UNIVERSITY OF CALGARY

Model-based Analysis of Software Requirements for Distributed Software Systems

by

Mohammad Moshirpour

A THESIS

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES
IN PARTIAL FULFILMENT OF THE REQUIREMENTS FOR THE
DEGREE OF DOCTOR OF PHILOSOPHY

GRADUATE PROGRAM IN ELECTRICAL AND COMPUTER ENGINEERING

CALGARY, ALBERTA

JANUARY, 2016

© Mohammad Moshirpour 2016
Abstract

Requirement elicitation is one of the most challenging stages of the software development lifecycle. Many bugs are introduced into the system as the result of incomplete or inconsistent requirements. Literature reports that detecting and removing unwanted behaviors during the design phase of a software system is about 20 times cheaper than finding them during the deployment phase. Moreover, planning the architecture and design of the system based on the requirements is a challenging task and is usually done using ad-hoc methodologies. However manual review of the system requirements is time consuming and inefficient, particularly for larger systems. Therefore, devising automated and systematic methodologies to analyze and verify software requirements is necessary.

Automating the process of software analysis is a challenging task because each software system requires its own domain knowledge. The existing approaches often require a great number of assumptions and inputs from the system engineer familiar with the domain. This research focuses on automating the process of modeling, analyzing, detecting, and resolving emergent behavior in distributed systems. We propose a framework to analyze and verify software system requirements with a substantially increased level of automation and consistency, as well as reduced human intervention. This framework employs an ontology-based methodology to analyze system requirements expressed by a set of scenarios. This methodology involves the automatic construction of a domain-specific ontology of the system, which is used in the analysis of system requirements. The data generated from this framework can be used to assist system engineers in constructing the design and architecture of the system. The devised methodologies from this research have been developed into a software analysis tool (EBD) to automate the process of fault detection.
The created framework and tool in the presented research have been used to analyze various real-life systems in this thesis, such as (1) a distributed system for real-time fleet-management; (2) a semantic search engine involving social network of multi-agent systems; and (3) an elevator system. The efficiency and effectiveness of this framework has been demonstrated through case studies and experiments.
Preface

The publications by the author of this dissertation which include materials and ideas presented in this thesis are listed in this preface.

Journal Publications


Book Chapters


Conference Publications

**C1.Y. Amannejad, M. Moshirpour, B. Far, R. Alhajj**, “From requirements to software design: An automated solution for packaging software classes” in Proceedings of the 15th IEEE
Information Conference on Reuse and Integration (IRI 2014), San Francisco, USA, August 13-15, 2014, pp. 36-43


Acknowledgements

I would like to express my deepest appreciation and gratitude to my supervisor Dr. Behrouz H. Far for inspiring me to learn, research and discover, and for supporting my work throughout the years. Without his knowledge, guidance, and patience, this research would not have been possible. He always encouraged me and provided me with the freedom to be creative. Dr. Far extended his advice to my academic career, helping me to grow as a researcher and providing me with the guidance to gain experience as an instructor. I am also thankful to Dr. Far for believing in this research and for investing his research funds into hiring interim software developers to help develop this work. I am grateful to my co-supervisor, Dr. Armin Eberlein, for always patiently reviewing my works and providing me with helpful advice and encouragement throughout my research.

I am grateful to my supervisory committee members, Dr. Mahmood Moussavi and Dr. Reda Alhajj for their generous help, guidance and support for the past years. I am grateful to Dr. Moussavi for inspiring me to become a software engineer and for teaching me much of my knowledge about software engineering and about being an instructor. I am grateful to Dr. Alhajj for being an inspiration, a teacher, and a mentor to me throughout the years.

I am grateful to Dr. Laleh Behjat and Dr. Bijan Raahemi, my examiner committee members, for reviewing this work and providing helpful guidance and feedback.

I am grateful to the researchers who collaborated with me on this research, Dr. Abdolmajeed Mousavi, Dr. Farid Shirkavand, Dr. Shimaa El-Sherif, Yasaman Amannejad, and Seyedehmehrman Mireslami. Their knowledge, expertise and innovativeness aided me greatly throughout this work.

I am also thankful to the software developers who helped program the models and algorithms for this research throughout the years: Beenish Khurshid, Maddie Lowe, Ananth Balashankar, Naomi
Hiebert, and Jitendra Gohil. Their knowledge, creativity, and hard work made the advancement of this research possible. I am thankful for Professor Lisa Bryce, and my sisters, Mojgan Moshirpour, and Mahtab Moshirpour, for their assistance in editing this thesis.

I would like to thank the Natural Sciences and Engineering Research Council of Canada (NSERC), Alberta Innovates Technology Futures (AITF) and University of Calgary Eyes High program for awarding funding to me for the research of this dissertation.

I would like to thank the Department of Electrical and Computer Engineering and the University of Calgary for providing financial and administrative support and for giving me the opportunity to teach while completing this dissertation.

Finally, I would like to express my most sincere gratitude to my family: my parents, Mahnaz and Hossein, my sisters, Mojgan and Mahtab, my brother Yousef and my fiancé Mahsa, for their unconditional support, unlimited kindness, and constant encouragements. I am confident that without their help and support this thesis would not have been possible.
Dedication

To my mother. To my father.
Table of Contents

Abstract ........................................................................................................................................... ii
Preface............................................................................................................................................... iv
Acknowledgements ......................................................................................................................... ix
Dedication ......................................................................................................................................... xi
Table of Contents ............................................................................................................................ xii
List of Tables ..................................................................................................................................... xv
List of Figures and Illustrations ......................................................................................................... xvi
List of Symbols, Abbreviations and Nomenclature ......................................................................... xix

CHAPTER ONE: INTRODUCTION .............................................................................................. 1
  1.1 Background ............................................................................................................................. 1
  1.2 Research Motivation and Relevance ....................................................................................... 1
  1.3 Research Objectives and Questions ....................................................................................... 1
  1.4 Research Methodology and Scope ......................................................................................... 5
    1.4.1 Behavior Modeling ............................................................................................................ 9
    1.4.2 Detection of Emergent Behavior ..................................................................................... 9
  1.5 Research Benefits ................................................................................................................... 11
  1.6 Contributions ........................................................................................................................ 12
  1.7 Thesis Organization ................................................................................................................. 13

CHAPTER TWO: BACKGROUND ................................................................................................. 15
  2.1 Scenario-Based Specification ................................................................................................. 15
  2.2 Analysis Methodologies for Scenario-Based specifications .................................................. 15
  2.3 Ontology ................................................................................................................................ 15
  2.4 Distributed and Multi-Agent Systems ................................................................................... 15
  2.1 Scenario-Based Specifications ............................................................................................... 15
  2.2 Analysis Methodologies for Scenario-Based Specifications .................................................. 17
  2.3 Ontology ................................................................................................................................ 19
    2.3.1 Ontological Engineering .................................................................................................. 20
    2.3.2 Ontology in Software Engineering .................................................................................... 23
    2.3.3 Different Views of Ontology ............................................................................................ 25
  2.4 Distributed and Multi-Agent Systems ................................................................................... 27

CHAPTER THREE: METHODOLOGY .......................................................................................... 29
  3.1 Definitions ............................................................................................................................... 30
  3.2 Input Processing ...................................................................................................................... 35
  3.3 Ontology Creation .................................................................................................................... 36
    3.3.1 Static Ontology Generation ............................................................................................. 36
    3.3.2 Dynamic Ontology Generation ....................................................................................... 37
  3.4 Behavior Modeling ................................................................................................................... 38
  3.5 Construction of the Domain Knowledge .................................................................................. 38
    3.5.1 Domain Theory ................................................................................................................. 39
8.1 Analysis Software Requirements to Detect Emergent Behavior ..................108
8.2 Summary .............................................................................................................113

CHAPTER NINE: CONCLUSIONS AND FUTURE WORK .......................114
9.1 Summary of Contributions of Thesis .................................................................114
9.2 Future Work .........................................................................................................118
List of Tables

Table 3-1. High level algorithm for synthesis of emergent while preventing overgeneralization ......................................................................................................................... 43

Table 4-1. Finding Semantic causality for state q31 .......................................................................................................................... 55
List of Figures and Illustrations

Figure 1-1. Model-based System Analysis Framework ................................................................. 7
Figure 2-1. Sequence Chart ........................................................................................................ 16
Figure 2-2. Ontology development process [55] ........................................................................... 23
Figure 4-1. High-level design of the fleet-management system ................................................. 46
Figure 4-2. System calculates bus arrival times and notifies users ........................................... 47
Figure 4-3. System recalculates bus arrival according to traffic update .................................. 47
Figure 4-4. System recalculates bus arrival according to data manually entered .................... 48
Figure 4-5. Static view of the ontology for the fleet management system ............................... 49
Figure 4-6. Dynamic ontology after the Add States step ............................................................ 50
Figure 4-7. Dynamic ontology after the Determine Next State step ......................................... 50
Figure 4-8. Dynamic Ontology after the Finality step .............................................................. 51
Figure 4-9. Generalized dynamic view of the system ............................................................... 51
Figure 4-10. FSM for the Server component in MSC 1 ............................................................. 52
Figure 4-11. FSM for the Server component in MSC 2 .............................................................. 53
Figure 4-12. FSM for the Server component in MSC 3 .............................................................. 53
Figure 4-13. Behavior model for the Server component in the fleet-management system ....... 54
Figure 4-14. Blending finite state machines for component Server .......................................... 56
Figure 4-15. Merging the identical states of finite state machines for component Server ....... 57
Figure 5-1. The initialization of concept learning process based on search query entered by a user ......................................................................................................................... 64
Figure 5-2. Steps of concept learner process .......................................................................... 64
Figure 5-3. Updating strengths of ties between a learner and one of the teachers (Peer B) ...... 65
Figure 5-4. Undesired behavior of concept learning system ..................................................... 66
Figure 5-5. Static ontology of the semantic search engine system .......................................... 67
Figure 5-6. Dynamic ontology of the semantic search engine
Figure 5-7. The union of state machines built from the legal set of scenarios
Figure 5-8. The tie between MAS A and B are established based on their mutual peers
Figure 5-9. The relation between MAS A and B is decided based on their ability to cooperate on a given query
Figure 5-10. The tie between MAS A and B are established based on how frequently they contact each other
Figure 5-11. The union of state machines built from MSCs 1 and 2
Figure 5-12. Behavior model of the system after scaling
Figure 5-13. Resulted FSM after merging identical states
Figure 6-1. Main components of the EBD tool in generating package diagrams
Figure 6-2. A simplified example of dynamic view of a system
Figure 6-3. Scenarios of the elevator system expressed using SDs
Figure 6-4. Result of hierarchal clustering on classes of sender-receivers of the elevator system
Figure 6-5. First level of packaging based on static view of the system
Figure 6-6. All the nested packages for P3 based on dynamic view
Figure 7-1 Model View Controller Architecture
Figure 7-2. EBD's high-level component diagram
Figure 7-3. Package diagram for backend of EBD
Figure 7-4. Important classes for the back-end of EBD
Figure 7-5. Important classes for the desktop version of EBD
Figure 7-6. Important classes for the web version of EBD
Figure 7-7. Create project window for EBD
Figure 7-8. Input window for EBD
Figure 7-9. Static ontology of the fleet-management system generated by EBD
Figure 7-10. Dynamic ontology of the fleet-management system generated by EBD ............ 102

Figure 7-11. Behavior model for the server component in the fleet-management system
 generated by EBD ................................................................................................. 103

Figure 7-12. Construction of the domain knowledge ................................................ 104

Figure 7-13. Generated report to show emergent behavior ........................................ 105

Figure 7-14. Automatically generated collaboration diagram by EBD .............................. 106

Figure 8-1. Dynamic ontology of the elevator system in iteration 1 ............................... 109

Figure 8-2. Unionized behavior model for the Door component in iteration 1 ............... 110

Figure 8-3. Unionized behavior model for the Door component in iteration 2 ............... 111

Figure 8-4. SDs marked by EBD for being the root cause of emergent behavior in component
 Door ...................................................................................................................... 112
List of Symbols, Abbreviations and Nomenclature

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>AOSE</td>
<td>Agent Oriented Software Engineering</td>
</tr>
<tr>
<td>CEBD</td>
<td>Component-level Emergent Behavior Detection</td>
</tr>
<tr>
<td>DFA</td>
<td>Deterministic Finite Automaton</td>
</tr>
<tr>
<td>FSM</td>
<td>Finite State Machine</td>
</tr>
<tr>
<td>eFSM</td>
<td>Equivalent Finite State Machine</td>
</tr>
<tr>
<td>GUI</td>
<td>Graphical User Interface</td>
</tr>
<tr>
<td>ITU</td>
<td>International Telecommunications Union</td>
</tr>
<tr>
<td>MAS</td>
<td>Multi-Agent Systems</td>
</tr>
<tr>
<td>MaSE</td>
<td>Multi-Agent Software Engineering (It is one of the AOSE methodologies)</td>
</tr>
<tr>
<td>MDTM</td>
<td>Model-based Detection and Testing of MAS</td>
</tr>
<tr>
<td>MSC</td>
<td>Message Sequence Chart</td>
</tr>
<tr>
<td>MVC</td>
<td>Model-View-Controller</td>
</tr>
<tr>
<td>NFA</td>
<td>Non-Deterministic Finite Automaton</td>
</tr>
<tr>
<td>OMG</td>
<td>Object Management Group</td>
</tr>
<tr>
<td>pMSC</td>
<td>Partial Message Sequence Chart</td>
</tr>
<tr>
<td>SBS</td>
<td>Scenario-Based Specifications</td>
</tr>
<tr>
<td>SBSE</td>
<td>Scenario-Based Software Engineering</td>
</tr>
<tr>
<td>SD</td>
<td>Sequence Diagrams</td>
</tr>
<tr>
<td>SEBD</td>
<td>System-Level Emergent Behavior Detection</td>
</tr>
<tr>
<td>SUD</td>
<td>System Under Development</td>
</tr>
<tr>
<td>TRAM</td>
<td>Transition between Requirements and Architecture using a Modularized framework</td>
</tr>
<tr>
<td>UML</td>
<td>Unified Modeling Language</td>
</tr>
</tbody>
</table>
Chapter One: Introduction

1.1 Background

Review of requirements and design documents before implementing software systems is an effective and efficient approach to prevent introducing flaws into a system. Literature suggests that detecting unwanted behavior during the design phase is about 20 times cheaper than finding them during the deployment phase [1]. Unfortunately, manual review of the requirements and design documents may not effectively detect all the design flaws because of the scale and complexity of real-life systems, and is too time and resource intensive.

An effective and efficient approach to describe system requirements is using scenario-based specifications. A scenario is a temporal sequence of messages sent between system components and actors using the system. Scenarios are appealing because they allow stakeholders to describe system functionality with 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 distributed 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 used to define these interactions. By following this approach, the overall behavior of the system can be defined by a comprehensive set of scenarios [3].

1.2 Research Motivation and Relevance

Several reasons contribute to the need for a systematic and automated analysis of software requirements expressed using Scenario-Based Specifications (SBS):

1. 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 as well as distributed systems, which consist of multiple system components. For instance, because each scenario gives only a local and partial story of a distributed system's behavior, the challenge is how to construct the overall behavior of a system from a set of scenarios and more importantly whether the derived behavior is acceptable or not [3]. Generally, system requirements described using scenarios are prone to several possible defects [4]:

   a. Scenarios are partial stories of the system's behavior, and each scenario is only an instance of the system's functionality. Therefore, defining system requirements of a large system using only scenarios raises issues of coverage and completeness.

   b. Scenarios are instances of system behavior and thus they need to be properly combined with other scenarios to have a full description of the system.
2. Although there are several methodologies [2-7] that take a systematic approach to analyzing system requirements expressed using SBS, many contain limitations and require a great number of assumptions and inputs from the system engineer. This research recognizes the importance of increasing the level of automation in these methodologies and strives to achieve this using semantic ontologies [8, 9].

3. Software engineering focuses on establishing criteria for good design such as designing system components to allow for low coupling and high cohesion. The best design choices, however, are not always obvious, particularly for more complex systems. Consequently, the quality of the system can be compromised due to over-looking details in the requirements such as the volume of message-passing between different components. Therefore, it is highly desirable to establish systematic and automated solutions to increase the quality of the design of software system while improving the efficiency of the design process [10].

1.3 Research Objectives and Questions

Lack of central control makes the design of distributed software systems a challenging task because of possible unwanted behavior at runtime, commonly known as emergent behavior. Emergent behavior is a specification of behavior that is exhibited by the distributed system but is not explicitly specified in its specifications. Generally, emergent behavior arises when there exists a state in which a system component becomes confused as to what course of action to take [3, 4]. The main objectives of this research are:

1. Developing methodologies to analyze system specifications to detect emergent behavior before implementing a system. Early identification of possible design flaws that may lead to the detection of emergent behavior in distributed systems by analyzing system specifications, can
result in significant savings in time and cost [1]. Manual review of requirements and design documents may not efficiently detect all the design flaws because of the scale and complexity of real-life systems. Therefore, automation of analysis methodologies is considered greatly beneficial [11-13].

2. As previously mentioned, a number of methodologies [2-7] take a systematic approach to analyzing system requirements expressed using scenarios. In Chapter 2 merits and disadvantages of each methodology are explained: finally, the work of [4] is selected and justified as the most effective and efficient approach to analyze scenario-based specifications. The approach of [4] is a semi-automated methodology that requires a great deal of input from the user for the analysis process. This research proposes an approach which decreases the amount of input needed from the user by extracting this input from the ontology which has been constructed for the system to be built [14]. The work done in [3] utilized and modified methodologies developed in [4] to analyze requirements and design for a variety of software systems such as distributed and multi-agent systems at the component level.

3. Software engineering focuses on establishing criterion for a good design such as designing system components to allow for low coupling and high cohesion. However the best design choices are not always obvious, particularly for more complex systems. Consequently, the quality of the system can be compromised due to overlooking details in the requirements such as the volume of message-passing between different components. Therefore, it is highly desirable to establish systematic and automated solutions to increase the quality of the design of the system while improving the efficiency of the design process. However, to make systematic analysis of system requirements possible, they need to be expressed using a precise notation. This research presents a unique solution towards automating the software design
process as part of a framework for the analysis of software requirements and design. This solution contributes to better design of the system architecture as it provides software engineer with a package diagram based on the system requirements in the early stages. Furthermore, software engineers can make decisions about the non-functional requirements of the system by considering the optimal arrangements of system components. This leads to shortening the release cycle as well as savings in time and cost of the project [15].

This research proposes an approach that decreases the input needed from the system engineer by extracting this input from the ontology that has been constructed by the system engineer for the system to be built [14]. This leads to the following research questions:

(Q1) what is the cause of emergent behavior in distributed systems?
(Q2) how to model and analyze the system to find emergent behavior?
(Q3) how to detect and resolve the problem?
(Q4) how to move from software requirements to design using a systematic approach?
(Q5) how to automate the developed methodologies?

1.4 Research Methodology and Scope
During the course of this research, a detailed review was conducted of existing methodologies in the literature devised to analyze the requirements and design of software systems was conducted. The result of this study is presented in Chapter 2 where the merits and disadvantages of each methodology are explained and the work of [4] is selected and justified as the most effective and efficient approach to analyze SBS. Thus, the methodology of [4] has been used as the starting point of this research.
During the initial phase of this research, which was presented as a MSc thesis [3], the approach of [4] was expanded and utilized in a variety of areas such as analysis of multi-agent systems [16], network security [17], social networks and agile development [18].

Although the approach of [4] is an automated one, as will be shown in Chapter 3, it requires a great deal of input from the user to calculate state values. This research proposes an approach that decreases the amount of input needed from the user by extracting this input from the ontology that has been constructed for the system to be built [14].

This research strives to establish a framework for analyzing the requirements and design of software systems. Figure 1-1 demonstrates the broad scope of this research. As shown, this framework takes system requirements expressed using scenarios as input and proceeds to extract the domain knowledge of the system using an ontology-based approach. In this approach, the domain theory (is defined in Chapter 3, Definition 5) of the system is constructed that contains causal relationships between pairs of messages [3, 4]. This information is used to detect emergent behavior in the system requirements. As shown in Figure 1-1, the areas highlighted in blue were developed in [3], while the areas highlighted in purple are completed in this thesis.
The framework developed in this research focuses on analyzing software requirements by

a) Automating the process of modeling, analyzing, detecting, and resolving emergent behavior in distributed systems [8, 9, 19-30].

b) Developing a systematic and automated approach to transition from software requirements to software design [15]

The research questions given in Section 1.3 are addressed in this thesis.

For Q1, a formal treatment of the cause of emergent behavior through defining semantic causality is provided. For Q2 we have devised a methodology for analysis and behavioral modeling at the component level. The main focus of this research is on Q3, Q4, and Q5, for which we provide mechanisms and a tool.
For Q3 we devise a process to model, analyze, detect and remove the emergent behavior at the component level. The process for Q3 is a 4-step process:

(1) **Modeling:** We use scenarios to describe system requirements. A scenario, commonly known as OMG's Sequence Diagram (SD) [31] or ITU's Message Sequence Chart (MSC) [32], is a temporal sequence of messages sent between system components. We devise methods to convert other modeling constructs (such as use-cases, user stories, and design constructs of agent-oriented software engineering, AOSE methods) to scenarios.

(2) **Analysis:** Scenarios give 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 whether the derived behavior is acceptable or not. The model that describes the behavior of each system component is called a behavioral model. We use state machines for behavioral modeling. Several studies have already been conducted to facilitate the procedure of converting a set of scenarios to state machines [4, 33-35]. In this phase, one state machine will be built for each system component in each scenario. Then, behavior of a component is described by parallel execution of all the state machines of that component.

(3) **Detection:** To construct the behavioral model for a component/system, we define the concept of identical states which are states that remain the same during execution of multiple scenarios. Identical states in the constructed behavioral model are the places where emergent behavior can potentially occur. We devise a method to identify identical states using the semantic causality and state values. A main contribution is devising a method to automatically identify the semantic causality using an ontology of the domain.

(4) **Resolution:** We merge identical states to create a final state diagram for a component/system and devise methods to examine and test the possible behavior by focusing on identical states.
A main difficulty for Q5 is to establish the semantic causality for events. We propose the use of an ontology-based approach to analyze system requirements. This methodology involves building a domain-specific ontology of the system, and examines the requirements based on this ontology. The advantages of this approach in comparison with other methodologies are its accuracy, consistency, and increased level of automation.

The general approach used for analyzing the requirements of software systems is done in two steps: *behavior modeling*, and *detection of emergent behavior*. The framework developed in this research fully automates these steps by using an ontology-based solution that creates the domain theory of the system automatically. These steps are briefly described in the following subsections.

### 1.4.1 Behavior Modeling

The model that 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, 6, 7, 35-37]. In the synthesis process, one state machine is 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.

### 1.4.2 Detection of Emergent Behavior

One of the challenges during the synthesis process, is *implied scenarios* [4, 5, 34, 38, 39], also known as *emergent behavior* [4]. 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 a state exists, in which the system component cannot determine what course of action to take next. This happens when identical states exist in the union of state machines obtained through behavioral modelling. A definition for identical states (Definition 7 given in Chapter 3) is needed for the detection of emergent behavior. To detect emergent behavior, we must first have a clear procedure for assigning values to the states of the state machines. This is a very important step and is performed differently by various researchers. For instance, the work presented in [2, 7] 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 [4] proposes an approach that makes use of an invariant property of the system called semantic causality. Semantic causality means that (a) a received event cannot happen unless its corresponding send event happened before; and (b) an event cannot happen until all the causal predecessors of it have been accomplished. The principle of semantic causality will be defined formally and explained in detail in Chapter 3 of 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 increase consistency, as discussed in Chapter 2. This is one of the main reasons that the approach of [4] was selected by this research as the most efficient methodology for analyzing scenario-based specifications, and it is used and expanded on in this work.
These behaviors are not inherent to the specifications and depend solely on the assumptions and the generalization techniques used in the synthesis approach. This is the reason they have been referred to in the literature as a side effect of generalization, also known as overgeneralization [37]. 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.

1.5 Research Benefits

Conventional approaches to develop and test distributed software systems for unwanted behavior are expensive and inefficient. Manual review of the requirements and/or design documents may not be effective, depending on the scale and complexity of the real-life systems. This work aims to develop a framework for analyzing the requirements and design of software systems. It proposes a methodology to automate the process of analysis which leads to savings in time and cost of software development [1]. Furthermore, this research introduces a software tool that uses this methodology to analyze the requirements and design artefacts of real-life software systems and to identify potential unwanted behavior.

This research has devised a unique solution which automates the generation of package diagrams in software design process. We have employed a clustering algorithm, and have defined a similarity measure for packaging classes of the software. The similarity measure is defined to increase the cohesion and decrease the coupling between the packages. The process of moving from requirement to design is traditionally done through an ad-hoc process. Although the criteria for a good design is well-defined in software engineering for different system architectures, the design of the system is only as good as the design choices of the engineers. Therefore, having a
systematic solution that recommends design choices based on system requirements is highly desirable, and it leads to increasing the quality of software as well as savings in cost and time.

There are a variety of different uses for the developed methodology in this research. For instance, the methodology in this research can be used to ensure the lack of certain unwanted behavior in a software system and thus verify the system against particular defaults [30, 40]. Alternatively, this methodology can also be used in analyzing and detecting security vulnerabilities in computer networks and filtering systems [17]. Also, this framework can help software engineers to transition from the requirement analysis phase to the design phase [15]

1.6 Contributions

The contributions of this research is a methodology and tool that automate the process of software requirement analysis. In detail, the contributions of this thesis are as follows:

1. A framework to analyze and verify software system requirements with a substantially increased level of automation and consistency, as well as reduced human intervention. This framework employs an ontology-based methodology to analyze system requirements expressed by a set of scenarios. This methodology involves the automatic construction of a domain-specific ontology of the system, which is used in the analysis of system requirements.

2. A new approach for synthesizing behavior models from scenarios while reducing the intervention of the domain expert in the synthesis process by extracting information from a defined ontology for the system. Using ontologies greatly simplifies and automates the process of emergent behavior detection.

3. A new methodology to construct the domain-specific ontology of the system based on the complete set of software requirements expressed in scenarios.
4. A new approach to construct a high-level package diagram and a system-level communication diagram of the entire system based on the constructed ontology and the complete set of software requirements expressed in scenarios. These diagrams can be used to guide system architects and engineers to transition from the requirement analysis to the design stage. This systematic approach leads to increasing the quality of software as well as savings in resources.

5. The framework created in this research has been implemented into a software tool to apply the proposed methodologies to requirement documents of a variety of software systems such as distributed and multi-agent systems.

6. Applying this framework to a wide range of software systems to demonstrate the efficiency and effectiveness of this approach as well as testing the usability of the implemented tool. These systems include information retrieval system, MAS, social networks, network security systems, engineering and manufacturing distributed systems [8, 9, 19-30].

The publications resulted from this work are listed in the Preface of this thesis.

1.7 Thesis Organization
This research is presented in nine chapters. Chapter 2 provides background information. Chapter 3 outlines the methodology devised by this research in discovering emergent behavior. This methodology includes some definitions necessary for the analysis process, the process of behavior modeling, and the process of automated ontology creation, as well as the algorithm that extracts the domain knowledge of the system from system scenarios and ontology, and the algorithm for detection of emergent behavior. Chapter 4 contains a case study of a real-time fleet management system using the devised methodology. In Chapter 5, the capability of this approach to certify a system against the occurrence of a particular scenario is illustrated using the case study of a social
network of agents. Chapter 6 illustrates the capabilities of this methodology to provide the means to transition to system design using the system requirements and the constructed ontology. The developed tool to analyze software systems (EBD) is presented in Chapter 7. Chapter 8 demonstrates how a software development project can be managed and verified at each iteration with no significant added overhead using the EBD software. Finally, the summary of the research and future works are given in Chapter 9.
Chapter Two: Background

2.1 Scenario-Based Specification
2.2 Analysis Methodologies for Scenario-Based specifications
2.3 Ontology
2.4 Distributed and Multi-Agent Systems

This research attempts to automate the analysis of requirements and design documents for software systems, particularly systems without central control such as distributed systems and Multi-agent Systems (MAS). Furthermore, during the course of this work, the developed methodologies have been utilized in the area of network security [20, 23], distributed and multi-agent systems [24, 25, 28], social network [26, 27, 30], and agile development [18]. This chapter provides background information necessary to understanding this research.

2.1 Scenario-Based Specifications
Scenarios have become popular as a powerful means of communication for system requirements due to their simplicity and expressive power [41]. Using scenarios, different groups of stakeholders can communicate their ideas about software systems in an easy-to-understand manner. In addition to their use in requirements engineering as shown in [41], scenarios have been utilized in other aspects of software engineering such as code synthesis [42], reverse system engineering [2], and model-based testing [43].
Different definitions of scenarios exist [44-47]; however, in general, scenarios are described as narrative stories of the interactions between 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.

Several approaches have been proposed in the literature for representing scenarios. These approaches include narrative text [48], annotated cartoons, video recordings, scripted prototypes, and sequence charts [2, 44, 49]. Each approach has 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 more informal and pose real challenges for automated analysis.

Among the abovementioned approaches for presenting scenarios, sequence charts are the most efficient for analyzing 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.

![Sequence Chart](image)

**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 generally used to describe scenarios are Message Sequence Charts (MSCs) standardized by the International Telecommunications Union (ITU) [32] and Sequence
Diagrams developed by the Object Management Group (OMG) as a part of UML [31]. 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 efficiently and intuitively. In this research, the prime focus is on MSCs, but the developed framework can work with both notations. There are several reasons for choosing 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 [31, 32]. Sequence diagrams include additional notations to support illustrating the design. In this research, because scenarios are used to communicate system requirements among 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 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. Therefore, a possible future extension for this research would be incorporating the sequence diagram notation in requirement validation methodologies.

2.2 Analysis Methodologies for Scenario-Based Specifications
As mentioned previously, several methodologies [2-7] take a systematic approach to analyzing system requirements expressed using scenario-based specifications. Each methodology has devised algorithms to take scenarios as input and identify emergent behaviors in the requirements. The general structure of these algorithms is similar and is often done in two major steps of behavior modeling and detection of emergent behavior as briefly explained in Chapter 1 of this document.
Recall that upon building the behavior model of a system component, different states of the model need to be identified. This is a 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 [6] 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 the states. The work of Whittle and Schumann [2, 7] 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 [50]. The OCL specifications include the declaration of state variables. A state variable represents some important aspects of the system such as whether 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 [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 values 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 to have a systematic approach for detecting emergent behavior, a consistent methodology for different domain experts is a must. The approach followed in the work of Mousavi [4], addresses this issue by making use of an invariant property of the system called semantic causality. This property is defined as the sequence of
messages (events) that a component has to keep to perform the subsequent operations. A formal definition of semantic causality is provided in Chapter 3 of this thesis where its pivotal role in the detection of emergent behavior is discussed.

Therefore, since the methodologies presented in [4] provide a 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 used as the basis of the presented methodology created in this research.

However, even though the approach of [4] is an automated one, it requires a great deal of input from the user to calculate state values as will be discussed further in Chapter 3. This research presents an approach that decreases the amount of input needed from the user by extracting this input from an ontology constructed for the system to be built.

2.3 Ontology

A commonly accepted definition for an ontology is as follows: "An ontology is an explicit and formal specification of a conceptualization of a domain of interest" [51]. Thus, two key points about ontology are established: (1) The conceptualization is formal; therefore, it allows reasoning by computers, and (2) A practical ontology is designed for a specific domain [51]. An ontology consists of concepts (also known as classes), relations (also known as properties), instances, and axioms. A more concise definition of an ontology is as a 4-tuple <C, R, I, A>, where 'C' is a set of concepts, 'R' a set of relations, 'I' a set of instances, and 'A' a set of axioms [52].

The importance of ontology has been recognized in a variety of areas such as semantic web applications, knowledge engineering, database design and integration [51, 53]. Depending on the
application, different views of ontology as well as different ontology languages and tools are utilized [54-56]. This section provides the necessary background on ontology.

2.3.1 Ontological Engineering

The set of activities that concern the ontology development process such as the ontology life cycle, the principles, methodologies for building ontologies, and the tools and languages that support them are referred to as ontological engineering [55]. Some of these ontology-related tools and methodologies are addressed in this section.

Ontology Languages

The creation of ontology languages started at the beginning of the 1990s [55]. These languages were mostly based on the evolution of existing knowledge representation (KR) languages [55]. The widespread popularity of the internet led to the development of ontology languages to support the characteristics of the web. These languages are typically referred to as Web-based Ontology Languages [55]. The syntax of such languages is based on existing markup languages such as HTML and XML [55]. Ontology languages that are currently actively supported are Resource Description Framework (RDF) [57], RDF Schema [58], and OWL [59].

Ontology Tools

Ontology tools were developed in the mid-1990s [55]. Ontology tools can be categorized as follows [55]:

- Tools that are specifically designed to work with a particular ontology language. Therefore, these tools are developed as ontology editors for specific languages. Some examples of these kinds of tools are SWOOP [60] and KAON2 [61], which support OWL.

- Tools that have an extensible architecture and their model is independent of ontology languages. Such tools provide a set of ontology-related services and are easily extended to
other modules to supply more functions. Some examples of these kinds of tools are Protégé [62], WebODE [63], and OntoEdit [64].

**Ontology Design Principles**

Some design criteria and principles that have been noted as useful in the literature are as follows [55, 65]:

- **Clarity** - The intended meaning of the defined terms should be communicated effectively by the ontology.
- **Minimal encoding bias** - The conceptualization should be defined at the knowledge level; independent of a particular symbol-level encoding. The goal should be to minimize encoding bias for knowledge sharing because the agents that share knowledge may be implemented in different ways.
- **Extendibility** - It should be possible to define new terms for special uses based on the existing vocabulary without the need to revise existing definitions.
- **Coherence** - An ontology should be coherent; it should authorize definitions that are consistent with its definitions.
- **Minimal Ontological Commitments** - Ontological commitment is based on the consistent use of vocabulary. Therefore, ontological commitment can be decreased by specifying the weakest theory and defining only terms which are essential to the communication of knowledge. For instance, a specific format for dates or currencies should not be defined when designing an ontology, because these details might vary in different systems.
**Ontology development life-cycle and process**

The ontology development process includes activities that are involved when building ontologies. These activities are categorized in the three categories of Ontology Management, Ontology Development Oriented, and Ontology Support as illustrated in Figure 2-2 [55]. Each of these three categories are briefly explained in this section [55]:

- **Ontology management Activities** - Contains tasks such as scheduling, control and quality assurance. Scheduling identifies the tasks to be completed, their order of completion and time, and the resources needed for each. The control activity ensures that the scheduled tasks are completed properly, and quality assurance ensures the quality of each activity.

- **Ontology development-oriented activities** - As shown in Figure 2-2, these activities are categorized into pre-development, development, and post-development activities.

- **Ontology Support Activities** - Contains a number of activities, which as shown in Figure 2-2, can be performed during the development-oriented activities. Without these activities, the ontology could not be constructed.
2.3.2 Ontology in Software Engineering

Ontologies have been used in many different aspects of software engineering [54, 56, 66-70]. As software systems grow more complex, software engineers and developers need to devise methodologies to manage the amount of information and knowledge. Ontologies provide means to store the knowledge generated during the software development process [66]. The work presented in [66] describes a Semantic Reuse System (SRS), which uses ontologies that are represented using the knowledge representation language of the semantic web for software development knowledge reuse.

Ontologies have been utilized in Software Development Environments (SDE) [69]. An SDE is a computational system that provides support for the construction, management, and maintenance
of a software product [71]. SDEs provide integrated case tools, guidance to the software process and common repositories to the development teams. However, SDEs do not provide any information about the domain of the related tasks [69]. To resolve this issue, the concept of SDE was extended by introducing domain and task knowledge to it in order to direct software engineers through software development phases leading to the development of the Domain-Oriented Software Development Environment (DOSDE) [72]. One of the main components for the infrastructure of DOSDE are ontologies [69].

The work in [68] presents examples of using ontologies in each stage of the software development lifecycle. For instance, in the requirement and design stage, [68] suggests the use of ontology for describing requirements specification document and formally representing requirements knowledge [73, 74]. The end-to-end use of ontologies in analysis, design, and implementation is appropriate for rapid application development. Moreover, using this approach, the interoperability of ontologies with other components or applications is improved [68]. In addition, using a web-based knowledge representation format allows engineers to discover sharable domain models from both internal and external repositories [68]. In addition, ontologies are utilized in software maintenance, testing, coding support, and code documentation [68, 70].

In addition to building ontologies to address different needs in software engineering, some research suggests extracting ontologies from software design artifacts such as software object models [56]. The work of [56] suggests that ontologies are closely related to modern object-oriented software engineering; therefore object oriented techniques can be adapted for the development of ontologies [56]. This research also states that there descriptive ontologies and database schemas are similar [56]. It can be deducted from this research that artifacts from object-oriented design can be a
starting point for building an ontology for a given system. Some of the differences between ontologies and object-oriented modeling are as follows [56]:

- Ontology is theoretically based on logic, so ontology allows automated reasoning. However, object-oriented modeling does not include inference.
- Ontology and object-oriented modeling differ in the treatment of properties. Ontologies give properties priority, while object oriented modeling does not. For instance, ontologies allow the inheritance of properties.

However, it is important to note that despite these differences, object-oriented modeling techniques and UML are accepted as practical ontology specification methods from the similarities mentioned and their extensive use in the industry and the multitude of existing models in UML [56].

2.3.3 Different Views of Ontology

There are diverse classifications of ontologies with different focuses in the literature. For instance, [53] classifies ontologies according to generality level; some of that are as follows:

- **High-Level ontologies**: Describe general concepts such as space or time. High-level ontologies are independent of a particular domain.
- **Domain ontologies**: Illustrate the vocabulary related to a generic domain by specializing of the introduced concepts of high-level ontologies.
- **Task ontologies**: Used to explain the vocabulary that is related to a generic activity by means of specialization of the introduced concepts of high-level ontologies.

Alternatively, [75] classifies ontologies based on the type of conceptualization structure.

- **Terminological ontologies**: State the terms that are used to represent the knowledge of a domain. They attempt to establish a unified language which is related to a particular area.
• **Information ontologies:** Determine a framework for the standard storage of information by stating the structure of database records.

• **Knowledge representation ontologies:** State the conceptualization of knowledge with an internal structure that surpasses those of the previous ones. These type of ontologies focus on a description of a specific knowledge use.

Another way of classifying ontologies is based on the nature of the real-world issues [76]:

• **Static ontologies:** These ontologies are used to describe things that exist: their attributes as well as relations between them. In this classification, it is assumed that the world is made up of entities that are unique. Static ontologies are often represented using tree structures [77].

• **Dynamic ontologies:** These ontologies describe aspects of the modeled world that are capable of changing with time. To model dynamic ontologies, it may be necessary to use finite state machines (FSM), Petri nets, etc. Examples of terminology commonly included in this category are process, state, or state transition [54].

• **Intentional ontologies:** Such ontologies describe the aspects of the world of motivations, goals, beliefs or choices of the involved agents. Some common terms used in association with these types of ontology include aspect, object support or agent.

• **Social ontologies:** These ontologies state social aspects such as organizational structures or networks. They include terms such as actor, roles, position.
2.4 Distributed and Multi-Agent Systems

Distributed systems consist of two or more autonomous components that communicate through a network [78]. These components interact with one another to achieve a common goal. Concurrency and lack of central control are among the most distinct characteristics of such systems [78, 79]. Multi-agent systems can be categorized as a type of distributed systems.

The concept of multi-agent software systems dates back to the early 1980s [80, 81]. Over the years, interest in this area has grown enormously. This is partially because agents are attractive software paradigms that provide the opportunity to exploit the possibilities presented by large open distributed systems such as the internet [81]. 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 [3, 81].

An agent is a computer system that is situated in an environment and is capable of autonomous actions in this environment to meet its design objectives [81]. 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 [81]. 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.

The current work on MAS verification is divided into two categories of axiomatic and model checking approaches [82]. In [83] 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 [82]. Furthermore, in design by contract [84],...
pre and post-conditions and invariants for the methods or procedures of the code are defined and verified in runtime: violating any of them results in an exception. However, as stated in [82], the main issue is that this technique does not check program correctness; rather, it 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 [85] and model checking of MAS with the MABLE programming language [86] are a few examples of model checking approaches that use the SPIN model checker [87] - a verification system for detection of faults in the design models of software systems.
Requirement elicitation is one of the most challenging stages of the software development lifecycle. Many bugs are introduced into the system as the result of incomplete or inconsistent requirements. However, manual review of the system requirements is time consuming and inefficient. Therefore, devising automated and systematic methodologies to analyze and verify software requirements is necessary.

This thesis introduces a framework to analyze software system requirements with a substantially increased level of automation and consistency, as well as reduced human intervention in comparison with other existing methodologies (see Chapter 2 for details). At the heart of this framework is an ontology-based methodology that analyzes software requirements defined using Scenario-Based Specifications (SBS). This methodology involves the automatic construction of a domain-specific ontology of the system, which is used in the analysis of system requirements. The devised methodologies from this research have been developed into a software analysis tool called...
Emergent Behavior Detector (EBD) to automate the process of fault detection as outlined in Chapter 7.

Chapter 2 presented a number of methodologies from the literature that are used to analyze system requirements [2-7]. The merits and disadvantages of each methodology were explained; finally the work of [4] was selected as the basis to analyze scenario-based specifications due to its consistency of assigning state values. Thus, the approach of [4] has been used as the starting point of this research.

However, although the approach of [4] is systematic, it requires a great deal of input from the user with regards to the domain knowledge to begin analysis. This research proposes an alternative solution that decreases the amount of input needed from the user by extracting this input from the ontology which has been constructed for the system to be built [14]. The proposed methodology and its differences with the work of [4] are outlined in this section. This methodology is divided into the following steps:

1. Input Processing
2. Ontology Creation
3. Behavior Modeling
4. Construction of the Domain Knowledge
5. Detection of Emergent Behavior

3.1 Definitions

In this section, key concepts and definitions related to scenario notation [5, 32, 88] and methodologies devised in this research are provided based on the work of Mousavi [4]:
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 $\Sigma_i = \{i! j(c), i? j(c) | j \in P \setminus \{i\}, c \in C\}$ 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 $\Sigma$ 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** [4] (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 $E! = \{e \in E | \exists i, j \in P, c \in C : \alpha(e) = i! j(c)\}$ and the set of receive events as $E? = E \setminus E!$.
- $\beta: E! \rightarrow E?$ 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 $\{(e_1, e_2) | e_1 < e_2, \exists i \in P: e_1, e_2 \in E_i\} \cup \{(e, \beta(e)) | e \in E\}$ 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 before its corresponding send
event. Second, a receive (or send) event cannot happen until all the previous events, which are its causal predecessors, have already been accomplished. Obviously, if all of the send events have their corresponding receive events (i.e. as defined by the function $\beta$), 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 $\beta$ 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** [4] (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 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 $\alpha_m(e_j) = m|_i[j−1]$, $0 < j < |m|_i|$. 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, 4, 5, 7] 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** [4] (Equivalent Finite State Machine for a projection): For the projection $m|_i$, we define the corresponding deterministic finite state machine $A^m_i = (S^m, \Sigma^m, \delta^m, q^m_0, q^m_f)$ such that:

- $S^m$ is a finite set of states labelled by $q^m_0$ to $q^m_{|m|_i}$.
- $\Sigma^m$ is the set of alphabet
- $q^m_0$ is the initial state
• \( q^m_f = q^m_{||m||} \) is the final state (accepting state)

• \( \delta^m \) is the transition function for \( A^m_i \) such that \( \delta(q^m_j, m|j]) = q^m_{j+1}, 0 \leq j \leq ||m|| - 1 \).

1. Thus the only word accepted by \( A^m_i \) 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 \( \omega \) 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 \( M \).

**Definition 4 [4]** (Semantic causality): A message \( m|_i[j] \) is a semantical cause for message \( m|_i[k] \) and is denoted by \( m|_i[j] \searrow m|_i[k] \), if component \( i \) has to keep the result of the operation of \( m|_i[j] \) in order to perform \( m|_i[k] \).

Using semantic causality, the approach of [4] proceeds to build the system’s domain theory as defined by Definition 5.

**Definition 5 [4]** (Domain theory): The domain theory \( D_i \) for a set of MSCs \( M \) and component \( i \in P \) is defined such that for all \( m \in M \), if \( m|_i[j] \searrow m|_i[k] \) then \((m|_i[j], m|_i[k]) \in D_i \).

The domain theory is then used to assign values to the states of FSMs according to Definition 6.
**Definition 6** [4] (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 $m |_i [j - 1] \xrightarrow{s_e} m |_i [l], 0 < j < k, k \leq l < f$

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

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

**Definition 7** [4] (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 $0 \leq t < j: m |_i [t] = n |_i [t]$

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

**Definition 8** [4] The component $l$ in MSC $m$ has emergent behavior in state $s_j^m$, if there exists MSC $n$ ($m$ and $n$ can be equal) and a state $s_k^n$, such that $s_j^m$ and $s_k^n$ are identical states and one of the following holds:

i) $m |_i [j] \neq n |_i [k] = i! l(c)$ where $l$ is a component and $i! l(c)$ represents a message with content $c$ sent from $i$ to $l$.

ii) $m |_i [j] \neq n |_i [k] = i? l(c)$ where $l$ is a component and $i? l(c)$ represents a message with content $c$ received by $i$ from $l$. Component $l$ sends a message with content $c$ to component $i(l! l(c))$ such that component $i$ does not receive this message before the event of $m |_i [j]$ in MSC $m$ and by removing this event, still component $l$ can send $l! i(c)$.
### 3.2 Input Processing

An effective and efficient way to describe system requirements is using scenario-based specifications. A scenario is a temporal sequence of messages sent between a system and actors using the system. Scenarios are appealing because they allow stakeholders to describe system functionality with 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.

This research has devised an automated solution for the analysis of software requirements defined using scenario-based specifications. However, in order to analyze software requirements in an automated manner, the requirements need to be expressed using a specific notation.

As outlined in Chapter 2, the two main notations for representing scenarios are Sequence Diagrams (SD) and Message Sequence Charts (MSC) [31, 89]. The solution devised in this research has the capabilities to analyze scenarios illustrated using either notations.
3.3 Ontology Creation

This research introduces a unique systematic and automated solution of using ontologies in creating the domain knowledge of the system. Ontologies can be used to greatly simplify and automate emergent behavior detection. Automating ontology generation greatly reduces the input required from the domain expert, and would then only require a comprehensive set of scenarios which describe the system. This research proposes that two different views of ontologies are built for the system; namely static view and dynamic view. An account of different views used to represent ontologies is presented in Chapter 2. The methodology used to automate ontology generation focuses on creating dynamic ontology and dismisses static ontology as a redundant element in the system.

3.3.1 Static Ontology Generation

The static view of ontology is much like a tree structure, where the elements are components of the system and are related to each other in this ontology based on their hierarchy within the system. The static ontology here is used to generate knowledge about the system. All required system knowledge should be present in the full set of MSCs. Thus, for each MSC i in MSCs, we find a non-mutually exclusive partial set of knowledge. For each component c in i, we consider adding c under the “System” node in the static ontology. If a similar node d exists under “System” we refrain from adding c to the ontology. Otherwise, we add c.

This algorithm has two main assumptions:

1) All knowledge in the system is present as components in the full set of MSCs; and
2) Any knowledge hierarchy within the system is irrelevant.
3.3.2 Dynamic Ontology Generation

To gather knowledge about the behavior of the system, we use the knowledge present in the static ontology as well as the MSCs. There are three steps to this technique: 1) Add states, 2) Determine next-state relationships, and 3) Determine finality.

**Step 1 – Add States**

For each node n in the static ontology under the “System” node, add a state s to the dynamic ontology.

**Step 2 - Determine Next State**

For a message m in MSC i, we note f as the component that m originates from, and t as the component to which m is sent. Find the state s’ that relates to f, and the state s’’ that relates to t. We will now consider choosing s’’ as a next state to s’. If s’’ is already a next state to s’, do not add s’’ to s’ as a next state. Otherwise, add s’’ as a next state to s’. Repeat this process for all messages m in all MSC i in the full set of MSCs.

**Step 3 – Finality**

For every state s in the Dynamic Ontology, if the state has no next-states, then mark it as final state. If the state is not a next state to any state, then mark it as initial state. Based on categorizing system components using the static ontology, a more generalized dynamic ontology is created. As is the case with state machines, the dynamic view of ontology will have three different types of states:

1) **Start State** - Resembles a category of components that will only send messages, but will not receive any messages.

2) **Final State** - Resembles a category of components in which a type sending or receiving a message results in completing a task.
3) **Transition State** - is a state that is determined to be neither a start nor a final state.

### 3.4 Behavior Modeling

The model that 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, 6, 7, 35-37]. In the synthesis process, one state machine will be built for each system component. The state machine includes all messages received or sent by that component. Then the behavior of the distributed system is described by the product (parallel execution) of all the state machines of the system elements.

The process of behavior modelling for distributed systems is illustrated in Chapter 4, using the case study of a real-time fleet-management system.

### 3.5 Construction of the Domain Knowledge

After the synthesis of behavior models for each system component, the state values for the models are to be calculated. To do this, an invariant property of the system called semantic causality is used. A pivotal step in behavior modeling is to assign state values. This is done differently in different works as outlined in detail in Chapter 2. 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 [4].
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 be defined by the messages that particular component needs in order to perform the messages that come after its current states.

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 component to be defined by the messages that the component needs to perform the messages that come after its current states.

Following the definition of semantic causality, the domain theory of the system is constructed according the definition of the domain theory (Definition 5). The steps to be followed to build the domain theory using the ontology-based approach are outlined in this section.

3.5.1 Domain Theory

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

However, building a domain theory can be very time-consuming. Therefore as a part of this systematic approach, a processes to automatically build the domain theory is introduced.

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.
3.5.2 Constructing Tables

To build the domain theory, tables are built for each state to discover semantic causality between messages. For each state, the rows are the transitions before the state, and the columns are the transitions after it. The sender/receiver component for each message is also indicated in the tables as demonstrated in Chapter 4.

The information required to fill the constructed tables is extracted from the static and dynamic views of the ontology as follows:

- For each row, consider the component that has sent/received that message
- Using the static view of the ontology, determine what category it belongs to
  - For each column, consider the component which has sent/received that message
  - Using the static view of the ontology, determine what category it belongs to
    - Now consider the dynamic view of the ontology
    - The category of the sender/receiver component of the message in the row is the starting point
    - The category of the sender/receiver component of the message in the column is the end point
    - If there is a path from the start point to the end point AND the end point is a final state
      - The message in the row is a semantic cause for message in the column and the table completion is completed
  - End For
- End For
3.6 Detection of Emergent Behavior

One of the challenges during the synthesis process is implied scenarios [3-5, 34, 38, 39] or emergent behavior. An implied scenario is a specification of behavior that is in the synthesized model of the distributed system and is not explicitly specified in its specification as a scenario. Generally, emergent behavior arises when there exists a state in which the system component becomes confused about what course of action to take. This occurs when identical states exist in the union of state machines obtained through behavioral modelling. Identical states are those states that remain the same during execution of multiple state machines [90]. There are various techniques to detect identical states, as outlined in Chapter 2 that rely heavily on the expertise and experience of the system designer [2-7]. This research introduces an ontology-based approach that automates the process of domain knowledge construction used to calculate state values.

After constructing the behavior model for a component, a state value is calculated for each state in the behavioral model using the domain theory. The calculated state values for each state are the basis for comparing states; consequently, discovering identical states is established. Identical states are defined in Definition 7.

The presence of identical states in the behavior models may result in emergent behavior since the component may be confused when performing the next message. Therefore, dealing with these issues is an important challenge in analyzing the behavior models. Once the identical states are found, the finite state machines are merged. This is done by merging the found identical states in the behavior models.

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

Since most of the existing approaches merge all of the identical states, they result in overgeneralized state machines. In [4], it is proposed to check the criteria for the identical states to identify the ones that have emergent behavior. Definition 8 is proposed to analyze the identical states and identify the ones which result in emergent behavior.

Only the identical states that result in emergent behaviors based on Definition 8 should be merged; unnecessary merging for the other identical states should be avoided. Therefore, in this paper, an automated algorithm, Algorithm Behavior Model, is proposed, which performs synthesis of behavior models while preventing the emergent behaviors which occur as a result of converting the scenarios to the state machines. This algorithm is shown in Table 3-1.
Table 3-1. High level algorithm for synthesis of emergent while preventing overgeneralization

<table>
<thead>
<tr>
<th>Algorithm: Behavior Model</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input: Set of message sequence charts (M)</td>
</tr>
<tr>
<td>Output: State machines for each component</td>
</tr>
<tr>
<td>1. For each component $i$:</td>
</tr>
<tr>
<td>2. Make domain theory $D_i$ based on Definition 5.</td>
</tr>
<tr>
<td>3. Make state machines and assign state values based on Definition 6.</td>
</tr>
<tr>
<td>4. Find identical states based on Definition 7.</td>
</tr>
<tr>
<td>5. For each set of identical states:</td>
</tr>
<tr>
<td>a. If they lead to emergent behavior based on Definition 8:</td>
</tr>
<tr>
<td>Merge the states to one single state.</td>
</tr>
<tr>
<td>b. Otherwise, no merge is needed.</td>
</tr>
</tbody>
</table>

3.7 Summary

This research proposes a model-based framework to analyze the requirements and design of distributed software systems. An important methodology of this framework is to detect emergent behavior in system requirements. This research uses the work of work of [4] as the starting point of this methodology. The main objectives of this research is to increase consistency and level of automation. To achieve this, a methodology that employs an ontology-based solution to analyze system requirements expressed using scenarios has been developed by this research. This methodology involves the automatic construction of a domain-specific ontology of the system to
be built, which is used in the analysis of system requirements. This ontology is then used to create the domain theory for the system to be built in an automated manner. Based on this methodology, a software tool called Emergent Behavior Detector (EBD) has been developed to automate the process of requirement analysis, which is presented in Chapter 7 of this thesis. EBD takes system requirements expressed using message sequence charts (MSCs) or sequence diagrams (SDs) and reports possible areas of emergent behavior. Using this approach, all possible emergent behavior in the system will be detected in accordance to Definition 8.
Chapter Four: Requirement Analysis Case Study

4.1 Case Study: Real-Time Fleet-Management System

4.2 Analyzing the Case Study Using the Ontology-Based Methodology

4.3 Summary

The ontology-based methodology to detect emergent behavior, as defined in Chapter 3, is an important part of the framework for the analysis of software requirements created by this research. The effectiveness of this methodology is illustrated using a case study of a real-time fleet-management system.

4.1 Case Study: Real-Time Fleet-Management System

The Real-time fleet-management system provides a multi-purpose solution for transportation companies and their customers. Such systems enable concise tracking of vehicles which results in accurate scheduling. A prime example of such systems is real-time city transit information systems, which have been employed in a number of large cities. Transit users are able to receive real-time schedules of bus stops and can even register to be notified of bus arrivals by receiving text messages or emails. Due to the diverse technologies used by users, the system must provide support for different platforms such as different browsers and client applications as shown in Figure 4-1.

The transit information system keeps track of the location of each bus, using GPS data received. It then attempts to estimate the time remaining until each stop by considering other data such as weather conditions received from weather network and traffic conditions received from roadside infrastructure. The requirements of this system are defined using scenarios as illustrated in Figure
4-2, Figure 4-3, and Figure 4-4. These MSCs depict the estimation of bus arrival times by the system. It can be assumed that each category of data (i.e. traffic, weather, and location data) reaches the server on set intervals, unless there is a sudden change of conditions. For instance, the traffic data for a given street is reported by the roadside infrastructure every hour. However if the traffic condition suddenly changes (such as a traffic jam due to an accident), this information is transmitted to the server right away, and arrival times are estimated and reported accordingly. Moreover, data can be entered manually to affect bus arrival times, such as mechanical failure (Figure 4-4).

![Diagram of the fleet-management system](image)

**Figure 4-1. High-level design of the fleet-management system**
Figure 4-2. System calculates bus arrival times and notifies users

Figure 4-3. System recalculates bus arrival according to traffic update
Figure 4-4. System recalculates bus arrival according to data manually entered

4.2 Analyzing the Case Study Using the Ontology-Based Methodology

In this section, the ontology-based methodology defined in Detail in Chapter 3 is applied to the requirements of the real-time fleet-management system. The methodology takes scenarios given in Section 4.1 and analyzes them according to the following steps.

4.2.1 Ontology Construction

The complete set of scenarios are compiled to create the static and dynamic ontologies of the system as outlined in Chapter 3.
**Static Ontology Generation** - As stated in Chapter 3, static ontology is similar to a tree structure which embodies the hierarchy of the system components within the system. Figure 4-5 illustrates the static ontology for the real-time fleet-management system.

![Static Ontology Diagram]

**Figure 4-5. Static view of the ontology for the fleet management system**

**Dynamic Ontology** – The dynamic ontology is automatically generated to gather knowledge about the behavior of the system. As outlined in Chapter 3, this is done using the static ontology, as well as the comprehensive set of scenarios in three steps: 1) Add states, 2) Determine next-state relationships, and 3) Determine finality, which are depicted in Figure 4-6, Figure 4-7, and Figure 4-8, respectively, for the real-time fleet-management system.
Figure 4-6. Dynamic ontology after the Add States step

Figure 4-7. Dynamic ontology after the Determine Next State step
Based on categorizing system components (e.g. controllers, receivers, etc.) using the static ontology, a more generalized dynamic ontology is shown in Figure 4-9. As defined in Chapter 3, state Data Rec. is considered a start state, where the “User Interface” is considered a final state.
4.3 Behavior Modeling

As defined in Chapter 3, the model that 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 system elements in scenarios is called **synthesis of behavioral models**, or simply, the **synthesis process** [3]. In the synthesis process, one state machine will be built for each system element. The state machine includes all the messages that are received or sent by that element. The behavior of the distributed system is then described by the product (parallel execution) of all the state machines of the system elements.

For any process $i$ of a MSC, an equivalent finite state machine can be constructed, as explained in Chapter 3. Figure 4-10, Figure 4-11, and Figure 4-12 illustrate the eFSM constructed for the infrastructure component in MSCs 1, 2, and 3 respectively (depicted in Figure 4-2, Figure 4-3, and Figure 4-4). It is important to note that, regardless of what type of data is received by the system, this data triggers the calculation of bus arrival times. Thus, for the sake of partiality, there is no sense in distinguishing between data types received in the behavior models. Therefore, all data received is simply denoted as "Received message" or "Rec. msg" for short.

![Figure 4-10. FSM for the Server component in MSC 1](image-url)
The behavior model for the Server component is obtained by the union of all the individual state machines created for this component as depicted in Figure 4-13.
Figure 4-13. Behavior model for the Server component in the fleet-management system

4.4 Construction of the Domain Theory

Upon the synthesis of the behavior models for each system component, the state values for each state are calculated according to semantic causality as defined in Chapter. To do this, an invariant property of the system called semantic causality is used:

For instance, in MSC1 shown in Figure 4-2, the message "Rec. msg." is the semantic cause for the message "Estimate time". Because 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 component be defined by the messages that the component needs to perform the messages that come after its current states.
Following the definition of semantic causality, the domain theory of the system is constructed according to Definition 5.

Following the example above, both messages "Rec. msg." and "Estimate time" are part of the domain theory as one is the semantic cause of the other. The steps to be followed to build the domain theory using the ontology-based approach are outlined in Chapter 3.

First, to build the domain theory, tables are built for each state to discover semantic causality between messages. For each state, the rows are the transitions before the state, and the columns are the transitions after it. The sender/receiver component for each message is also indicated in the tables as shown in Table 4-1. The information required to fill the constructed tables is extracted from the static and dynamic views of the ontology as follows as outlined in Chapter 3.

<table>
<thead>
<tr>
<th>Table 4-1. Finding Semantic causality for state $q_3^1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Estimate time [Server]</td>
</tr>
<tr>
<td>-----------------------</td>
</tr>
<tr>
<td>Rec. msg [Weather Data Rec]</td>
</tr>
<tr>
<td>Rec. msg [Traffic Data Rec]</td>
</tr>
<tr>
<td>Rec. msg [Location Data Rec]</td>
</tr>
</tbody>
</table>

55
After building the domain theory based on semantic causality, we proceed to assign state values to the states of the constructed eFSM as explained in Definition 6.

For example, to calculate the state value for the state $q^m_{41}$, the following steps are followed: From the domain theory of Definition 5, it can be deducted that the maximum index $j$ for which $m1 |_{Server} [j - 1]$ is a semantical cause for a message in the transitions after $q^m_{41}$ is $j = 3$ for which $m1 |_{Server} [j - 1] = Rec. msg$. For example, the message "Rec. msg" is a semantic cause for message "Update".

Therefore, it is concluded that the value of the state $q^m_{41}$ is obtained using case (ii) of Definition 6 as follows: $v_{Server} |(q^m_{41}) = m |_{Server} [4 - 1]v_{Server} |(q^m_{31})$. This becomes: $v_{Server} |(q^m_{41}) = (Rec. msg.) v_{Server} |(q^m_{31})$.

4.4.1 Detection of Emergent Behavior

After calculating the state values using the constructed domain theory according to Definition 6, the identical states are determined according to Definition 7.

4.4.1 Detection of Emergent Behavior

After calculating the state values using the constructed domain theory according to Definition 6, the identical states are determined according to Definition 7.

---

**Figure 4-14. Blending finite state machines for component Server**
Following the example presented in MSCs 1 – 3 (depicted in Figure 4-2, Figure 4-3, and Figure 4-4) the finite state machines of MSC1 and 2 are merged in Figure 4-14 where the states are assigned with the appropriate state values. Note that the values of the initial and final states are supposed to be 1.

The presence of identical states in the behavior models may result in emergent behavior since the component may be confused when performing the next message. Therefore, dealing with these issues is an important challenge in analyzing the behavior models. Once the identical states are found, the finite state machines are merged. This is done by merging the found identical states in the behavior models. In Figure 4-10 and Figure 4-12, emergent behavior occurs for component Server as a result of identical states $s_0^{m3}, s_3^{m3}$ and $s_6^{m3}$ in MSC m3 when the content of "Estimate time" messages are not considered. These identical states are then merged to remedy the occurrence of emergent behavior as shown in Figure 4-15.

![Figure 4-15. Merging the identical states of finite state machines for component Server](image)

In Figure 4-3, another example is considered where the finite state machines for MSC 2 are given. The states are then assigned by the values found based on Definition 6, and the identical states are found to be $s_0^{m4}, s_3^{m4}$ and $s_6^{m4}$ based on Definition 7. However, these identical states do not lead
to emergent behaviors according to Definition 8 since the component never gets confused in
performing the order of messages. Therefore, merging them is not necessary.

4.5 Summary

Literature indicates that detecting unwanted behaviors during the design phase is as much as 20
times cheaper than finding them during the deployment phase [1]. However, many of the existing
methodologies used to analyze system requirements and design documents require a great deal of
manual input from the system engineer and thus introduce overhead to the software development
lifecycle [3]. The goal of this research is to devise a systematic approach to analyze system
requirements for emergent behavior, while saving on overhead by replacing ad-hoc methodologies
with automated ones [3, 91]. The main issue in automating the analysis of software systems is that
each system has its own particular domain knowledge.

This research proposes a model-based framework in analysis of requirements and design of
distributed software systems. To increase efficiency, this framework employs an ontology-based
methodology to analyze system requirements expressed using scenarios. This methodology
involves the automatic construction of a domain-specific ontology of the system, which is used in
the analysis of system requirements. The greatest contributions of this methodology include
consistency and increased level of automation. Based on this methodology, a software tool (EBD)
has been developed to automate the process of requirement analysis. EBD takes system
requirements expressed using message sequence charts (MSCs) and reports possible areas of
emergent behavior.

58
Chapter Five: System Verification and Scalability

5.1 Background
5.2 Case Study: Semantic Search Engine
5.3 Methodology to Verify Lack of Emergence of a Specific Illegal Scenario
5.4 Analysis of System Scalability
5.5 Summary

Modifying the scale of software applications is in general a non-trivial endeavor, especially for larger systems because of evolution and rapid growth. Therefore, these systems face a scalability challenge due to their lack of central control as well as their rapid growth, which results in increased complexity of the system. To avoid introducing bugs into the system, the correctness and integrity of the system must be preserved after scaling. This is of great benefit as research suggests that detection of failures and removal of faults during field use of a system is about 20 times more expensive than detection and removal in the requirement and design phase [92].

This research attempts to establish a framework for analysis and validation of requirements and design documents for software systems. Thus, in addition to finding emergent behavior in a set of scenarios, the methodologies in this work can:

a) Verify the lack of emergence of a specific type of scenario in the system
b) Analyze the scalability of the system when adding new functionalities

A large prototype of a social network of MSA for semantic search is used to illustrate the developed methodologies in this Chapter.
5.1 Background

Some background knowledge with regards to scalability of social networks as well as the tie management among nodes in social networks is provided in this section.

5.1.1 Scalability of Social Networks

A simple paradigm to avoid the scalability challenge in social networks is with a fully distributed architecture of the network. It is not always possible to achieve this due to resource scarcity. There is always a trade-off between functionality and future scalability.

Several recent works in the literature try to solve the problem of social networks scalability. In [93], the authors propose a system to scale up a centralized social networks design without undergoing a costly transition to a fully distributed system. They take advantages of the structural properties of social networks to propose their paradigm. They call it One-Hop Replication (OHR). OHR uses some of structural characteristics of social networks. For instance, most information is one-hop away, and the topology of the network of connections among nodes displays a strong community structure. This system is composed of two components: first component is the controller, which is responsible for assigning users to servers. The second component is the middleware, which ensures replication consistency.

Social Partition And Replication (SPAR) is another paradigm, which is implemented in [93]. SPAR is a middleware that leverages the social graph structure to achieve data locality and minimize replication at the same time. SPAR constricts all relevant data for a user on a server and guarantees that for all users’ direct neighbors, data is co-located on the same server. In this paradigm, scalability is achieved by adding commodity servers with low memory and network I/O requirements.
In contrast, [94] introduces a decentralized scalable social network (eXO) to solve the problem of centralization in order to face the scalability challenge. eXO offers a fully decentralized social network that can efficiently index and search globally for top-k users and content based on metadata information. The architecture of eXO provides also content replication as in P2P networks. eXO is based on Distributed Hashed Table (DHT) and it adds methods on top of it for efficient indexing and search and retrieval of users and content in the social networks scenarios.

5.1.2 Managing the Ties between Nodes

The strength of a tie is affected by several factors. [95] proposed four dimensions that may affect tie strength: the duration of the relationship, the intimacy between the two actors participating in the relationship, the intensity of their communication with each other, and the reciprocal services they provide to each other. In human social networks, other factors such as socioeconomic status, educational level, political affiliation, race, and gender are also considered to affect the strength of ties [96].

Structural factors, such as network topology and information about social circles, may affect the tie strength [97]. The work in [98] suggests quantitative measures (variables) for tie strength that include intensity variable, days passed since the last communication, and duration. Another variable that may affect the strength of the tie is the neighborhood overlap variable [99], which refers to the number of common friends the two actors have. The work in [100] introduces mutual confidence between the actors of social networks. In [101] we propose a new methodology to calculate the strength of ties between agents in a social network using Hidden Markov Models (HMM) [102].

In [101] we showed that tie strength depends on several factors: Closeness factor - by measuring how close two agents are to each other (i.e. the degree of similarity between the two ontologies
used by the two agents participating in the relationship); Time-related factor - combines all time
factors that affect the strength of the relationship (e.g. duration of the relationship, frequency of
communication between the two agents, time passed since the last communication); Mutual
confidence factor - clarifying the nature of the relationship under measure, if it is a one-sided
relationship or a mutual relationship. Then we built an HMM model to measure the strengths of
ties between agents in a social network using those factors.

5.2 Case Study: Semantic Search Engine

A model for semantic search was presented in [103]. This model uses a spiral workflow to
incorporate both search and concept learning in the semantic search process [104].
Semantic search depends on understanding the meaning of the concepts used in the context of
other words. Thus it then tries to retrieve the related documents to these concepts. The foundation
of semantic search is the semantic interoperability which is the main ingredient for notation
extraction from the search phrase. Utilizing social networks in this system provides great
flexibility; in particular when dealing with concepts in ontologies. It allows MAS to understand
the meaning of the same concept even though its definition might be slightly different in each
agent’s ontology.
In our framework proposed in [103], we assume that in a society of \( n \) multi-agent systems; \( \text{MAS}_1, \)
\( \text{MAS}_2 \ldots \text{MAS}_n \), each multi-agent system, \( \text{MAS}_i \), controls a repository \( R_i \). Each repository uses an
ontology that consists of a set of concepts and some documents to represent examples of these
concepts.
The key requirements for our proposed semantic search system are given below:
1. Software agents from different MASs must be able to communicate with each other and be able to exchange information.

2. MASs must be responsible for organizing data in their own repository by annotating documents in their local repositories.

3. MASs must be able to reorganize their local repositories based on updates of concepts.

4. Agents from different MASs must be able to cooperate with each other to learn and teach new concepts.

5. MASs should be able to hide the complexity of learning process and semantic search from the user.

5.3 Methodology to Verify Lack of Emergence of a Specific Illegal Scenario

Consider the following scenarios as input for this approach. The system is initialized by a user sending a query statement to one MAS in our system. The query handler tokenizes this statement to detect concepts and it cooperates with other agents to figure out if there are any new concepts to be learned. The concept learning process is initialized when new concepts are identified. It is achieved by concept learner agent as shown in Figure 5-1.
Figure 5-1. The initialization of concept learning process based on search query entered by a user

The concept learning process itself is described in Figure 5-2. This figure shows interaction done by the learner and one teacher (B) only. This process is repeated for all detected peers.

Figure 5-2. Steps of concept learner process
As the system is developed further, the strengths of ties between the learner and all peers need to be updated. It is assumed that the strength of ties between each two nodes in the network (i.e. each MAS) depends on the following criteria (see Figure 5-3):

a) Number of times they have been able to successfully cooperate (frequency of interactions)

b) Number of peers they have in common

c) Closeness between ontologies

![Diagram showing the process of updating strengths of ties between a learner and a teacher](image)

**Figure 5-3. Updating strengths of ties between a learner and one of the teachers (Peer B)**

Figure 5-4 shows undesired behaviour, where the strength of tie between the learner and a teacher (peer D) is so low that the learner cannot depend on that teacher to learn new concepts. However, the learner did not validate the strength of the tie and send a learning request to that teacher.
5.3.1 Verification Methodology

By modifying the functionality of the social network of MAS, it is highly desirable to verify the system’s behavior. The methodology to detect emergent behavior in the requirements of this system was presented in [103, 105]. A solution to verify the system against a particular unwanted behavior, such as the one depicted in Figure 5-4, is presented in this section.

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

A. A set of MSCs containing scenarios describing the system's behavior (Figure 5-1, Figure 5-2 and Figure 5-3)

B. A set of illegal scenarios which are undesirable to occur (Figure 5-4)

By having these two sets of scenarios, this methodology verifies whether scenarios in set B can be derived from scenarios of set A. In other words, this methodology ensures that a system’s behavior does not contain scenarios from set B. This methodology is divided into two parts of constructing
the behavioral model [3, 79] and ensuring the lack of invalid scenarios in the built models, as presented in the following subsections.

5.3.1.1 Ontology Construction

The static and dynamic ontologies as defined in Chapter 3 are depicted in Figure 5-5 and Figure 5-6, respectively:

![Figure 5-5. Static ontology of the semantic search engine system](image)

![Figure 5-6. Dynamic ontology of the semantic search engine](image)
5.3.1.2 Behavior Modeling and Domain Theory Construction

The process of behavior modeling was explained in detail in Chapter 3. The behavior model for the tie-manager form the legal set of scenarios, which is set A (i.e. Figure 5-1, Figure 5-2 and Figure 5-3) is shown in Figure 5-7. The domain theory is constructed according to Definition 5, using the constructed ontologies and the comprehensive set of scenarios.

![Diagram](image)

**Figure 5-7. The union of state machines built from the legal set of scenarios**

5.3.1.3 Analysis Result

Upon the identification of cases of emergent behavior, a new set C is 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 5-7 with the behavior model constructed from set B it becomes evident that the illegal scenario of set B cannot emerge from the scenarios of set A.
5.4 Analysis of System Scalability

Consider the following set of scenarios used to demonstrate the effectiveness of the analysis of system scalability. The original functionality for tie-management in this system is described using scenarios in MSCs 1 and 2, depicted in Figure 5-8 and Figure 5-9, respectively [103]. In this case study, three arbitrary MAS of A, B and C are selected from the social network. It is assumed that ties already exist between A and C as well as between B and C; however, there are no relations between A and B. The following scenarios, expressed using MSCs, define the behavior of the network in identifying peers. It is important to note that these scenarios have been devised from the perspective of MAS_A.

For the initial requirements of this system, it was assumed that the strength of ties between each two nodes in the network (i.e. each MAS) depends on the following criteria:

a) Number of times they have been able to successfully cooperate

b) Number of peers they have in common

Figure 5-8. The tie between MAS A and B are established based on their mutual peers
Figure 5-9. The relation between MAS A and B is decided based on their ability to cooperate on a given query

This system was scaled up by increasing the functionalities of the tie-manager and adding criterion c:

c) Number of times nodes contact one another (Figure 5-10)

Figure 5-10. The tie between MAS A and B are established based on how frequently they contact each other

5.4.1 Analysis Methodology

By scaling up the functionality of the social network of MAS, it is highly desirable to verify the system’s behavior. The process of ontology and domain theory construction are not shown in this section; however, the constructed behavior model and the result of analysis are as follows [3, 79].
5.4.1.1 Behavior Modeling

The behavior model for the initial tie-management mechanism, shown in MSCs 1 and 2, is depicted in Figure 5-11 [103].

![Figure 5-11. The union of state machines built from MSCs 1 and 2](image)

To verify that the system’s integrity is preserved after adding the new functionalities, the new behavior model for the system is constructed as shown in Figure 5-12.

![Figure 5-12. Behavior model of the system after scaling](image)

5.4.1.2 Detection of Emergent Behavior

By calculating all state values, equivalent states are merged as shown in Figure 5-13 [103].

![Figure 5-13. Resulted FSM after merging identical states](image)
As shown in Figure 5-13, state S1 is where the tie manager of MAS_A falls into confusion. That is, as illustrated in MSC3 (Figure 5-10), time manager will not be able to distinguish whether it should increase the tie with MAS_B or decrease it. As a result of the systematic approach in detecting emergent behavior in MAS, the system engineers were notified of such possible scenarios and are able to make modifications where necessary.

5.5 Summary
The constant growth of size and available functionalities in large-scale software systems such as social networks makes scalability of the system vital in the design of such systems. Some of the failures in software systems can be directly attributed to their design. Research suggests that detection of failures and removal of faults during field use of a system is about 20 times more expensive than detection and removal in the requirement and design phase [92]. Unfortunately, manual review of the design documents may not efficiently detect all the design flaws due to the scale and complexity of the system. Therefore, devising an automated and systematic methodology to analyze system requirements is greatly beneficial.

This research presents a framework for the automated analysis of software requirements. The methodologies used in this framework can be applied to verify the correctness and integrity of the design of software systems after applying the changes in system requirements. These techniques were illustrated in this chapter using a large prototype of a social network of multi-agent systems for semantic search.
Traditionally, requirement elicitation and software design have been two discrete steps of the software development process, and software architecture has helped smooth the transition between the two. In modern software development with shorter release cycles, the borderline between the two steps has faded, and architectural decisions are emphasized [10].

Software engineering focuses on establishing criteria for a good design such as designing system components to allow for low coupling and high cohesion. However, the best design choices are not always obvious, particularly for more complex systems. Consequently, the quality of the system can be compromised due to overlooking details in the requirements such as the volume of message-passing between different components. Therefore, it is highly desirable to establish systematic and automated solutions to increase the quality of the system’s design while improving the efficiency of the design process. Using scenarios expressed in precise notations such as SDs or MSCs [31, 32] to describe system requirements makes systematic analysis of system requirements possible.

This thesis presents a unique solution for automating the software design process as part of a framework for the analysis of software requirements and design. This solution contributes to better design of the system architecture because it provides software engineers with a package diagram based on the system’s requirements in the early stages. In addition, software engineers can make
decisions about the non-functional requirements of the system by considering the optimal arrangements of system components. This leads to shortening the release cycle as well as savings in time and cost of the project. In the methodology outlined in Chapter 3, a solution to automatically generate the system’s static and dynamic ontology based on software requirements defined using SBS was presented [106]. The generated views of the ontology were then used to examine system requirements for possible faults and emergent behaviors.

In this chapter, a methodology is presented where the static and the dynamic views of the system are used to extract information about the association of system components with one another. This approach automates the generation of package diagrams in the software design process. A clustering algorithm is employed to define a similarity measure for packaging components or classes of the software. The similarity measure is defined to increase the cohesion and decrease the coupling between packages.

6.1 Background

Requirement elicitation is one of the most important parts of the software development life-cycle. Many bugs are introduced in the system as the result of incomplete requirements or faulty design based on incorrect interpretation of system requirements [3, 10]. There are several methodologies [2-5] that take a systematic approach to analyze the system requirements. Traditionally, the requirement elicitation and design are completed separately using an ad-hoc approach, which is error prone. The work in [10] suggests a systematic approach to bridge the gap between requirements and design to increase efficiency and decrease the risk of errors by introducing the Transition between Requirements and Architecture using a Modularized framework (TRAM).
The main elements of the system and their interactions are discovered in the requirement gathering and analysis stage. Therefore, it is highly desirable to have a systematic approach towards making design decisions. As the size of the system increases, the need to group classes into packages becomes apparent. There are many different approaches towards placing the elements into packages such as by architectural layer, by sub-system, or by use-case specifications [107]. Regardless of the approach, the criteria for good packaging of system elements is to have loosely coupled and highly cohesive packages [108]. The goal is to group interacting classes together, and also to minimize the interaction between the packages as much as possible [107].

In the context of automatic packaging, the work of [109] has previously used a clustering technique for packaging of test cases of a system. Our work, which is part of a solution for requirement analysis and design, is different from their work. We aim to package software classes rather than test documents. Also, a clustering technique and similarity measure specific for packaging of software classes are defined and used in this work.

6.2 Methodology

This thesis presents a framework for the requirement analysis. This framework receives software requirement documents in the form of scenarios expressed in SDs or MSCs and automatically extracts static and dynamic ontologies of the Software Under Development (SUD) using the approach described Chapter 3 [110]. Then, the constructed ontologies are used to detect faulty requirements by employing Label Transition System (LTS) as described in Chapter 3 [111]. When the requirements are verified, the analysis process is finished, and the verified requirements are entered into the design process. This chapter outlines the contributions of this thesis to the design process.
In the design process, software design diagrams, such as class and package diagrams need to be generated. The methodology designed in this research automates the generation of package diagrams based on the extracted static and dynamic ontology of the SUD. The software tool created based on this framework is fully explained in Chapter 7 of this thesis.

In Figure 6-1, the three main modules of the tool used in the process of generating the package diagram are introduced:

1) Static view of ontology generator
2) Dynamic view of ontology generator
3) Package diagram recommender

Definitions and the methodologies for the automatic generation of static and dynamic ontologies of the SUD are described in Chapter 3 of this thesis [110, 112]. In this section, a short overview of these concepts with respect to design are provided, and then the automatic packaging solution is described.

Figure 6-1. Main components of the EBD tool in generating package diagrams
**Static view generator** – The static ontology is used to generate knowledge about the system. The static view of ontology has a tree structure [112], where the components of the system are linked based on their hierarchy within the SUD. In the static ontology, components with similar functionalities are grouped together. For example, the static view of the software in [112] categorizes the components of the system into three main groups: Sensors, Controllers, and Motors. Therefore, this view of the system shows a higher level of the components’ functionality. In this thesis, the grouping of the static view as the first level of the packaging solution is used. An example of first level packaging based on this view is illustrated in Figure 6-5.

**Dynamic view generator** - The dynamic view of ontology represents the interactions between the system components and is constructed either manually by the domain expert, or automatically as described in Chapter 3 [110]. An example of dynamic view of ontology for a fleet-management system is shown in Chapter 4 [110]. The dynamic view of ontology in used in the packaging solution devised in this thesis as it shows the communication behavior of system components. Packaging solutions aim to assign components and classes with similar functionalities and also high communication rates to the same package. This will increase the cohesion and reduce the coupling of the packages. An example of packaging based on dynamic ontology is Figure 6-6.

**Package diagram recommender** – The devised packaging solution uses the results of static and dynamic views of ontology to automatically generate a package diagram for the SUD. The core of this packaging solution is a hierarchal clustering algorithm. The first layer of the static view of the system is used as the first packaging criteria, and then a clustering algorithm is applied within each group to construct appropriate sub-packages. A hierarchal clustering algorithm is selected because it is able to demonstrate all sub-packages. The rest of this section describes details of the approach.
A simplified example of the dynamic view is shown in Figure 6-2. Three groups of processes identified by the static view are shown using dashed boxes. The weights of the links between two processes show the number of communication between the processes. The EBD tool automatically generates this view of the system from the inputted set of scenarios.

![Figure 6-2. A simplified example of dynamic view of a system](image)

The core of the devised packaging solution is a clustering algorithm. Selecting a proper clustering algorithm for each problem is important. After reviewing the possible clustering methods, the Agglomerative Clustering Algorithm [113] was selected as the best match for this purpose because Agglomerative is a hierarchal clustering algorithm and is able to build the hierarchy from the individual elements by progressively merging the clusters. This algorithm allows for the generation of all possible cases for the nested packages.

A general agglomerative algorithm for hierarchical clustering has complexity of $O(N^3)$, and can be described as follows:

1. Determine all inter-object similarities.
Step 2 - Form cluster from the two closest objects.

Step 3 - Redefine similarities between new cluster and other objects.

Step 4 - Return to Step 2 until all objects are in one cluster.

In order to apply the algorithm on this view, a feature vector needs to be extracted for each process. Here, the feature vector of each process is defined based on its communication with other processes in the same static group. Followings are the feature vectors of the four processes related to group S2 of the static view in Figure 6-2. Each value in the feature vector of a process shows its communication size with other processes. For example, values 0, 1, 8, 2 in FVP2 correspond to the communication size of P2 with P2, P3, P4, and P5, respectively. The higher value of communication links between the processes shows the tighter dependency among the processes.

FVP2: \(<0, 1, 8, 2>\)

FVP3: \(<1, 0, 3, 10>\)

FVP4: \(<8, 3, 0, 0>\)

FVP5: \(<2, 10, 0, 0>\)

For any clustering algorithm, defining a similarity measure is crucial. Defining a similarity measure is problem-specific. This means that an appropriate measure should be defined considering the context of the problem. In this study, since the aim of the high-level packaging solutions is to categorize dependent components and classes together, the similarity measure of the clustering algorithm is defined based on the communication pattern of the process. Two processes are considered to be in a same package if there is a high number of communications between them, and they have similar communication patterns with other processes. The similarity between the two processes, \(P_i\) and \(P_j\), is defined as follows:
\[ \text{Sim} \left( P_i, P_j \right) = \text{cs}(i,j) - \sum_{k=1}^{M,k \neq i,j} |\text{cs}(i,k) - \text{cs}(j,k)| \]  (1)

where \( \text{cs}(i,j) \) is the communication size between process \( i \) and \( j \) (number of messages between \( P_i \) and \( P_j \)), and \( M \) is the total number of classes in a static group on which the clustering method is applied.

When the algorithm is applied, the result is a tree of nodes, which shows the possible sub-packages that can be used. The generated tree can be traversed from the root and based on the desired number of packages. A packaging solution can be selected by the software engineer. For example, in Figure 6-4, the tree is traversed from the top, and two different levels of clustering are selected for having 3 and 4 number of packages.

Real-life software systems are too complex for manual analysis and therefore, they can greatly benefit from using this automated approach. In the following section, the applicability of this methodology is demonstrated by applying it to a case study of an elevator control system designed in [114].

6.3 Case Study and Results

The case study selected to demonstrated the capabilities of this devised solution is an elevator control system designed in [114]. This system has the basic function of any elevator system, such as moving up and down, opening and closing doors, and picking up passengers. Figure 6-3 shows the requirements of the system in the form of sequence diagram. Thirteen scenarios of the system are inputted into the devised approach as shown Figure 6-3.
a) SD1 - Hall call service

b) SD2 - Car call service
c) SD3 - Indicating the car position

d) SD4 - Indicating the moving direction

e) SD5 - Emergency Brake - The elevator keeps going when limit is reached
f) SD6 - Door reversal

g) SD7 - Emergency Brake - The car won’t stop at desired floor

h) SD8 - Emergency brake – the car won’t move
i) SD9 - Emergency brake – the door won’t open when the elevator stops

j) SD10 - Emergency brake – the doors open when the elevator is moving

k) SD11 - Moving the car from stop to slow then to fast
Based on the static view, the first packaging level is done for the system. This guarantees that the packages are generated based on the functionalities of the system. Figure 6-5 shows the first level of packaging of the elevator control system. This is generated based on the static view of the system. In order to generate the nested packages, the proposed clustering solution is applied on each static
group. For this purpose, dynamic ontology view of the system is generated and used. Figure 6-6 shows the result of applying the packaging solution on the sender-receiver static ontology.

In the rest of this section, the details of applying the similarity measure and the clustering method for the ‘sender-receiver’ components in the static ontology view of the system were explained. The classes of this group, ‘HallButton’, ‘HallButtonController’, ‘Dispatcher’, ‘DriveControl’, ‘Drive, DoorControl’, ‘Door’, ‘CarButton’, ‘CarButtonControl’, and ‘Safety’, are shown with S1 to S10 respectively. Matrix M1 shows the extracted feature vectors for these classes.

$$M1 = \begin{bmatrix} S1 & S2 & S3 & S4 & S5 & S6 & S7 & S8 & S9 & S10 \\ S1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ S2 & 3 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ S3 & 0 & 1 & 0 & 4 & 5 & 6 & 1 & 0 & 1 \\ S4 & 0 & 0 & 4 & 0 & 2 & 2 & 0 & 2 & 0 \\ S5 & 0 & 1 & 5 & 2 & 2 & 0 & 6 & 0 & 1 & 3 \\ S6 & 0 & 0 & 6 & 0 & 6 & 0 & 1 & 9 & 0 & 0 \\ S7 & 0 & 0 & 1 & 2 & 0 & 1 & 9 & 0 & 0 & 2 \\ S8 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 3 & 0 & 0 \\ S9 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 3 & 0 & 0 \\ S10 & 0 & 0 & 0 & 0 & 0 & 3 & 0 & 2 & 0 & 0 \end{bmatrix}$$

The Similarity measure is applied on M1, and the result is shown in M2. A higher value in this matrix shows that classes that are tightly connected are candidates to be included in the same package. According to M2, S4, S5, and also S6, S7 are selected to be grouped in this step.

In the next step, these classes are merged, and according to the hierarchal clustering process, the new distances are calculated in M3. Here, S4, and S5 are replaced with C1, and S6, S7 with C2.

\[M3 = \begin{bmatrix}
S1 & S2 & S3 & C1 & C2 & S8 & S9 & S10 \\
S1 & 0 & 1 & -19 & -23 & -20 & -6 & -8 & -8 \\
S2 & 1 & 0 & -18 & -22 & -19 & -8 & -6 & -8 \\
S3 & -19 & -18 & 0 & 5 & 2 & -19 & -18 & -15 \\
C1 & -23 & -22 & 5 & 0 & 3 & -23 & -22 & -14 \\
C2 & -20 & -19 & 2 & 3 & 0 & -20 & -18 & -10 \\
S8 & -6 & -8 & -19 & -23 & -20 & 0 & 1 & -8 \\
S9 & -8 & -6 & -18 & -22 & -18 & 1 & 0 & -8 \\
S10 & -8 & -8 & -15 & -14 & -10 & -8 & -8 & 0 
\end{bmatrix}\]

In the next step, S3 will be merged with C1, as C3. Then, C2, C3 are selected to be merged. The process continues until all classes are merged in one c group. The resulting hierarchal clustering is depicted in Figure 6-4. The constructed hierarchy shows all the possible nested packages. When the hierarchy is constructed, the tree can be traversed from the root toward its leaves, and the recommended solution can be selected based on the intended number of packages. For example, for packaging the classes of the SUD to 3 packages, the solution from step four of the algorithm will be selected. This will result in three packages as follows:
Figure 6-4. Result of hierarchal clustering on classes of sender-receivers of the elevator system

P31: HallButton, HallButtonController

P32: Dispatcher, DriveControl, Drive, DoorControl, Door, Safety

P33: CarButton, CarButtonControl

Figure 6-5 shows the first layer packaging of the classes, which is done solely based on static view ontology. Three main packages are identified.
Figure 6-5. First level of packaging based on static view of the system

Figure 6-6 shows the complete packaging solution for P3, which contains all the recommended sub-packages of the P3 package in Figure 6-5. This diagram is constructed using a dynamic view of the system.

Figure 6-6. All the nested packages for P3 based on dynamic view
6.4 Summary

This research presents a unique and practical study toward automating analysis and design of software. In this chapter, a unique solution that automates the generation of package diagrams in software design process is introduced. This solution bridges the gap between requirement analysis and design by extracting an optimized design from software requirements defined using SBS. After compiling all scenarios together, this solution automatically groups system components with similar functionalities and a high degree of communication in one package, and recommends a package diagram for the system. Using the static view guarantees that components with similar functionalities group together, and the dynamic view guarantees that the more interacting components are grouped together. In large scale systems, where analyzing and finding appropriate packaging is not trivial, our approach can generate a recommended solution for software engineers to facilitate the design process of software system. This approach contributes to better understanding of how each system component contributes to the overall system architecture, which is a huge asset in the requirement analysis stage. A better understanding of the optimal system architecture can contribute to savings in time and cost of development, better planning of non-functional requirements, and shortening the software development lifecycle. Furthermore, this methodology can be extended to help with more design decisions such as recommending design patterns based on the optimal architecture.
This thesis presents a solution for the automatic analysis of software requirements. This solution involves the automatic review of software requirements to detect possible unwanted behavior as well as bridging the gap between requirement analysis and design by automatic generation of package diagrams through data mining of system requirements. The existing approaches for software analysis often require considerable input from the system engineer familiar with the domain. This framework employs an ontology-based methodology to analyze system requirements expressed by a set of scenarios. This methodology involves the automatic construction of a domain-specific ontology of the system, which is used in the analysis of system requirements. Consistency and increased level of automation are among the important advantages of this approach. The devised methodologies by this research have been developed into a software analysis tool (EBD) to automate the process of software analysis. Effectiveness of this framework is illustrated using a case study of a real time fleet management system presented in Chapter 4.

7.1 Design of the Requirement Analysis Tool: (EBD)

Upon receiving system requirements in the form of scenarios that can be expressed using MSCs or SDs, the EBD software is capable of the following:

a) Analyzing system requirements for possible emergent behavior and generating a report accordingly.
b) Generating an optimum packaging of system components

To increase the usability of the software, SDs can be constructed or imported via a variety of different software packages, such as IBM rational Rose and Microsoft Office Visio, in order to account for the convenience of the users. In this thesis the version of the software which works with Visio is demonstrated.

![Diagram of Model View Controller Architecture](image)

**Figure 7-1 Model View Controller Architecture**

The architecture of EBD reflects the use of both the Model-View-Presenter (MVP) [115] and the Model-View-Controller (MVC) [116] design patterns, though it more closely follows the MVP pattern as can be observed by the component diagram in Figure 7-2. The View can be taken as the Graphical User Interface (GUI) for this tool. Three user interfaces are currently developed: (1) Desktop Application, (2) Website, and (3) Eclipse Plugin. A variety of front ends demonstrate how this tool’s back end can be reused easily. The user interfaces are kept similar in style, with the same back end, but on different platforms to appeal to different users.
The Model is present in the package ‘Backend’ in Figure 7-3. The back end of this tool consists of the following:

1) Data structures

2) Input Parsers

3) Analysis algorithms which includes:
   a) Ontology constructor
   b) Behavior modeler
   c) Domain theory builder
   d) Emergent behavior detector
   e) Design constructor
Data structures and algorithms are the Model. The data structures provide a uniform way of storing information read in through various file formats. The Analysis, the core of the tool, can then manipulate the data and produce more data representing the desired output of the program.

Figure 7-3. Package diagram for backend of EBD

The presenter is present in two locations: (1) Parser Package and (2) Event Handlers as part of the user interface. These two together provide all communication necessary for the backend be useful to the end user. Event-handlers deal with taking in input from the user and conveying it as meaningful data and or communication with the backend, as well as using the information created or stored in the backend, and displaying it and allowing the user to manipulate it. Because each
view uses different means of communication with the end user and requires different inputs, it is kept independent of the backend and is not reusable.

On the other hand, the parser package is responsible for converting input files into data structures stored in the backend. This is included in the reusable backend package as different user interfaces may be requesting the same file inputs that require the same parsers. The important classes of the backend, the desktop application, and the web application are depicted in Figure 7-4, Figure 7-5, and Figure 7-6, respectively. The web application also contains typical functionalities such as create account, log in/out, store projects, and email results.

The use of Service Oriented Architecture (SOA) is crucial for making the development of this tool as efficient as possible. SOA aims to design software in a way that it is easily reused for a different purpose than what it was designed for. SOA uses loose coupling of services to allow for reuse by different applications [117]. SOA enables us to reuse functionality designed for a different purpose to meet our needs. Tailoring the functionality to adapt to what we require is a process that in many cases, may be much less costly than designing a system from scratch. An example of utilizing SOA by EBD is through the use of the Visio Object Model.

The Visio Object Model is designed to allow developers to automate drawing creation using an ActiveX Visio drawing box tool. It provides an interface for programmatical interpreting diagrams. We have also used this interface to create and import SDs.
Figure 7-4. Important classes for the back-end of EBD
Figure 7-5. Important classes for the desktop version of EBD
Figure 7-6. Important classes for the web version of EBD
7.2 Requirement Analysis Using EBD

Users can start the analysis by making a project and importing their scenarios. Currently, the Visio version of EBD supports all Visio programs up to and including the Office 365 version. Users can choose to import system ontologies or to simply have EBD auto-generate them as shown in Figure 7-7. Upon importing the scenarios, users can browse through them or change them as desired in the EBD window shown in Figure 7-8. The scenarios used to demonstrate the capabilities of EBD are from the fleet management system, which was presented in Chapter 4.

![Create project window for EBD](image-url)
7.2.1 Ontology Creation

As outlined in Chapter 3, EBD generates two type of system ontologies - namely static ontology and dynamic ontology. The static view of ontology is much like a tree structure, where the elements are components of the system and are related to each other in this ontology based on their hierarchy within the system. The static ontology here is used to generate knowledge about the system. The static ontology generated by EBD is shown in Figure 7-9.
To gather knowledge about the system’s behavior, we use the knowledge present in the static ontology, as well as the comprehensive set of inputted scenarios to build the dynamic ontology as outlined in Chapters 3 and 4. The final Dynamic Ontology generated by EBD is shown in Figure 7-10.
7.2.2 Behavior Modeling

As defined in detail in Chapter 3, the model that 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 system elements in scenarios, is called synthesis of behavioral models, or simply, the synthesis process [118]. 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. EBD constructs an equivalent finite state machine for each component, based on all the scenarios as shown in Figure 7-11. EBD provides a user-friendly interface where the user can browse between the constructed behavior models of each system component as shown in Figure 7-11. Moreover, the user can choose to view the merged FSM or the unionized FSM for each component (See Chapter 3 for details).
7.2.3 Creating the Domain Knowledge

After the synthesis of behavior models for each system component, the state values for the models are to be calculated based on semantic causality, which is presented in Definition 4 (See Chapter 3). Using the concept of semantic causality, the domain theory of the system is constructed according to Definition 5 given in Chapter 3.

EBD automatically constructs the system domain knowledge using the comprehensive set of inputted scenarios, as well as the static and dynamic ontologies that it automatically generated as

Figure 7-11. Behavior model for the server component in the fleet-management system generated by EBD
shown in Chapters 3, 4, and 5. As shown in Figure 7-12, EBD generates and automatically fills the tables that make the domain theory of the system as defined in Chapter 3.

![Figure 7-12. Construction of the domain knowledge](image)

### 7.2.4 Detection of Emergent Behavior

Upon building the domain theory based on semantic causality, we proceed to assign state values to the states of the constructed eFSM using Definition 6. Calculated state values are the basis for comparing states and consequently discovering identical states according to Definition 7. For each component, EBD generates a report of the identical states and the states causing emergent behavior according to Definition 8. EBD generates a report for the user outlining all areas of emergent behavior (Figure 7-13).
7.2.5 Packaging of Software Components

Our packaging solution uses the results of static and dynamic view of ontology to automatically generate a package diagram for the SUD. The core of our packaging solution is a hierarchal clustering algorithm as discussed in Chapter 6. The first layer of the static view of the system is used as the first packaging criteria, and then a clustering algorithm is applied within each group to construct appropriate sub-packages. A hierarchal clustering algorithm is selected since it is able to demonstrate all sub-packages. EBD also generates a system collaboration diagram by compiling all inputted scenarios as shown in Figure 7-14. This collaboration diagram helps system designers in the packaging process. Moreover, EBD enables users to see the collaboration diagram for each individual SD.
7.3 Summary

Literature indicates that detecting unwanted behaviors during the design phase is as much as 20 times cheaper than finding them during the deployment phase [3]. However, many of the existing methodologies used to analyze system requirements and design documents require a great deal of manual input from the system engineer and thus introduce overhead to the software development lifecycle [1]. The goal of this research is to devise a systematic approach to analyze system requirements for emergent behavior, while saving on overhead by replacing ad-hoc methodologies with automated ones [3]. The main issue in automating the analysis of software systems is that each system has its own particular domain knowledge.

This research proposes a model-based framework in analysis of requirements and design of distributed software systems. To increase efficiency, this framework employs an ontology-based
methodology to analyze system requirements expressed using scenarios. This methodology involves the automatic construction of a domain-specific ontology of the system. The constructed ontology is used to provide a comprehensive domain knowledge of the system, and is used in the analysis of system requirements that are expressed using scenarios. The greatest contributions of this methodology include consistency and increased level of automation. Based on this methodology, a software tool (EBD) has been developed to automate the process of requirement analysis. EBD takes system requirements expressed using SDs or MSCs and reports possible areas of emergent behavior. The architecture of EBD is modular, which allows for adding more algorithms and methodologies in the future.
Chapter Eight: Software Development Management using EBD

8.1 Analyzing Software Requirements to Detect Emergent Behavior

8.2. Summary

As outlined in Chapter 7, the methodologies developed in this research is developed into a software tool, EBD, to automate the process of requirement verification and analysis. The metrics of EBD is particularly evident when used to analyze larger systems and case studies that are too difficult to analyze manually. This section contains the analysis of the case study of the elevator system defined in Chapter 6 using EBD.

8.1 Analysis Software Requirements to Detect Emergent Behavior

To allow for iterative development, the requirements of the elevator system shown in Chapter 6 are analyzed in iterative steps. In this chapter we will look at the following two iterations:

- Iteration 1: SDs 1 – 7
- Iteration 2: SDs 1 – 10 (i.e. 3 SDs are added to the original set)

The dynamic ontology of the first iteration as constructed by EBD is shown in Figure 8-1. Using this ontology and that of the static ontology of the system, the domain theory of the system (Definition 5 in Chapter 3) is constructed. Then as defined in Chapter 3, the behavior model for each component is constructed. Figure 8-2 demonstrate the unionized behavior model of the for the “Door” component.
Figure 8-1. Dynamic ontology of the elevator system in iteration 1
There are no emergent behavior reported for the door component in the first iteration. However EBD reports the following pairs of states as potentially problematic: q16 and q7, and q16 and q11. This is because of the difference in wording between the two messages “Closed” and “DoorClosed” which appear after each of the aforementioned states.

After adding the SDs 8-10 to the existing scenarios and running EBD, 2 cases of emergent behavior are identified for the Door component by the EBD: states q18 and q1, and q18 and q3 (Figure 8-3). This behavior is particularly due to SDs 1 and 2 and SD 9. EBD marks the related messages in the generated report as shown in Figure 8-4.
Figure 8-3. Unionized behavior model for the Door component in iteration 2
Figure 8-4. SDs marked by EBD for being the root cause of emergent behavior in component Door
The report shown in Figure 8-4 is produced for every component in the system. As more SDs are added to the system, they can be verified against the old requirements to ensure correctness.

### 8.2 Summary

Automating the methodologies developed in this research provides a powerful tool to software professionals as well as researchers in verifying and managing software project at each iteration. Using the ontology-based approach devised in this research greatly reduces the amount of input required of the user. Therefore, using EBD in analyzing and verifying software requirements adds no significant overhead to a project, but it has great benefits.
Chapter Nine: Conclusions and Future Work

9.1 Summary and Contributions of Thesis

9.2 Future Work

Requirement elicitation is one of the most challenging stages of the software development lifecycle. Many bugs are introduced into the system as the result of incomplete or inconsistent requirements. Literature indicates that detecting unwanted behavior during the design phase is as much as 20 times cheaper than finding them during the deployment phase [91]. Unfortunately, manual review of the system requirements is time consuming and inefficient. Therefore devising automated and systematic methodologies to analyze and verify software requirements is highly desirable. However, to make systematic analysis of system requirements possible, they need to be expressed using a precise notation.

An effective and efficient way to describe system requirements is using scenario-based specifications (SBS). A scenario is a temporal sequence of messages sent between a system and actors using the system. Scenarios are appealing because of their simplicity and expressive power [2]. There are two main ways of representing scenarios, namely Sequence Diagrams (SD) developed by the Object Management Group (OMG) and Message Sequence Charts (MSC) developed by the International Telecommunication Union (ITU) [1, 32].

This thesis introduces a framework to analyze software system requirements defined using SBS. The merits of this framework are a substantial increase in level of automation and consistency, as well as reduced human intervention in comparison with other existing methodologies [3-7, 31].

9.1 Summary of Contributions of Thesis

The framework developed in this research focuses on analyzing software requirements by
c) Automating the process of modeling, analyzing, detecting and resolving emergent behavior in distributed systems [2, 8, 9, 19-29].

d) Developing a systematic and automated approach to transition from software requirements to software design [30]

The main contributions of this research are as follows:

Contribution 1) A framework to analyze and verify software system requirements with a substantially increased level of automation and consistency, as well as reduced human intervention. This framework employs an ontology-based methodology to analyze system requirements expressed by a set of scenarios. This methodology involves the automatic construction of a domain-specific ontology of the system, which is used in the analysis of system requirements.

Contribution 2) A new approach for synthesizing behavior models from scenarios while reducing the intervention of the domain expert in the synthesis process by extracting information from a defined ontology for the system. Using ontologies greatly simplifies and automates the process of emergent behavior detection.

Contribution 3) A new methodology to construct the domain-specific ontology of the system based on the complete set of software requirements expressed in scenarios.

Contribution 4) A new approach to construct a high-level package diagram and a system-level communication diagram of the entire system based on the constructed
ontology and the complete set of software requirements expressed in scenarios. These diagrams can be used to guide system architects and engineers to transition from the requirement analysis to the design stage. This systematic approach leads to increasing the quality of software as well as savings in resources.

Contribution 5) The framework created in this research has been developed into a software tool to apply the proposed methodologies to requirements and design documents of a variety of software systems such as distributed and multi-agent systems.

Contribution 6) Applying this framework to a wide range of software systems to demonstrate the efficiency and effectiveness of this approach as well as testing the usability of the EBD software. These systems include information retrieval system, MAS, social networks, network security systems, and engineering and manufacturing distributed systems [8, 9, 15, 19-29].

At the heart of this framework is an ontology-based methodology that analyzes software requirements defined using Scenario-Based Specifications (SBS). This methodology involves the automatic construction of a domain-specific ontology of the system, which is used in the analysis of system requirements.

Chapter 2 provides an account of existing methodologies in the literature [3-7, 30]. The merits and disadvantages of each methodology were explained; the work of [2] was selected as the basis to analyze scenario-based specifications due to its consistency of assigning state values. Thus, the methodologies of [4] has been used as the starting point of this research. Although the approach
of [4] is systematic, it still requires a great deal of input from the user with regards to the domain knowledge in order to commence analysis.

In Chapter 3, the proposed ontology-based methodology to detect emergent behavior is defined and its differences with the work of [4] are outlined and Contributions 1-3 are demonstrated. In Chapter 4, the ontology-based methodology is used to analyze the requirements for a real-time fleet-management system in order to detect emergent behavior. In Chapter 5, it is demonstrated that in addition to finding emergent behavior in a set of scenarios, the methodologies in this work can:

   c) Verify the lack of emergence of a specific type of scenario in the system

   d) Analyze the scalability of the system when adding new functionalities

A large prototype of a social network of MSA for semantic search is used to illustrate the effectiveness of this approach. In both Chapters 4 and 5, Contributions 1-3 and 6 are realized.

Chapter 6 introduces a unique solution towards automating the software design process as part of a framework for the analysis of software requirements and design. This solution contributes to better design of the system architecture because it provides software engineers with a package diagram based on the system requirements in the early stages. Furthermore software engineers can make decisions about the non-functional requirements of the system by considering the optimal arrangements of system components. This leads to shortening the release cycle as well as savings in time and cost of the project. This methodology is described using the case study of an elevator system. Contributions 1, 5, and 6 are realized in this Chapter.
The devised methodologies by this research have been implemented into a software analysis tool, EBD, to automate the methodologies of the presented framework. This work, which is outlined in Chapter 7 aligns with Contributions 1 and 5. Chapter 8 demonstrates how a software development project can be managed and verified at each iteration with no significant added overhead using the EBD software which also aligns with Contributions 1 and 5.

9.2 Future Work

Future work may be to implement the proposed algorithm as a syntax checker to provide an automated tool to check and correct system designs. These techniques can also be extended to verify the design of multi-agent systems. In [28], this methodology was used to analyze MAS design artifacts produced using MaSE (Multi-agent Software Engineering). Further research can be done in this area, such as devising algorithms to analyze design artifacts of other Agent Oriented Software Engineering (AOSE) methodologies such as GAIA.

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 [4]. Contributions regarding this task involve: (1) revisiting the notion of safe realizability for MSC specifications [4] 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” [4]); 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.
References


[38] *Casual Closure for MSC Languages* 2005.


