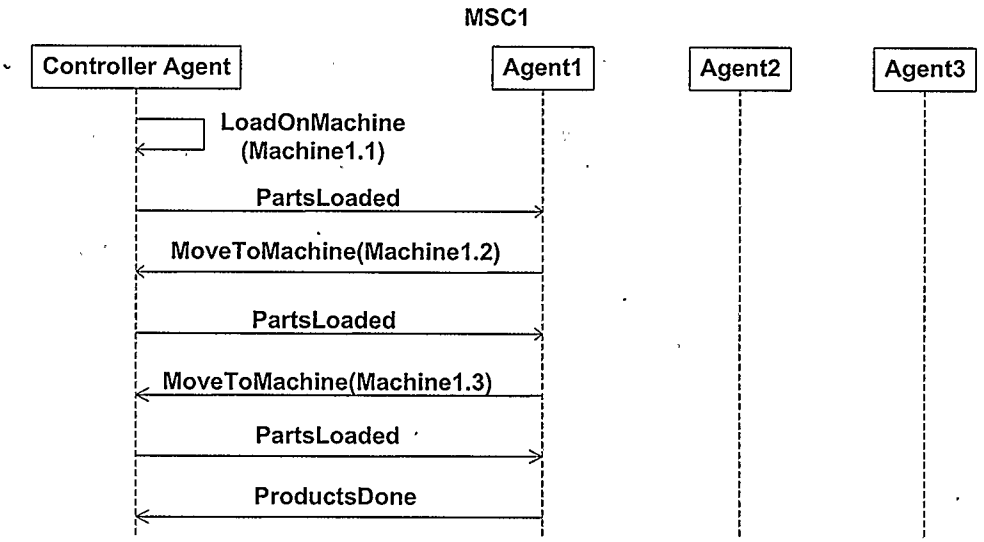


Figure 5.5 - Extracted MSC from MaSE models

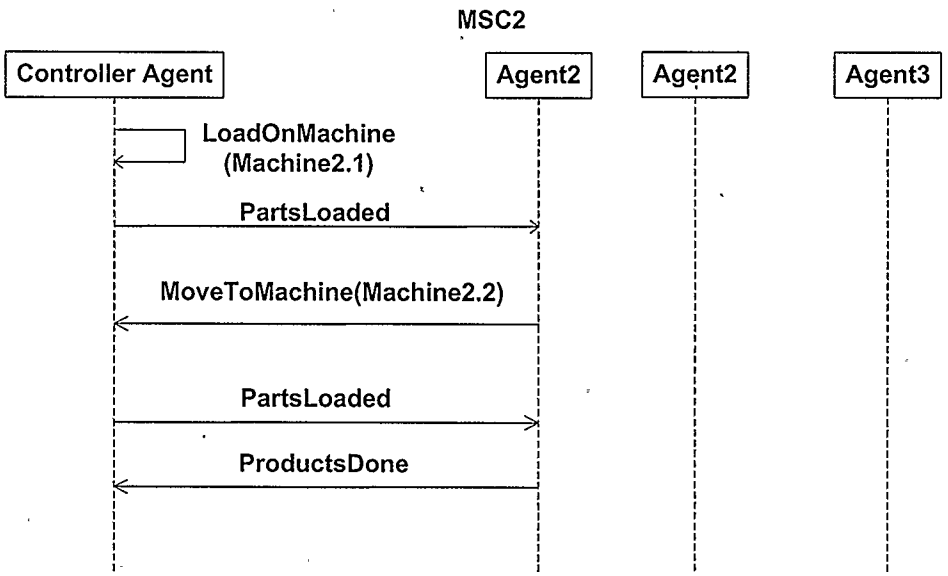


Figure 5.6 - Extracted MSC from MaSE models

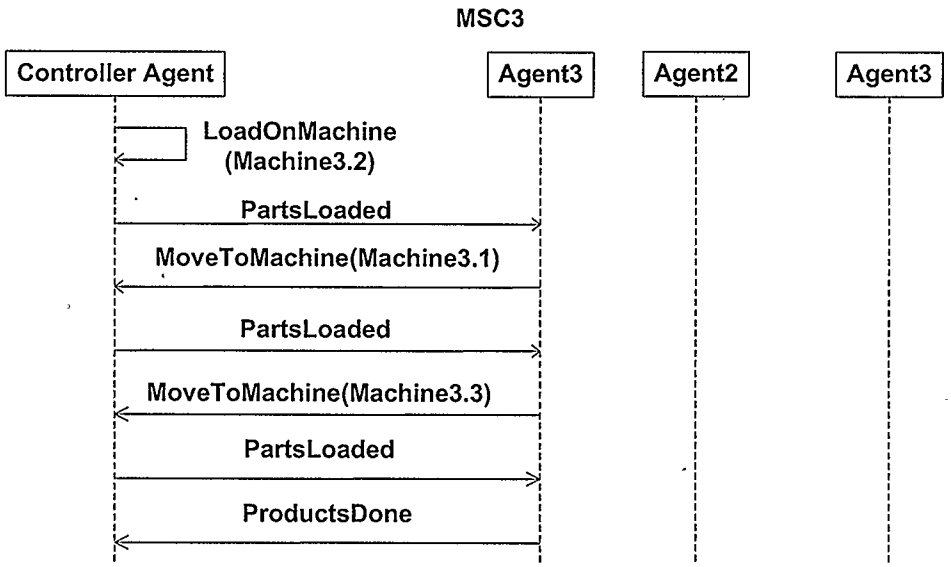


Figure 5.7 - Extracted MSC from MaSE models

5.3 MAS Behavior Modeling

By extracting the partial scenarios of the system based on the MaSE models, we proceed to the modeling of the system's behavior. As mentioned previously, developing a methodology which can systematically discover and remove system design faults prior to the implementation phase results in huge savings in cost and time. The first step of the methodology proposed by this research is the synthesis of state machines from MSCs. This step was demonstrated in Chapter 3 for the distributed system of a mine sweeping robot and is followed in this section using the case study of the manufacturing MAS.

We start by constructing behavior models for individual agents using finite state machines (FSMs). The process of building FSMs from message sequence charts (MSCs) is generally referred to as behavior modeling. For any agent i of a partial message sequence chart (pMSC) defined in Definition 1, an equivalent state machine (Definition

3) can be constructed. For instance, in the case of the manufacturing MAS the behavior model of the "Controller Agent" is demonstrated. Figures 5.8, 5.9 and 5.10 show the eFSMs built for the "Controller Agent" in MSCs 1, 2 and 3 respectively (Figures 5.5-5.7).

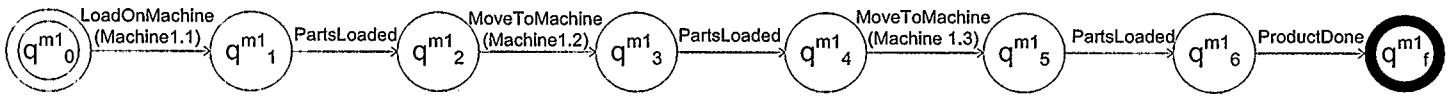


Figure 5.8 - eFSM for the controller agent in MSC1 of Figure 5.5

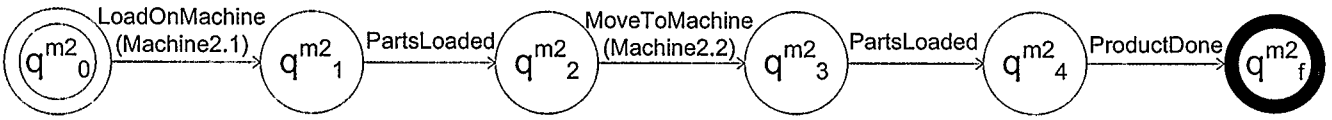


Figure 5.9 - eFSM for the controller agent in MSC2 of Figure 5.6

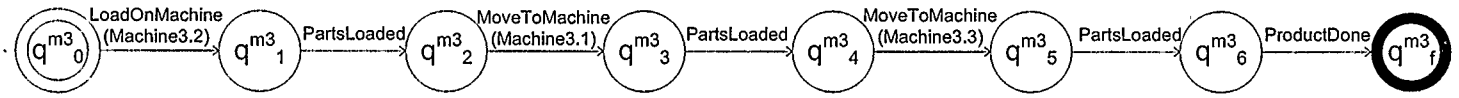


Figure 5.10 - eFSM for the controller agent in MSC3 of Figure 5.7

To complete the behavior modeling for the Controller Agent, the union of eFSMs built for each eFSM from MSCs 1-3 of Figures 5.5-5.7 is constructed as shown in Figure 5.11.

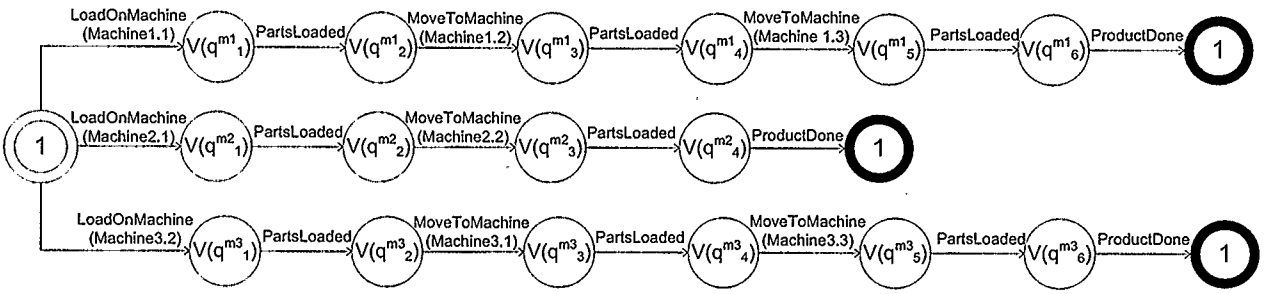


Figure 5.11 - The union of eFSMs built from MSCs 1-3

5.4 Detection of Emergent Behavior in MAS

In this section detection of emergent behavior in MAS is illustrated using the manufacturing case study. Since this method is quite similar to the one used for distributed systems, this section is kept very brief. We start by calculating state values. For instance in order to calculate the state value for state q_4^{m1} we proceed as follows: from the domain theory of the system (Definition 5) we learn that the case ii of Definition 6 applies. Therefore the value of state q_4^{m1} is calculated as $v_{Controller\ Agent}|(q_4^{m1}) = m\,|_{Controller\ Agent}\,[4-1]$ = PartsLoaded. Continuing the same approach the values of states q_4^{m2} and q_4^{m3} are also calculated to be "PartsLoaded".

By considering the resulting FSM in Figure 5.11, we select pairs of states with the same incoming transitions and evaluate their state to look for identical states. Figure 5.12 illustrates the constructed FSM as the result of merging two of the discovered identical states. Thus as shown in Figure 5.12, it is discovered that emergent behavior exists in S4. In this state, the controller agent becomes confused as to whether the right action is to move to machine 1.3, or whether to conclude its work.

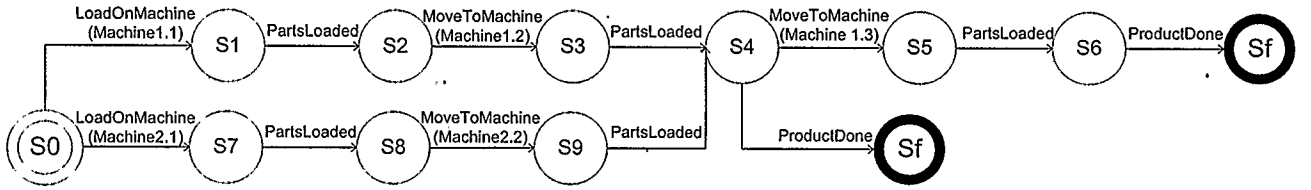


Figure 5.12 - Resulted FSM after merging identical states

5.5 Summary

This chapter aims to detecting emergent behavior in the requirements of multi-agent systems (MAS) using systematic and automated methodologies. This is accomplished by formulating a link between MaSE, which is one of the prime AOSE methodologies, and scenario-based software engineering (SBSE). This endeavour is commenced by converting MaSE artefacts to scenario-based specifications, represented by message sequence charts (MSCs). These specifications are then used to construct behavior models for all agents involved in the MAS. Finally the behavior models are analyzed for validating the design of MAS and ensuring the lack of emergent behavior.

In Chapter 6 the requirement and design documents, as well as the prototype for the design validation tool which automates the methodologies outlined in this chapter and Chapter 4 are presented.

Chapter Six: Design and Implementation of an Automated Tool

As presented in the preceding sections of this thesis, despite many advantages of using scenario-based specification for the design of distributed systems, there are certain limitations to this approach. Therefore in order to use scenarios, it is greatly beneficial and even necessary to devise methodologies which verify the resulted design of the system. Consequently in order to enable the efficient and effective use of these methodologies in real world projects, they need to be made automated in a user-friendly software package. In this section, the design and implementation of this tool is presented.

6.1 System Requirements

The functional and non-functional requirements of this tool are outlined in this section.

6.1.1 Functional Requirements

F1: Import Message Sequence Charts (Evident)

The user must be able to import one or more (1-200) message sequence charts from a third party software such as Eclipse, Microsoft Visio or IBM Rational Rose (Assuming only Eclipse in preliminary requirements).

F2: Parse Imported MSCs (Hidden)

The MSCs that the user has imported are in XML. The system must parse them and extract relevant information from those files.

F3: Synthesis Behavior Models (Hidden)

MSCs are to be converted into finite state machines. This is to be done as follows: For each component in each MSC, a finite state machine is to be defined in such

a way that messages to and from that component are the transitions between states. To complete the synthesis step for each component, union all the resulting FSMs from all the different scenarios which contain that component.

F4: Converting the resulting NFAs to DFAs (Hidden)

As the resulting FSMs from F3 are unions of other FSMs, they are by definition non-deterministic finite state machines (NFA). These NFAs must be made into DFAs and simplified.

F5: Building the Domain Theory (Evident)

In order to determine the identical states (which is the ultimate goal of the tool) the domain expert must specify certain architectural information known as semantic causality between pairs of messages. Semantic causality is formally defined in Definition 4 of Chapter 3.

This information will be known as the "domain theory" and will assist the system to determine the state values of the resulted FSMs. The formal definition for the domain theory has been given in Chapter 3 as Definition 5.

However it is obvious that prior to selecting pairs of messages between which semantic causality exits, state values of the FSM must be calculated. The state values in FSMs are provided in Definition 6 in Chapter 3.

Building a full domain theory would mean having the domain expert establish every semantic causality relationship for each component in the system. Therefore a light domain theory must be made as follows:

For each component, consider the resulting FSM (DFA) from F4. For every pair of states that have the same incoming transitions, seek user input and calculate state values.

F6: Detection of Emergent Behavior (Hidden)

Based on the domain theory constructed in F5, identical states are to be identified according to Definition 7 given in Chapter 3.

F7: Producing Report(Evident)

Using the results from F6, a report is generated for the system engineer to inform them as to whether or not emergent behavior exists in the system.

6.1.2 Non-Functional Requirements

The system's non-functional requirements are defined in the following categories:

Usability

The system is being developed to be used mostly by software engineers and designers. It is envisioned to be used to design a vast range of software systems such as distributed systems and multi-agent systems. Therefore it is of vital importance for this tool to be versatile and easy to use. To achieve this, an easy to use and user-friendly graphical user interface (GUI) will be designed and implemented. Furthermore the steps the software will take to achieve its goals will be well thought of, so that they follow a logical flow. To assess whether or not this requirement has been realized, several users will be selected to use this tool. Their feedback will be recorded to evaluate the usability of this tool.

Reliability

This tool is planned to be a desktop application, therefore reliability becomes a much simpler concept to deal with. However this tool is to be depended upon by engineers and software designers, and it must be made reliable. This tool will be designed in such a way that data is not lost if the program is to terminate. Rigorous testing must be done throughout to ensure the software is developed correctly. Both black-box and white-box testing must be done to remove faults. Software reliability engineering techniques must be employed to asses and ensure reliability of the software.

Performance

This tool will potentially have to process a great amount of data in one run. Numerous scenarios can be chosen to be checked using this tool at one time. The core algorithms have already been designed and have been optimized to match this requirement. It is set as a limit that the response time for any number of input scenarios should not exceed 10 minutes. Since the maximum number of input scenarios as specified in the functional requirements (F1) is 200, the performance of the tool can simply be measured against time.

Supportability

This system is to be designed to be modular and by extension maintainable. There are a number of different algorithms that will be implemented and added to this tool. Therefore the design of this tool is to be in such a way that updating and correcting algorithms and adding new ones are made simple. To asses the achievement of this requirement the modularity of the architecture of this tool can be analyzed.

6.2 Design Documents

High level design documents of this tool have been provided in this section.

6.2.1 List of Actors

There are two types of actors in this system as explained in this section:

Domain Expert (System's user)

The domain expert is the human user of the system. This user is typically the requirement engineer or the designer of the software. To start with, the domain expert chooses a set of message sequence charts as input for the system. Then as the system is running, the domain expert is asked to help with building the "domain theory". That is, the domain expert will be asked to provide input related to the architecture of the system to be built. Upon completion, the domain expert is presented with the output.

Eclipse

Message sequence charts (MSCs) are produced using Eclipse. They are then imported by our system to start analysis.

NOTE: As the software is developed, the goal is to have the tool connect to other tools such as IBM's Rational Rose and Microsoft Visio.

6.2.2 System's Use Cases

The names and brief descriptions for the systems use cases are provided in this section.

Import XML Files

This use case is responsible for importing MSCs from Eclipse, parsing them and extracting relevant information from those files.

Construct Finite State Machines (FSM)

As defined in functional requirements (F3), each component in the imported MSCs is to be converted to finite state machines.

Synthesize Behavior Model

As required by the functional requirements (F3), the resulted FSMs for each component are to be unionized. This is done to obtain the full behavior of the component in the system. As defined by F4, the resulting FSM is simplified to obtain a deterministic finite automaton (DFA).

Build Domain Theory

As required by F5 the architectural properties of the system are defined in this use case. All requirements stated by F5 are realized here.

Detect Identical States

As required by F6, based on the domain theory constructed, identical states are to be identified.

Produce Report

This use case is responsible for making a report based on the result of the analysis.

The use case diagram of the system is presented in Figure 6.1.

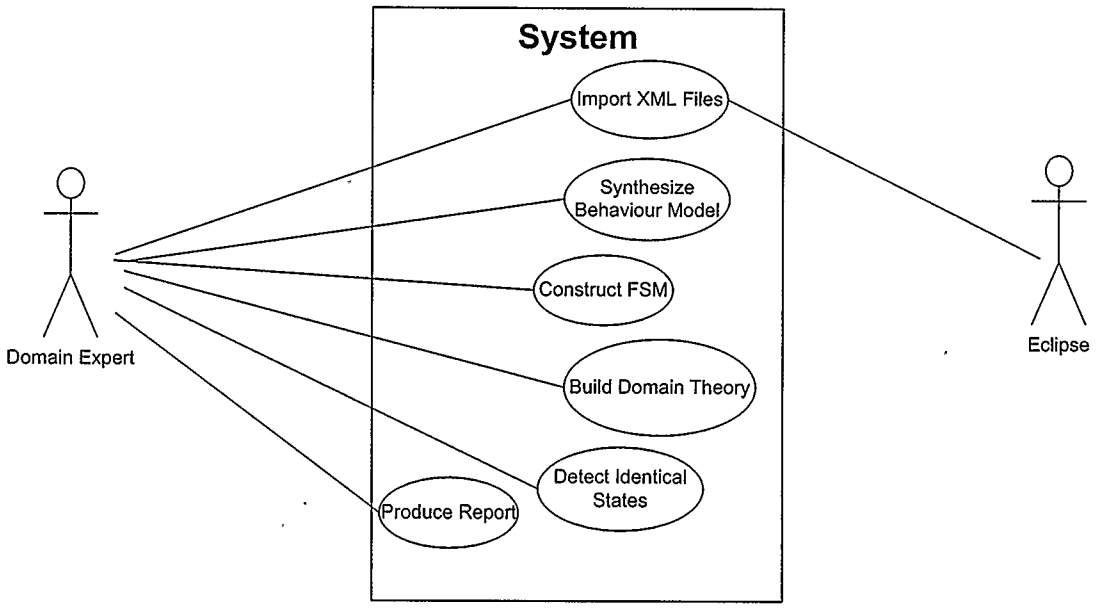


Figure 6.1 - Use case diagram

6.2.3 Flow of Events

The flow of events is done for the following two use cases:

Synthesis of Behavior Model

- Each component of each scenario is converted to a finite state machine as stated in F3
- The resulting state machines of each components from different scenarios are unionized together
- If the resulting FSM is non-deterministic, convert it to a deterministic finite state machine (i.e. DFA)
- Simplify the resulting DFA (remove unnecessary states)

Build Domain Theory

- For a given FSM, search for pairs of states which have the same incoming transitions

- For each of the states obtained in the previous step, obtain its semantic cause
- Using semantic causality, calculate the state's value (Definition 4)

The activity diagrams for the above two use cases are illustrated in Figures 6.2-6.3.

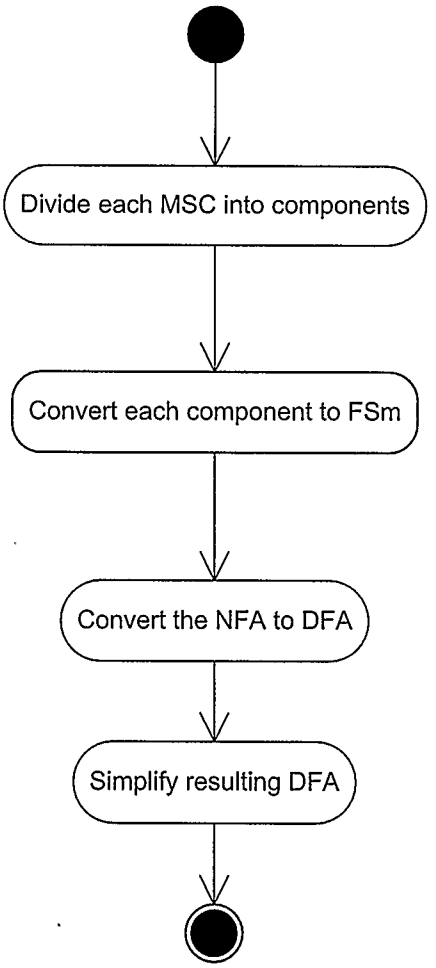


Figure 6.2 - Activity diagrams for the Synthesis Behavior Model

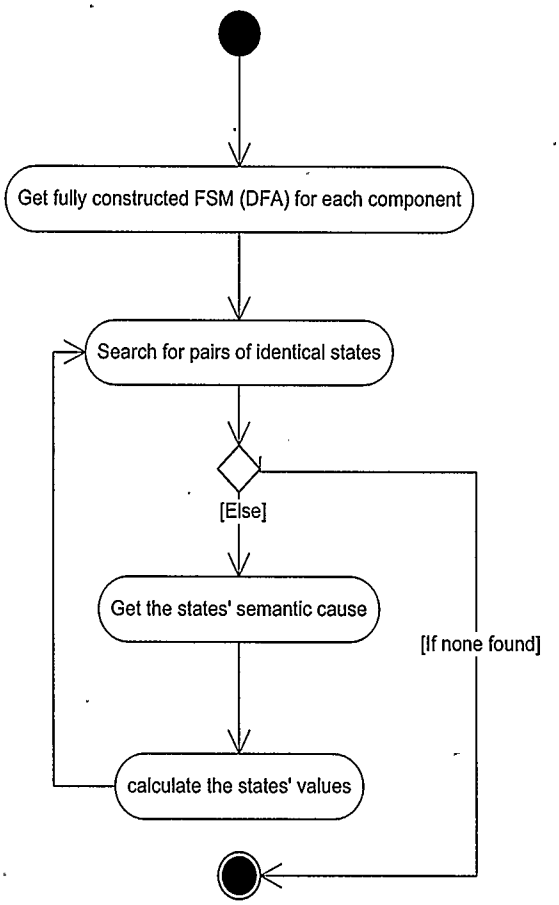


Figure 6.3 - Activity diagram for Build Domain Theory

Domain Class Diagram

The domain class diagram according to the system requirements is presented below:

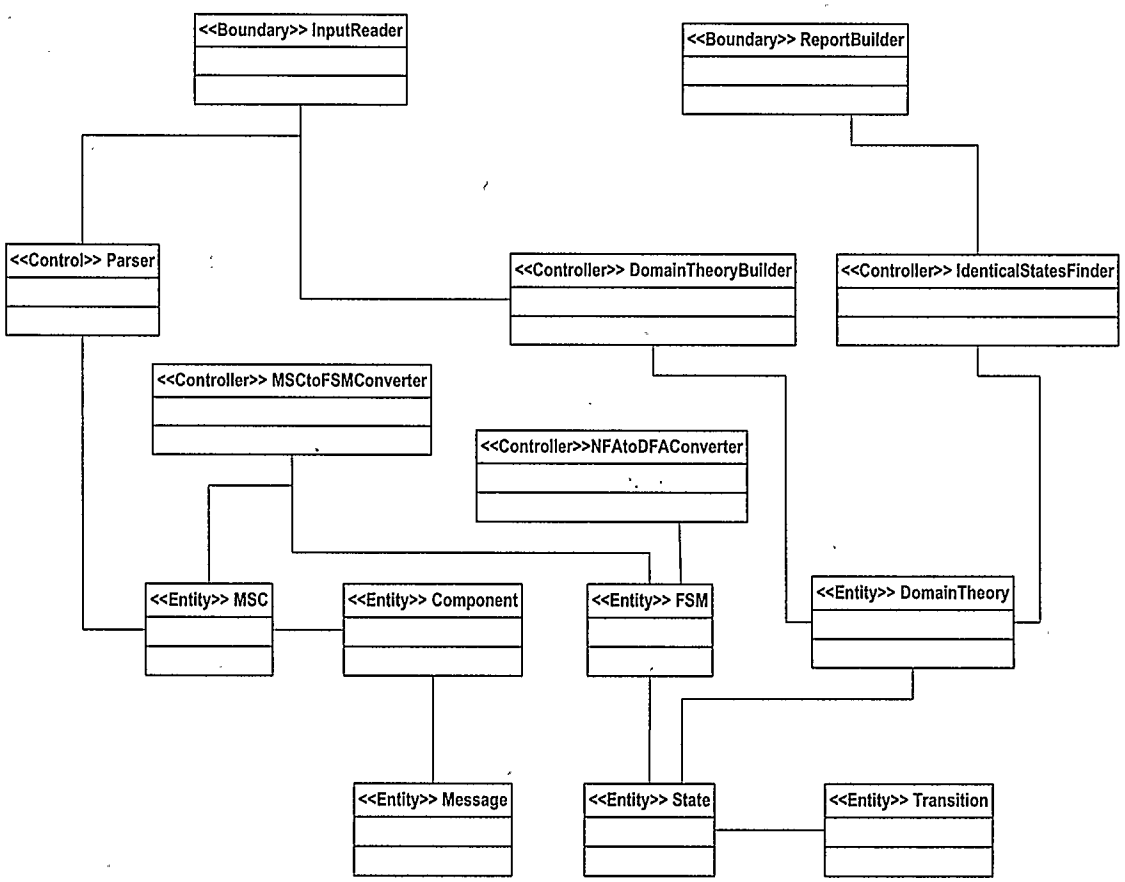


Figure 6.4 - Domain class diagram

6.3 System prototype

The prototype of the design validation tool is presented in this section [63]. There were two major concerns in the development of this tool. First, as there is still a vast amount of research remaining in this area, it is essential for this tool to be modular and scalable. To comply with this requirement, this tool was built on the two pillars of encapsulation and parallel execution.

The second concern was that since this tool is developed to increase the efficiency of the development life cycle, it is highly desirable that it is easy to use. To incorporate the usability of the tool, an easy to follow graphical user interface (GUI) has been developed (Figure 6.5) which closely represents the logical flow of the developed methodology. To test the usability of this tool, as mentioned in the non-functional requirements section of this chapter, several users will be selected to utilize this tool. Their feedback will be recorded to evaluate the usability of this tool.

Moreover, as MSCs are constructed using a variety of different software packages, in order to account for the convenience of the users, this tool can import MSCs from a variety of different tools such as IBM Rational Rose and Microsoft Office Visio.

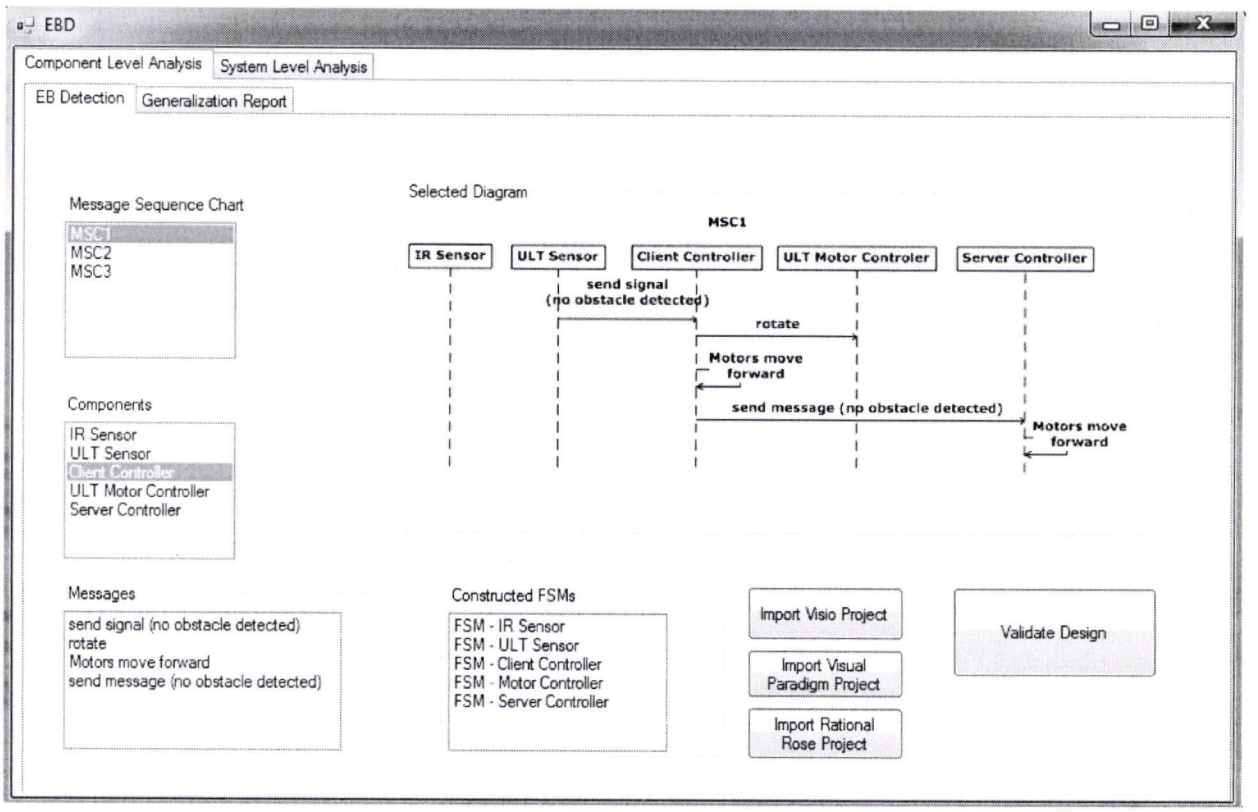


Figure 6.5 - Snapshot of the GUI of the tool; displaying an imported MSC

As it can be seen from the snapshot of the tool's graphical user interface shown in Figure 6.5, upon importing a design project from one of the above mentioned tools, the data boxes of the GUI are populated automatically with appropriate data. By clicking any of the imported MSCs, the components of that MSC will be shown in the *Component* subsection of the GUI and the actual MSC will be shown in the *Selected Diagram* area. Consequently, by selecting a component, the messages associated with that component will be shown in the *Message* subsection of the GUI. The synthesis of the behavior model, explained in Sections 3.2 and 4.3 of this thesis is conducted immediately upon importing related MSCs. As the result the built FSMs will be shown in the *Constructed*

FSMs subsection of the GUI. By clicking on the title of any of the FSMs the constructed figure will be shown in the Selected Diagram area as shown in Figure 6.6.

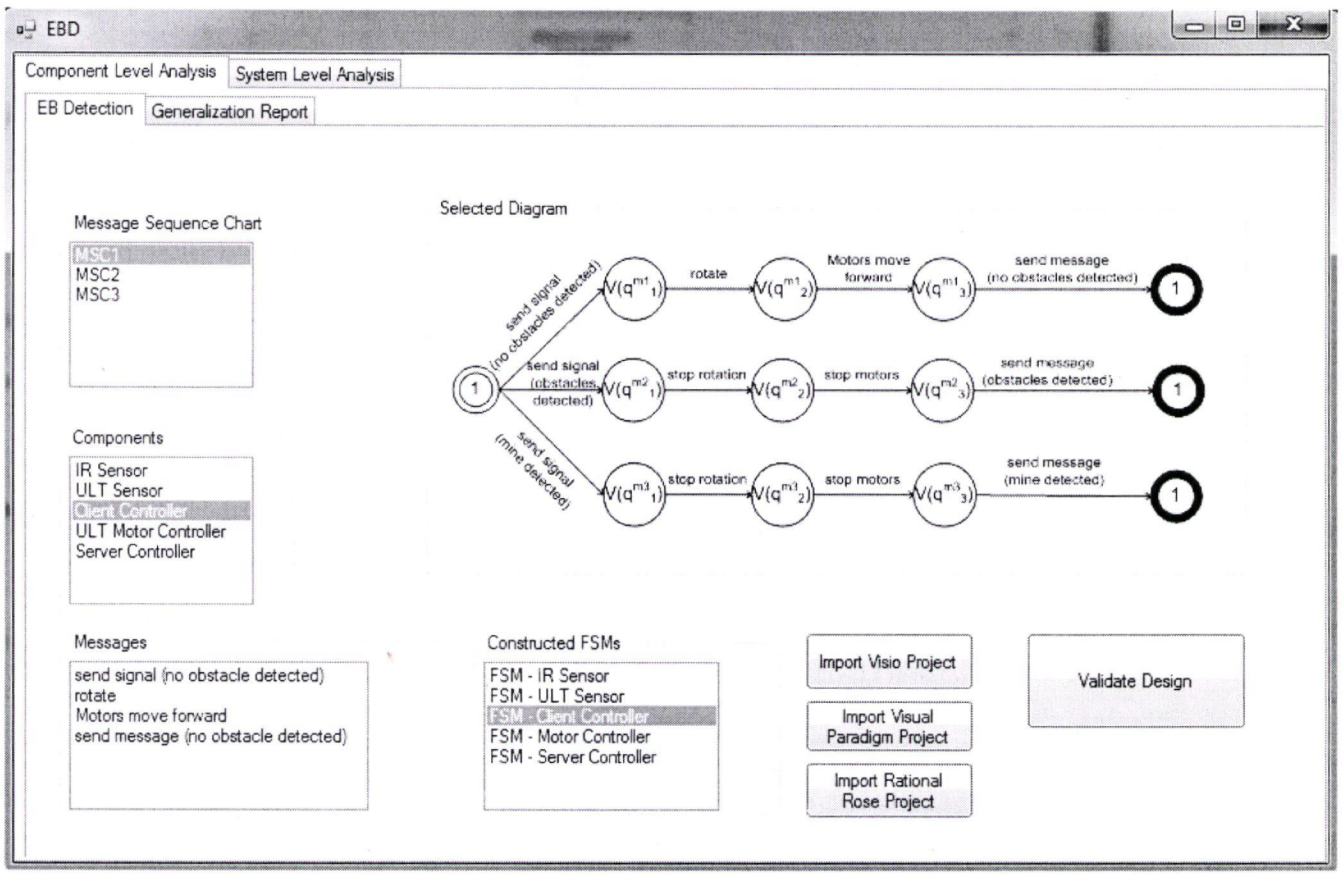


Figure 6.6 - Snapshot of the GUI of the tool; displaying a constructed FSM

At this point, by clicking the Validate Design button, the methodology commences. The user will be asked to assist in constructing the domain theory which will be used to find identical states as outlined in sections 3.3 and 4.4 of this thesis. Upon completion of the analysis, the user will be presented with a report outlining the areas in which indeterminism could occur.

6.4 Summary

Manual review of requirements and design documents, particularly for large-scale software systems such as MAS and distributed systems is somehow inefficient. This research attempts to devise methodologies to analyze software design and requirements using a systematic approach as outlined in this thesis. This chapter presents the steps towards automating the proposed methodologies as a software tool.

The requirements and design documents of the design validation software tool, as well as its preliminary prototype are presented in this chapter. Using this tool, system engineers can analyze system requirements in an efficient and effective manner.

Chapter Seven: Conclusions and Future Work

Scenario-based specifications are an effective and efficient way to describe the requirements of a variety of software systems such as multi-agent systems (MAS) and distributed systems. Scenarios enable engineers and designers to describe system functionality using the partial interactions of the system elements. Moreover, due to their simplicity and expressive power, scenarios are the perfect medium through which all stakeholders can communicate in an efficient and effective manner.

However scenario-based specifications are prone to deficiencies such as incompleteness and contradictions. It is of vital importance that these deficiencies are identified prior to implementation as many system failures can be attributed to faulty requirements and design of software. Studies suggest that the discovery and elimination of faults and failures during field use of a system is estimated to be about 20 times more expensive than detection and removal of faults in the requirement and design phase [64].

The main goal of this research is to identify possible design flaws that might lead to run time problems in software systems by analyzing the system specification expressed by scenarios. Unfortunately, manual review may not efficiently detect all of the design flaws due to the scale and complexity of the systems. In this research we have provided sound techniques to automate the specification and design review of the software systems and detect a subset of unwanted run time behaviors, including implied scenarios.

Furthermore, in this thesis a method to identify the exact cause of implied scenarios is provided, so that by capturing it, implied scenarios can be detected and removed. This method is novel in the sense of formalization of the cause of implied

scenarios. We believe that this is the main reason for some shortcomings and conflicts in the current works, as they have been revealed in [4, 26, 65].

The proposed methodologies in this thesis were applied to a variety of different software systems such as distributed and multi-agent systems. In Chapter 4, two approaches for the validation of software requirements for distributed systems were introduced. The first approach which was illustrated using the case study for a mine-sweeping robot involves compiling all system scenarios and conducting behavioral modeling in order to discover all cases of emergent behavior in system's requirements. The second approach which was illustrated using an online commerce application involves ensuring the lack of existence of particular scenarios in the system behavior. Furthermore the applicability of scenario-based software engineering (SEBE) and the emergent behavior detection methodologies in agile development was addressed in this chapter. Incorporating requirement and design validation methodologies in agile development goes a long way towards quantifying agile approaches and can be particularly useful in building large scale and complex software such as distributed or multi-agent systems.

In Chapter 5 a comprehensive methodology was introduced to analyse AOSE design documents to ensure validation of MAS requirements and the prototype of the design validation software was presented in Chapter 6. This software provides an easy to use and practical tool to apply these algorithms to requirements and design documents.

For future work, the proposed methodologies can be extended to a comprehensive framework for model based analysis and testing of distributed and multi-agent software

systems. Furthermore, this technique can be modified to take the UML's sequence diagrams as input and thus incorporate analyzing object oriented design. These techniques can also be extended to systematically analyze the design of multi-agent systems as well as social networks. In this research plug-ins to convert MaSE artefacts to MSCs were developed. For future work additional plug-ins can be developed to convert modeling constructs of other AOSE methodologies to scenario-based specifications (Figure 1.1).

As the produced software closely follows the principle of encapsulations and modularization, it is intended to add the future results of this research to this tool upon completion. In addition, since this tool already has the capabilities to import projects from IBM's Rational Rose, it can be modified and implemented as a plug-in for it. Figure 1.1 illustrates the general structure of the comprehensive framework to be built. This framework will take input from a variety of different models and its outputs are component-level emergent behavior detection (CEBD), system-level emergent behavior detection (SEBD) and model based detection and testing of MAS (MDTM).

Moreover as the analysis of the requirements of software systems were conducted at the component level in this research, research can be done in analyzing at the system level. In system level analysis it is assumed that the emergent behavior for the components has already been resolved. Here scenarios are further analyzed for detecting possible system level implied scenarios [3]. Contributions regarding this task involve: (1) revisiting the notion of safe realizability for MSC specifications [3] in order to make it computationally implementable; (2) devising an algorithm for "strong safe

realizability" (i.e. an implementation that covers the behavior described by the specification while avoiding "stuck states" [3]); and (3) devising an algorithm for detecting implied scenarios. The term stuck states (i.e. a super set for deadlock) is new and it means that a message sent has no receiver to catch it, and a receiver waits for a message that has never been sent.

The efficiency of the proposed methodologies is highly dependent on building the domain theory for the system. Future work can be done to add efficiency to the building, maintenance and using of the domain theory. Data mining techniques and ontology can be employed to improve the current state of domain theory in the proposed methodologies.

References

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

[2] J. Whittle and J. Schumann, "Scenario-Based Engineering of Multi-Agent Systems," in *Agent Technology from a Formal Perspective*, Third ed London: Springer-Verlag, 2006.

[3] A. Mousavi, "Inference of Emergent Behaviours of Scenario-Based Specifications," in *Department of Electrial and Computer Engineering*. vol. PhD Calgary: University of Calgary, 2009.

[4] A. J. Mooij, N. Goga, and J. M. T. Romijn, "Non-local choice and beyond: Intericacies of MSC choice nodes," in *M. Cerioli (ed): FASE 2005, LNCS 3442*: Springer, 2005, pp. 273-288.

[5] J. Whittle and J. Schumann, "Generating statecharts designs from scenarios," in *ICSE* Limerick, Ireland, 2000.

[6] D. Harel and H. Kugler, "Synthesizing state-based object systems from lsc specifications," *International Journal of Foundations of Computer Science,* 2002.

[7] I. Kruger, R. Grosu, P. Scholz, and M. Broy, "From mscs to statecharts," in *Franz j. rammig (ed.): Distributed and parallel embedded systems*: Kluwer Academic Publis, 1999.

[8] E. Makinen and T. Systa, "MAS - an interactive synthesizer to support behavioral modeling in UML," in *ICSE 2001* Toronto, 2001.

[9] S. Uchitel, J. Kramer, and J. Magee, "Negative scenarios for implied scenario elicitation," in *10th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE 2002)* Charleston.

[10] S. Uchitel, J. Kramer, and J. Magee, "Synthesis of behavioral models from scenarios," *IEEE Transaction on Software Engineering,* pp. 99-115, February 2003.

[11] B. Adsul, M. Mukund, K. N. Kumar, and V. Narayanan, "Casual Closure for MSC Languages " in *FSTTCS*, 2005, pp. 335-347.

[12] R. Alur, K. Etessami, and M. Yannakakis, "Inference of Message Sequence Charts," *IEEE Transaction on Software Engineering,* pp. 623-633, July 2003.

[13] H. Muccini, "Detecting implied scenarios analyzing nonlocal branching choices," in *FASE 2003* Warsaw, Poland.

[14] G. J. Holzmann, D. Peled, and M. H. Redberg, "Design Tools for Requirement Engineering," *Bell Labs Technical Journal,* 1997.

[15] D. Harel, "From Play-in Scenarios to Code: An Achievable Dream," *IEEE Software,* vol. 34(1), pp. 53-60, 2001.

[16] J. Grabowski, "Test Generation and Test Case Specification with Message Sequence Charts," in *Institute for Informatics and Applied Mathematics*: Universitat Bern, 1994.

[17] J. M. Carroll, *Scenario-based Design: Envisioning Work and Technology in System Development.* New York: Wiley, 1995.

[18] I. Jacobson, J. Rumbaugh, and G. Booch, *The Unified Software Development Process*. Harlow: Addison-Wesley, 1999.

[19] T. Quatrani, *Visual Modeling with Rational Rose 2000 and UML*. Boston: Addison Wesley, 2004.

[20] P. P. Texel and C. B. Williams, *Use Cases Combined with Booch, OMT, and UML*: Prentice-Hall, 1997.

[21] K. Weidenhaupt, K. Pohl, M. Jarke, and P. Haumer, "Scenarios in System Development: Current Practice," *IEEE Software,* vol. 15(2), pp. 34–45, 1998.

[22] S. Robertson and J. Robertson, *Mastering the Requirements Process*. Harlow, England: ACM Press, 1999.

[23] "ITU: Message Sequence Charts. Recommendation, International Telecommunication Union. ," 1992.

[24] "Unified Modeling Language Specification. Version 2. Available from Rational Software Corporation," Cupertino, CA, 2006.

[25] G. R. Andrews, *Foundations of Multithreaded, Parallel, and Distributed Programming*: Addison–Wesley, 2000.

[26] M. Moshirpour, A. Mousavi, and B. Far, "Detecting Emergent Behavior in Distributed Systems Using Scenario-Based Specifications," in *Proceedings of the International Conference on Software Engineering and Knowledge Engineering* San Francisco Bay, USA, 2010.

[27] P. Smith, *Client/Server Computing*, Second ed.: Sams Publishing, 1994.

[28] S. H. Kaisler, *Software Paradigms*: John Wiley & Sons, Inc., 2005.

[29] S. Ruby, D. Thomas, and D. H. Hansson, *Agile Web Development with Rails*, Third ed.: The Pragmatic Programmers LLC, 2009.

[30] G. Weib, *Agent Orientation in Software Engineering*: Cambridge University Press, 2001.

[31] M. Wooldridge, *An Introduction to Multi-Agent Systems*, Second ed.: John Wiley & Sons, 2009.

[32] S. A. DeLoach, "The MaSE Methodology," in *Methodologies and Software Eng. for Agent System*, F. Bergenti, M.P.Gleizes, and F. Zambonelli, Eds. Boston: Kluwer Academic Publishers, 2004, pp. 107-147.

[33] S. A. DeLoach, "The MaSE Methodology," in *Methodologies and Software Eng. for Agent System.* vol. 11, F. Bergenti, M.P.Gleizes, and F. Zambonelli, Eds. New York: Kluwer Academic Publishers, 2004.

[34] G. M. Saba and E. Santos, "The Multi-Agent Distributed Goal Satisfaction System," in *ICSC Symposium on Multi-Agents and Mobile Agents in Virtual Organizations and E-Commerce*, 2000.

[35] J. T. McDonald, M. L. Talbert, and S. A. DeLoach, "Heterogeneous Database Integration Using Agent Oriented Information Systems," in *the International Conference on Artificial Intelligence*, 2000.

[36] P. K. Harmer and G. B. Lamont, "An Agent Architecture for a Computer Virus Immune System," in *Genetic and Evolutionary Computation Conference*, 2000.

[37] f. Bergenti, M. P. Gleizes, and F. Zambonelli, *Methodologies and Software Engineering for Agent System* vol. 11. New York: Kluwer Academic Publishers, 2004.

[38] A. H. Elamy and B. Far, "A Statistical Approach for Evaluating and Assembling Agent-Oriented Software Engineering Methodologies," in *Agent-Oriented Information Systems IV*. vol. 4898/2008 Berlin / Heidelberg: Springer, 2008, pp. 105-122.

[39] M. Wooldridge, N. R. Jennings, and D. Kinny, "The Gaia Methodology for Agent-Oriented Analysis and Design," in *Autonomous Agents and Multi-Agent Systems*. vol. 3, 2000, pp. 285-312.

[40] F. Giunchiglia, J. Mylopoulos, and A. Perini, "The tropos software development methodology: processes, models and diagrams," in *the first international joint conference on Autonomous agents and multiagent systems*, Italy, 2002, pp. 35-36.

[41] F. Bergenti, M.P.Gleizes, and F. Zambonelli, *Methodologies and Software Engineering for Agent System* vol. 11. New York: Kluwer Academic Publishers, 2004.

[42] F. Giunchiglia, J. Mylopoulos, and A. Perini, "The tropos software development methodology: processes, models and diagrams," in *Proceedings of the first international joint conference on Autonomous agents and multiagent systems*, Italy, 2002, pp. 35 - 36

[43] M. J. Wooldridge and P. Ciancarini, "Agent-Oriented Software Engineering: The State of the Art," in *Proc. of the Workshop on Agent-Oriented Soft. Eng.*, 2000, pp. 1-28.

[44] B. Meyer, "Applying Design by Contract," *IEEE Computer,* vol. 25, pp. 40–51, 1992.

[45] H. L. Timothy and S. A. DeLoach, "Automatic Verification of Multiagent Conversations," in *the Annual Midwest Artificial Intelligence and Cognitive Science* Fayetteville, 2000.

[46] M. J. Wooldridge, M. Fisher, M. Huget, and S. Parsons, "Model Checking Multi-Agent Systems with MABLE," in *Proc. of the Int. Joint Conf. on Autonomous Agents and Multiagent Systems*, 2002, pp. 952–959.

[47] G. J. Holzmann, "The Model Checker Spin," *IEEE Trans. on Soft. Eng.,* vol. 23, pp. 279–295, 1997.

[48] K. Koskimies, T. Mannisto, T. Systa, and J. Tuonmi, "Automated support for modeling oo software," *IEEE Software,* pp. 15(1):87–94, 1998.

[49] J. Warmer and A. Kleppe, *The Object Constraint Language: Precise Modeling with UML*: Addison-Wesley, 1999.

[50] "Recommendation Z.120: Message Sequence Chart(MSC)," Geneva, 1996.

[51] M. Lohrey, "Safe realizability of high-level message sequence charts," *CONCUR,* vol. 177-192, 2002.

[52] M. Moshirpour and B. H. Far, "Formal Verification of Lack of Existence of Illegal Scenarios in the Requirements of Distributed Systems," in *The IASTED*

International Conferences on Informatics 2010 Software Engineering and Applications (SEA 2010), Marina del Rey, USA, 2010.

[53] J. Hunt, *Agile Software Construction*. London: Springer-Verlag, 2006.

[54] E. Mnkandla and B. Dwolatzky, "Agile Software Methods: State-of-the-Art," in *Agile Software Development Quality Assurance*, I. Stamelos and P. Sfetsos, Eds. Hershey, London, Melbourne, Singapore: Information Science Reference, 2007.

[55] T. Gilb and L. Brodie, "What's Wrong with Agile Methods? Some Principles and Values to Encourage Quantification," in *Agile Software Development Quality Assurance*, I. G. Stamelos and P. Sfetsos, Eds. Hershey, London, Melbourne, Singapore: Information science reference, 2007.

[56] V. Monochristou and M. Vlachopoulou, "Requirements Specification using User Stories," in *Agile Software Development Quality Assurance*, I. G. Stamelos and P. Sfetsos, Eds. Hershey, London, Melbourne , Singapore: Information science reference, 2007.

[57] K. Beck, *Extereme Programming Explained: Embrace Change*. Reading, MA: Addison Wesley, 2000.

[58] M. Shehata, A. Eberlein, and A. Fapojuwo, "IRIS: A Semi-Formal Approach for Detecting Requirements Interactions," in *Proceedings of the 11th IEEE International Conference and Workshop on the Engineering of Computer-Based Systems (ECBS'04)*, 2004, pp. 273 - 281.

[59] M. Moshirpour, A. Mousavi, and B. Far, "Model Based Detection of Implied Scenarios in Multi Agent Systems," in *Proceedings of the International Conference on Information Reuse and Integration*, Las Vegas, USA, 2010.

[60] N. Mani, V. Garousi, and B. H. Far, "Search-based Testing of Multi-Agent Manufacturing Systems for Deadlock Detection Based on UML Models," *Special issue of the International Journal on Artificial Intelligence Tools (IJAIT),* vol. 19, pp. 417-437, 2010.

[61] N. Mani, V. Garousi, and B. H. Far, "A UML-Based Conversion Tool for Monitoring and Testing Multi-Agent Systems," in *20th IEEE International Conference on Tools with Artificial Intelligence*, Dayton, Ohio, USA, 2008, pp. 212-219.

[62] S. A. DeLoach, "The MaSE Methodology," in *Methodologies and Software Eng. for Agent System.* vol. 11, F. Bergenti, M.P.Gleizes, and F. Zambonelli, Eds. New York: Kluwer Academic Publishers, 2004, pp. 107-147.

[63] M. Moshirpour, A. Mousavi, and B. Far, "A Technique and Tool to Detect Emergent Behavior of Distributed Systems Using Scenario-Based Specifications," in *Proceedings of the International Conference on Tools with Artificial Intelligence*, Arras, France, 2010.

[64] D. R. Goldenson and D. L. Gibson, "Demonstrating the Impact and Benefits of CMMI: An Update and Preliminary Results," *CMU/SEI-2003-SR-009,* October 2003.

[65] A. Mousavi and B. Far, "Eliciting Scenarios from Scenarios," in *Proceedings of 20th International Conference on Software Engineering and Knowledge Engineering (SEKE 2008)* San Francisco Bay, USA: July 1-3, 2008, 2008.